Orthogonal Graph Drawing with Constraints: Algorithms and Applications

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Martin Siebenhaller
aus Meersburg

Tübingen
2009

Tag der mündlichen Qualifikation:	16.12.2009
Dekan:	Prof. Dr.-Ing. Oliver Kohlbacher
1. Berichterstatter:	Prof. Dr. Michael Kaufmann
2. Berichterstatter:	Prof. Dr. Wolfgang Küchlin

Acknowledgments

Writing this thesis would not have been possible without the great support of various people. First of all I want to thank my advisor Prof. Dr. Michael Kaufmann for all his support, for introducing me to the interesting field of graph drawing, for giving me the opportunity to be part of his research group and for providing me the possibility to attend several international workshops and conferences. I am also very grateful to Dr. Markus Eiglsperger for guiding me through my first steps of designing and implementing graph drawing algorithms. I would like to thank the Deutsche Forschungsgemeinschaft (DFG) for financial support of our research under grant Ka 812/8-3 and the yWorks company for providing both the yFiles library and their technical support. Further, I want to thank all my co-authors for the successful cooperation as well as the rest of the research group, namely Philip Effinger, Andreas Gerasch, Markus Geyer, Stephan Kottler and Dr. Katharina Zweig, for providing a pleasant working environment and for relaxing coffee breaks. Special thanks also to the proofreaders Marc Bégin, Philip Effinger, Stephan Kottler and Thomas Wurst. Last but not least, I want to thank my parents for supporting my studies.

Deutsche Zusammenfassung

Die Visualisierung ist ein bewährtes und probates Mittel für die Repräsentation von Daten. Aufgrund der kognitiven Fähigkeiten von Menschen erleichtert eine Visualisierung (graphische Darstellung) das Aufnehmen und Verstehen der in den Daten enthaltenen Informationen. Die vorliegende Arbeit beschäftigt sich mit der Graph-basierten Visualisierung, d. h. der Visualisierung von Daten, die durch Graphen beschrieben werden können.

Ein Graph ist ein Konstrukt der Mathematik (Graphentheorie) und wird zur Modellierung von binären Relationen zwischen Objekten verwendet. Die Objekte werden dabei häufig als „Knoten“ und die Relationen als „Kanten“ bezeichnet. Das Forschungsgebiet, welches sich mit der Visualisierung von Graphen, im Speziellen mit dem automatischen Anordnen (Layout) von Knoten und Kanten beschäftigt, heißt „Graphenzeichnen“. Beim Zeichnen von Graphen werden Knoten üblicherweise als Punkte/Rechtecke und Kanten als Kurven repräsentiert. Außerdem werden verschiedene ästhetische Anforderungen berücksichtigt, z. B. soll die Anzahl der Kantenkreuzungen in der resultierenden Zeichnung möglichst gering sein.

In orthogonalen Zeichnungen von Graphen werden die Kanten durch Polygonzüge dargestellt, die aus abwechselnd horizontalen und vertikalen Liniensegmenten bestehen. Das bekannteste Verfahren zur Erzeugung von orthogonalen Zeichnungen ist der „Topology-Shape-Metrics“ Ansatz (TSM-Ansatz), der aus den Phasen Planarisierung, Orthogonalisierung und Kompaktierung besteht. Im Gegensatz zu den sogenannten kräftebasierten und hierarchischen Zeichenverfahren, wird der TSM-Ansatz in der Praxis nur selten eingesetzt. Das liegt zum einen an dessen höherem Implementierungsaufwand, zum anderen daran, dass im Vergleich zu den beiden anderen Verfahren nur wenige Erweiterungen zur Einbeziehung von Nebenbedingungen bezüglich der Anordnung der Graphelemente bekannt sind.

Die Formulierung solcher Nebenbedingungen ist notwendig, um geeignete Visualisierungen für Anwendungen aus Bereichen wie Software-Technik, Bioinformatik und Netzwerkanalyse zu erzeugen. In der vorliegenden Arbeit wird ein neues Zeichenverfahren präsentiert, welches auf dem TSM-Ansatz basiert und die folgenden Nebenbedingungen unterstützt:

- Eine Teilmenge der Kanten wird durch monoton steigende Kurven repräsentiert. Diese Kanten besitzen also eine einheitliche Flussrichtung.
- Eine Teilmenge der Knoten wird „bimodal“ gezeichnet, d. h. so, dass die eingehenden und ausgehenden Kanten eines Knotens, bezüglich ihrer Reihenfolge um den Knoten, getrennt voneinander erscheinen.
- Zusammengehörige Knoten werden zu Clustern gruppiert. Jeder Cluster wird durch eine rechteckige Region repräsentiert, welche genau die zum Cluster gehörenden Knoten beinhaltet. Die Position eines Clusters wird dabei nicht vorgegeben.
- Die Zeichenfläche wird in Rechtecke zerlegt. Jeder Knoten wird innerhalb eines vorgegebenen Rechtecks platziert.
- Die möglichen Anschlusspunkte einer Kante an einem dazugehörigen Knoten werden durch die Vorgabe einer Seite oder einer genauen Position am Knotenrand eingeschränkt.

Bislang ist kein TSM-basiertes Verfahren bekannt, dass eine solch komplexe Kombination von Nebenbedingungen zulässt. Für die Entwicklung der entsprechenden Algorithmen wird auf bewährte Methoden und Konzepte aus dem Bereich des Graphenzeichnens zurückgegriffen. Wie der Titel der Arbeit bereits verrät, werden nicht nur neue Algorithmen präsentiert, sondern diese auch im praktischen Einsatz getestet. Die erzielten Ergebnisse bestätigen, dass das Verfahren gut für die Visualisierung praktischer Anwendungen geeignet ist.

Im Einzelnen gliedert sich die Arbeit wie folgt: Nach einer kurzen Einführung in die Thematik werden im zweiten Kapitel zunächst grundlegende mathematische Konzepte und Definitionen aus dem Bereich der Graphentheorie erläutert. Danach wird ein Überblick über verschiedene Anforderungen und Konventionen, die beim Zeichnen von Graphen von Bedeutung sind, gegeben. Es werden die einzelnen Phasen des TSM-Ansatzes beschrieben und das hierarchische Zeichenverfahren von Sugiyama (Sugiyama-Verfahren) vorgestellt.

Im dritten Kapitel werden die oben genannten Nebenbedingungen analysiert und ein Überblick über bereits bestehende verwandte theoretische und praktische Arbeiten gegeben. Nach der Vorstellung der verschiedenen Nebenbedingungen werden die Probleme, die durch deren gleichzeitige Verwendung entstehen können, betrachtet. Außerdem wird ein Überblick über die einzelnen Schritte des neuen Zeichenverfahrens gegeben und dessen Schnittstelle beschrieben.

Im vierten Kapitel wird das neue Planarisierungsverfahren eingeführt, welches die fünf Nebenbedingungen einbindet. Dabei wird auch eine besonders effiziente Implementierung des Sugiyama-Verfahrens vorgestellt, welche

den existierenden Implementierungen bezüglich der Laufzeit und des Speicherverbrauchs weit überlegen ist. Die entwickelte Planarisierungsphase verwendet diese Implementierung in einem Zwischenschritt.

In Kapitel fünf wird ein Orthogonalisierungsverfahren vorgestellt, welches die Nebenbedingungen berücksichtigt. Die Orthogonalisierung basiert auf dem bekannten „Kandinsky"-Verfahren und dessen Erweiterungen, welche das Vorgeben von Kantenknicken und Winkeln zwischen Kanten ermöglichen. Es wird aufgezeigt, wie die verschiedenen Nebenbedingungen mithilfe dieser Erweiterungen realisiert werden können.

Im sechsten Kapitel werden alternative Planarisierungs- und Orthogonalisierungsverfahren für die Modellierung von Nebenbedingungen, welche die möglichen Anschlusspunkte der Kanten an Knoten einschränken, beschrieben. Dabei wird auch ein orthogonales Zeichenverfahren vorgestellt, welches nicht auf dem TSM-Ansatz, sondern einer anderen Methode beruht.

Die Tauglichkeit der in dieser Arbeit vorgestellten Algorithmen und Methoden wird im siebten Kapitel demonstriert. Zuerst wird das neue Verfahren zum Zeichnen von UML-Aktivitätsdiagrammen eingesetzt. Dazu werden die entsprechenden Anforderungen ermittelt, ein Überblick über bestehende verwandte Arbeiten gegeben und die Resultate der in bekannten UML-Werkzeugen verwendeten Zeichenverfahren untersucht. Die Qualität des Verfahrens wird anhand verschiedener Beispieldiagramme belegt. Im zweiten Teil des Kapitels wird eine Visualisierungsmethode für Ausführungsgraphen von parallelen Programmabläufen präsentiert. Diese Methode basiert auf der schnellen Implementierung des Sugiyama-Verfahrens. Wieder werden die entsprechenden Anforderungen ermittelt und verwandte Arbeiten auf diesem Gebiet vorgestellt. Die Visualisierungsmethode wird mittels einer repräsentativen Beispielsanwendung veranschaulicht. Das Kapitel wird mit einer experimentellen Untersuchung abgeschlossen, welche die Qualität und die Laufzeit der vorgestellten Algorithmen analysiert.

Die Arbeit endet mit einer Zusammenfassung der wichtigsten Ergebnisse und einem Ausblick über mögliche Weiterentwicklungen.

Contents

1 Introduction **1**

2 Basics of Graph Drawing **7**

2.1 Basic Terms and Concepts 7

2.1.1 Graphs 7

2.1.2 Drawings and Planarity 9

2.2 Requirements of Drawings 11

2.2.1 Drawing Conventions 12

2.2.2 Drawing Aesthetics 12

2.2.3 Drawing Constraints 13

2.3 The Topology-Shape-Metrics (TSM) Approach 14

2.3.1 Planarization 15

2.3.2 Orthogonalization 19

2.3.3 Compaction 23

2.4 Sugiyama's Approach 25

2.4.1 Cycle Removal 27

2.4.2 Layer Assignment and Normalization 29

2.4.3 Crossing Reduction 30

2.4.4 Horizontal Coordinate Assignment 31

3 Orthogonal Graph Drawing with Constraints **35**

3.1 Drawing Constraints 35

3.1.1 Bimodal Drawings (BIMODAL) 35

3.1.2 (Mixed) Upward Drawings (FLOW) 37

3.1.3 Cluster Drawings (CLUSTER) 38

3.1.4 Partitioned Drawings (PARTITION) 42

3.1.5 Port/Side Preserving Drawings (PORT/SIDE) 44

3.2 Combining Drawing Constraints 47

3.2.1 Drawing Compatibilities 48

3.2.2 Related Work 50

3.3 Constraint-Kandinsky 52

3.3.1 Input and Interface 53

3.3.2 Overview 55

4 Planarization with Constraints 59
4.1 The Orientation Problem 59
4.2 Mixed Upward Planarization 61
4.2.1 Construction of an Initial Drawing 63
4.2.2 Construction of a Planar Embedding 66
4.2.3 Rerouting of Edges 67
4.3 Bimodal Planarization 68
4.4 Planarization of Clusters and Partitions 69
4.4.1 Construction of an Initial Drawing 69
4.4.2 Construction of a Planar Embedding 76
4.4.3 Rerouting of Edges 78
4.4.4 Optimal Swimlane Order 80
4.5 Including Port and Side Constraints 82
4.5.1 Construction of an Initial Drawing 82
4.5.2 Remaining Steps 89
4.6 Fast Implementation of Sugiyama's Approach 89
4.6.1 Basic Idea 90
4.6.2 Efficient Crossing Reduction 93
4.6.3 An Efficient Data Structure 98
4.6.4 Runtime and Space Complexity 101
4.6.5 Including Drawing Constraints 101

5 Orthogonalization with Constraints 105
5.1 Tamassia's Approach 105
5.2 The Kandinsky Network 107
5.2.1 Basic Network Formulation 107
5.2.2 Incorporating Prescribed Angles and Bends 109
5.3 Incorporating Constraints 111
5.4 Adding Port and Side Constraints 114
5.5 Handling of Self-Loops 117
5.6 Placement of Labels 118

6 Alternatives for Realizing Port/Side Constraints 119
6.1 An Alternative Drawing Method 119
6.1.1 The `Odevs` Model 120
6.1.2 Incorporating Port/Side Constraints 125
6.2 Alternative Planarization Approaches 135
6.2.1 Successive Planarity Testing 135
6.2.2 Spanning Tree-Based Planarization 136
6.2.3 GT-Based Planarization 136
6.3 Alternative Orthogonalization Approaches 137
6.3.1 An Integer Linear Program Formulation 137
6.3.2 Orthogonalization Without Fixing a Skeleton 139
6.4 Additional Requirements 141

7 Usability Study 143
7.1 Visualization of Activity Diagrams 143
7.1.1 Layout Requirements 144
7.1.2 Related Work 147
7.1.3 Layout Capabilities of UML Tools 148
7.1.4 Applying Constraint-Kandinsky 151
7.1.5 Examples 151
7.2 Visualization of Parallel Computations 154
7.2.1 A Graph-Theoretic Model for Parallel Computations . 154
7.2.2 Requirements for Visualizing Parallel Computations . 156
7.2.3 Related Work 158
7.2.4 A Layout Algorithm for Execution Graphs 159
7.2.5 Tool Integration 163
7.2.6 Analysis Methodology and Usability Example 165
7.3 Experiments 169
7.3.1 Data and Experimental Setting 169
7.3.2 Test Results for Fast-Sugiyama 170
7.3.3 Test Results for Constraint-Kandinsky 172

8 Conclusion 191
8.1 Results 192
8.2 Directions for Future Research 193

Graph drawing is the best possible field I can think of: It merges aesthetics, mathematical beauty and wonderful algorithms. It therefore provides a harmonic balance between the left and right brain parts.

Donald E. Knuth

Chapter 1
Introduction

In the last two decades, with the emergence of computer graphics, the research field of *information visualization* has become increasingly important. Information visualization utilizes graphical techniques to support people in understanding and analyzing the information content of data. The importance of visualization is already pointed out in the proverb "a picture is worth a thousand words". Due to the capabilities of the human visual system, data represented in (two-dimensional) visual form can be better recognized and understood than data represented in textual or mathematical (one-dimensional) form. Visualization especially reveals information about topological and geometric relations. For example, properties of mathematical functions like symmetry, critical points or inflection points are readily identifiable in function graphs but not in mathematical formulas or value tables.

In this work we consider graph-based visualizations, i.e., the visualization of data which can be described by *graphs*. A graph is a mathematical construct that consists of a set of objects (called vertices) and a set of binary relations between these objects (called edges). There are various structures and problems that can be mapped to graphs. *Graph drawing*, as a branch of graph theory, deals with the design and implementation of *automatic layout* algorithms for generating readable drawings (diagrams) of graphs. Graphs are usually drawn on a plane using points or rectangles to represent vertices and lines to represent edges. Automatic graph layout is motivated by applications such as software engineering, social network analysis, bioinformatics and many more. Two example applications are given in Fig. 1.1.

(a) UML class diagram

(b) Database schema

Figure 1.1: (a) shows a UML class diagram taken from [67]. Red edges denote generalization relations, which are drawn in upward direction and in a hyperedge representation. (b) presents a database schema drawn with the "MySQL Explorer" tool from Toolmagic Software (http://www.sqltoolkit.com). Note that edges are attached to specific attributes on the rectangles representing database tables.

A central problem in graph drawing is how to produce readable drawings, i.e., diagrams that best reveal the information contained in the underlying data. In order to produce such drawings, we have to identify criteria that determine if a drawing is "good" or "bad". Besides general aesthetic criteria like the number of bends or crossings of edges, there are also criteria depending on the semantics and structure of the data.

There are several reasons why we need automatic layout approaches for graphs: First of all, graphs are often generated automatically; e.g., UML class diagrams are often generated from existing source code using a reverse engineering tool. Moreover, there are several applications that use log files to generate a graph-based model that has to be visualized. Of course, the layout can be generated manually, but obviously this does not scale very well with increasing graph size. Drawing large graphs by hand is a very time-consuming and exhausting task. In practice there are several applications where automatic events trigger changes in the graph structure. With an automatic layout approach, the corresponding diagrams can be updated immediately, which guarantees synchronized views. Another point is that automatic layout algorithms offer different views of the underlying data by changing the included layout criteria. Automatic layout also facilitates conformance with given style guidelines, which improves communication among users.

The most established drawing approaches for general graphs which have received a lot of attention in the graph drawing community are the force-directed approach, the layered approach (also known as the hierarchical approach) as well as the *topology-shape-metrics approach (TSM approach)*. While the first two approaches are very popular in practice and there exist various graph drawing tools supporting them, the TSM approach has never gained this attention there. While all three approaches produce fairly good layout results, the force-directed and layered approaches additionally support several drawing constraints and are easier to implement.

In [64] Eiglsperger showed that the TSM approach can still be applied successfully to complex real-world applications, i.e., to the layout of UML-class diagrams. This thesis builds on results described in that work and demonstrates how to include various drawing constraints into the TSM approach. The TSM approach is based on the *orthogonal drawing* paradigm, in which all edges are represented by an alternating sequence of horizontal and vertical line segments. It consists of three phases: planarization, orthogonalization and compaction.

Drawing constraints specify additional requirements for a drawing and are given as additional input to the layout algorithm. In this work, we consider the following five drawing constraints which arise in diagrams used in several practical applications like software engineering, database modeling and VLSI (very large-scale integration) design. Handling these constraints is very important for producing adequate visualizations for such applications.

- **FLOW:** Edges of a given set are represented by monotonically increasing curves. Fig. 1.1(a) gives an example.
- **BIMODAL:** Incoming edges (and thus outgoing edges) are consecutive with respect to the circular edge order around vertices.
- **CLUSTER:** Given subsets of vertices are placed inside rectangular cluster regions.
- **PARTITION:** Each vertex is placed inside a predetermined partition cell of a rectangular partitioned drawing area.
- **PORT/SIDE:** Port constraints specify the exact location (pin) where an edge should leave/enter its incident vertices, while side constraints specify on which side of its incident vertices an edge should leave/enter. An application using port constraints is shown in Fig. 1.1(b).

While for some of the above constraints there are already planarization and orthogonalization approaches, there is still no approach which allows one to combine several of these constraints. In this work, we present for the first time an (automatic) orthogonal drawing approach called `Constraint-Kandinsky` which is based on the TSM approach and is able to include all of the aforementioned constraints at the same time. Therefore, we adapt several sophisticated graph drawing methods and algorithms. As the title of this work suggests, we do not only present algorithms but also demonstrate their efficiency on real-world applications.

This work is structured as follows: Chapter 2 provides the basic definitions and mathematical concepts used in this work. We give an overview of different requirements for drawing graphs and review the TSM approach. Furthermore, we describe the layered drawing approach of Sugiyama, which is used as an intermediate step in our visualization.

In Chapter 3 we introduce the five different drawing constraints in more detail. In addition to theoretical results, we will also state relevant practical approaches done so far. After presenting the single constraints, we take a look at issues arising when we combine them. We present the interface of our new approach and give an overview of its single steps.

Chapter 4 presents our new planarization framework, which is capable of simultaneously including the five aforementioned drawing constraints. After introducing the so-called "orientation problem", we describe the basic concept behind our planarization approach. We also present a fast implementation of Sugiyama's approach, which provides a significant runtime improvement over existing implementations.

In Chapter 5 we show how to perform the orthogonalization phase for the different constraints. Therefore, we review the popular network flow-based approach of Tamassia [147], which computes bend-minimum orthogonal drawings for plane 4-graphs with given embedding. We also present

the related Kandinsky approach [78] and its extensions for handling prescribed edge bends as well as prescribed angles between adjacent edges [26]. Afterwards we illustrate how to incorporate our set of constraints into this approach. At the end of this chapter, we briefly sketch how to place labels on graph elements.

In Chapter 6 we present and analyze different alternative approaches for realizing port/side constraints. We describe a fast orthogonal drawing approach which implements the so-called three-phase method [13] instead of the TSM approach. Additionally, we present different alternative planarization approaches as well as two alternative orthogonalization approaches which produce port/side constraint preserving drawings.

In Chapter 7 we investigate the usability and performance of the methods and algorithms presented in this work. First we apply our layout approach to UML activity diagrams. To this end, we identify the corresponding requirements, and give an overview of related work as well as a brief evaluation of layout capabilities of different popular UML tools. We also provide some layout examples to demonstrate the quality of our approach. In the second part, we introduce our visualization method for execution graphs of parallel computations, which is based on our fast implementation of Sugiyama's algorithm. Again we identify the corresponding requirements and take a look at related work done in this area. Furthermore, we demonstrate our visualization methodology using a representative example application. At the end of this chapter we give an experimental evaluation of the runtime and quality of different algorithms presented in this work.

The content and contributions of this thesis are reviewed in Chapter 8. Some of the presented results have already been published in [14, 15, 67, 71, 135, 136, 137, 138].

CHAPTER 2

Basics of Graph Drawing

In this chapter we introduce the basic definitions and algorithms which provide the basis for this work. First, we review relevant terms and mathematical concepts. Afterwards, we state different requirements for drawings of graphs. Our new visualization approach is based on the topology-shape-metrics approach, which we present in Section 2.3. In the last section, we describe the layered drawing approach of Sugiyama, which we use as an intermediate step during our visualization.

2.1 Basic Terms and Concepts

In the following we take a look at the mathematical concepts used in this work – graphs, drawings of graphs and planarity.

2.1.1 Graphs

Below, we present basic terms and concepts of graph theory. For a more comprehensive overview of this topic, we refer the reader to [37, 51].

A *graph* is an ordered pair $G = (V, E)$ which consists of a set of *vertices* V and a set of *edges* $E \subseteq V \times V$. The vertices of an edge e are called the *endpoints* of e. If all vertex pairs in E are ordered (directed), we call G a *directed graph*, and if all pairs are unordered we call it an *undirected graph*. Two vertices are called *adjacent* if they are connected by an edge and two edges are adjacent if they share an endpoint. If v is an endpoint of edge e, v and e are called *incident*. For a directed edge $e = (v, w)$ we call v the *source* and w the *target* of e. A *self-loop* is an edge that starts and ends at the same vertex. If there are multiple edges between a pair of vertices, those edges are called *multi-edges*. A *simple graph* is a graph that neither contains self-loops nor multi-edges.

A graph $G' = (V', E')$ is called a *subgraph* of graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. G' is called an *induced subgraph* of G on V' if $V' \subseteq V$ and for any pair of vertices $v, w \in V'$ is $(v, w) \in E'$ if and only if $(v, w) \in E$. An undirected graph $G = (V, E)$ is called *bipartite* if V can be partitioned into two disjoint vertex sets $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$ such that $E \subseteq V_1 \times V_2$.

A simple *path* between vertex v and w is a vertex sequence $(v = u_1, u_2, \dots, u_k = w)$, such that $u_1, \dots, u_k \in V$, $(u_i, u_{i+1}) \in E$, $1 \le i \le k-1$ and $u_i \ne u_j$, $1 \le i \ne j \le k$. It is denoted by $v \to_G^* w$. A *cycle* is a non-empty path with $u_1 = u_k$, $k > 1$. If a graph contains no cycles it is called *acyclic*. An undirected graph is called *connected* if there is a path between every pair of vertices. A directed graph is connected if its undirected version is connected.

The *degree* $\delta_G(v)$ of a vertex $v \in V$ is the number of edges incident to v. If G is directed, $\delta_G(v)$ can be divided into the *out-degree* $\delta_G^+(v) = |\{w \mid (v,w) \in E\}|$ and the *in-degree* $\delta_G^-(v) = |\{w \mid (w,v) \in E\}|$. A vertex $v \in V$ with $\delta_G^-(v) = 0$ is called a *source* of G and a vertex $v \in V$ with $\delta_G^+(v) = 0$ a *sink*. A directed acyclic graph G is called an *st-graph* if it has exactly one source and one sink.

A *tree* is an undirected, acyclic and connected graph ($\Rightarrow |E| = |V| - 1$). In a *rooted tree* T one vertex is designated as the root. If the order of the subtrees of T is significant it is called an *ordered tree*. The *depth* of a vertex v in T is the length of the path from the root to v. The *lowest common ancestor* of two vertices v and w is defined as the deepest vertex in T that has both v and w as descendants (where we allow a vertex to be a descendant of itself). A *spanning tree* of a connected graph G is a subgraph of G that is a tree and contains all vertices of G. If G has a weight function on the edges, a spanning tree is called a *minimum spanning tree*, if the sum of weights of its edges is minimum.

Let $\mathcal{N} = (U, A)$ denote a directed graph with given cost function $c\colon A \to \mathbb{N}$ and capacity function $u\colon A \to \mathbb{N}$ on the edges. Furthermore, for each vertex $v \in U$ the function $b\colon U \to \mathbb{Z}$ gives the supply (if $b(v) > 0$) or demand (if $b(v) < 0$) of v. A graph $\mathcal{N}$ with these functions is called a *flow network*. A *minimum cost flow problem* on $\mathcal{N}$ can be stated as follows:

$$\text{minimize} \sum_{e \in A} c(e) f(e)$$

subject to

$$\begin{aligned} \sum_{e=(v,w)\in A} f(e) - \sum_{e'=(w,v)\in A} f(e') &= b(v) \qquad \forall\, v \in U \\ 0 \le f(e) &\le c(e) \qquad \forall\, e \in A \end{aligned}$$

The function $f\colon A \to \mathbb{N}$ is called a *flow*. A flow f is called *feasible* if it satisfies all of the above equations and inequalities. In order to be feasible a minimum cost flow must satisfy $\sum_{v \in U} b(v) = 0$. A comprehensive overview of minimum cost flow networks can be found in [1].

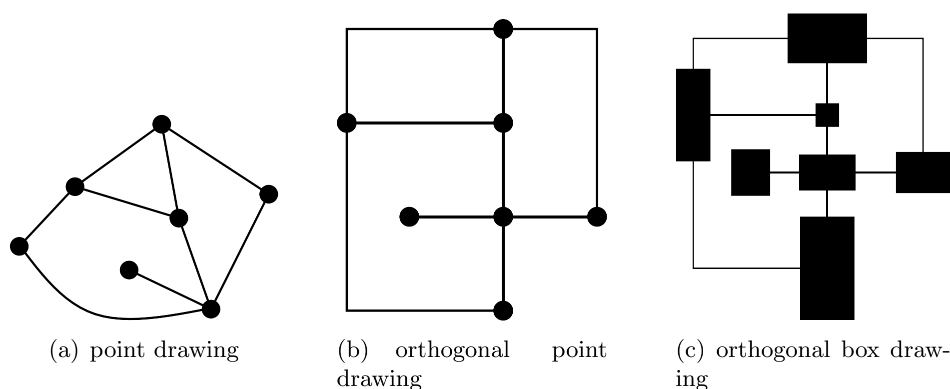
A *topological ordering* of a directed acyclic graph $G = (V, E)$ is a linear ordering $\pi\colon V \to \mathbb{N}$ of its vertices such that $\pi(v) < \pi(w)\ \forall (v,w) \in E$. A

well-known result is that a directed graph is acyclic if and only if it has a topological ordering.

2.1.2 Drawings and Planarity

In the following we give relevant terms and definitions in the area of graph drawing and planarity. For a more detailed introduction refer to [46, 103].

A *point drawing* of a graph $G = (V, E)$ is a mapping of the vertex set V to distinct points in the plane and the edge set E to open Jordan curves. The curve of an edge (v, w) connects the points that represent vertices v and w. In a *box drawing* the vertices are mapped to boxes (rectangles) instead of points. A point drawing is called an *orthogonal drawing* if the curve of each edge is represented by an alternating sequence of horizontal and vertical line segments. If, furthermore, all vertices and bends along the edges have integer coordinates, the drawing is called an *orthogonal grid drawing*. Note that a graph has an orthogonal grid drawing if and only if it is a 4-graph (that is, each vertex has a degree of at most four). A drawing is called an *orthogonal box drawing* if it is an orthogonal drawing and each vertex is mapped to a box. In an *orthogonal box grid drawing*, the center of the boxes and the edges' bends have integer coordinates. Examples of different drawing styles are given in Fig. 2.1.



(a) point drawing (b) orthogonal point drawing (c) orthogonal box drawing

Figure 2.1: Different drawings of the same graph.

A drawing of a graph is called *planar* if it has no edge crossings, that is, no two Jordan curves representing edges intersect except at common endpoints. A graph is called planar, if it has a planar drawing. A planar drawing partitions the plane into regions called *faces* (Fig. 2.3(a)). There is exactly one unbounded region which is called the *outer face*.

An *embedding* of a graph is given by the clockwise cyclic ordering of the incident edges around each vertex. An embedding is called planar if there is a planar drawing of the graph which preserves this ordering. Planarity testing of a graph can be done in linear time [21, 99]. Note that a planar

graph may have an exponential number of different planar embeddings. A famous theorem about planar graphs was discovered by Euler around 1750:

Theorem 2.1 (Euler's Formula) *Let $G = (V, E)$ denote a connected, planar graph and F the set of faces. Then*

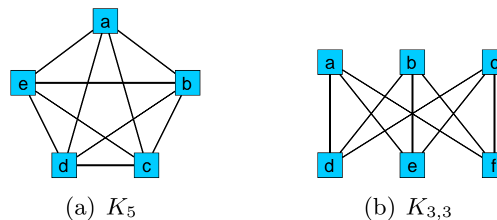
$$|V| - |E| + |F| = 2.$$

Another important result based on this fact is the following:

Corollary 2.2 *A planar graph $G = (V, E)$ with $|V| \geq 3$ has at most $3|V| - 6$ edges.*

A *subdivision* of an edge $e = (v, w)$ in graph $G = (V, E)$ can be obtained by adding a vertex u to V and replacing e by two edges $e_1 = (v, u)$ and $e_2 = (u, w)$. A graph $G' = (V', E')$ is called a subdivision of graph $G = (V, E)$ if G' can be obtained by a series of subdivisions of edges of E.

A graph is called a *complete graph* if each pair of vertices is connected by an edge. The complete graph with n vertices is denoted as K_n. Analogously, in a *complete bipartite graph* $G = (V_1 \cup V_2, E)$, every vertex of V_1 is connected to every vertex of V_2 (thus $E = V_1 \times V_2$). A complete bipartite graph is denoted as $K_{p,q}$ where $p = |V_1|$ and $q = |V_2|$. Fig. 2.2 shows example drawings of the complete graphs K_5 and $K_{3,3}$.



(a) K_5 (b) $K_{3,3}$

Figure 2.2: The complete graph with 5 vertices (K_5) and the complete bipartite graph with 3 vertices in each set ($K_{3,3}$).

The following characterization for planar graphs was given by Kuratowski:

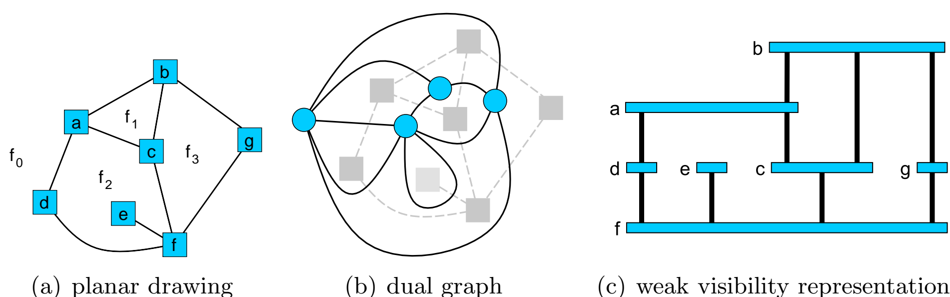
Theorem 2.3 (Kuratowski's Theorem [108]) *A graph G is planar if and only if it does not contain a subdivision of $K_{3,3}$ or K_5 as subgraph.*

A planar subgraph $G' = (V, E')$ of a graph $G = (V, E)$ is called a *maximum planar subgraph* if there is no other planar subgraph of G having more edges. In [83] it is shown that the calculation of a maximum planar subgraph is NP-hard.

In a *weak visibility representation* of a planar graph $G = (V, E)$, each vertex $v \in V$ is mapped to a horizontal segment and each edge $e \in E$

to a vertical segment. Furthermore, the vertical segment representing an edge (v, w) has its endpoints on the horizontal segments representing v and w, and does not intersect with any other horizontal segment. An example is given in Fig. 2.3(c). A linear time algorithm for constructing such a representation for a 2-connected planar graph was given in [119]. In [52] it was independently shown that every planar graph admits a weak visibility representation.

The *dual graph* D_G of a planar embedding of G has a vertex v_f for each face f of G and an edge (v_f, v_g) for each edge of G separating two (not necessarily distinct) faces f and g (Fig. 2.3(b)). Hence, the size of the dual graph is linear. Furthermore, the dual graph is always planar.



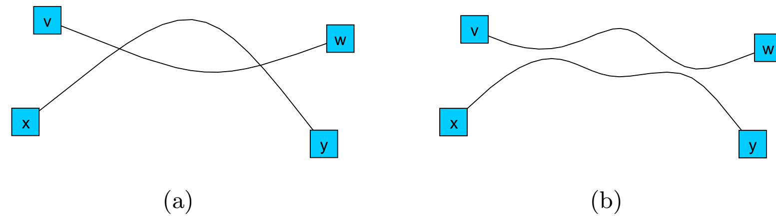
(a) planar drawing (b) dual graph (c) weak visibility representation

Figure 2.3: (a) shows a planar drawing of a graph and its faces (f_0 denotes the outer face). (b) shows the corresponding dual graph. Its vertices are denoted by circles and the edges by solid curves. A visibility representation of the planar graph is given in (c).

The *crossing number* $cr(G)$ of a graph G is the minimum number of edge crossings in any drawing of G in the plane. Determining the crossing number of non-planar graphs is NP-hard [84]. From Corollary 2.2 we know that each graph $G = (V, E)$ has at least one crossing if $|E| > 3|V| - 6$. In [120] it was shown that the lower bound on the number of crossings for any graph G with $|E| \geq 7.5|V|$ is $\frac{1}{33.75}\frac{|E|^3}{|V|^2}$. The result is based on work independently done by [2] and [110]. A simple upper bound on the number of crossings is $cr(G) = O(|E|^2)$. If every pair of edges crosses at most once, then the number of crossings is $O(|E|^2)$. Assume that there are multiple crossings between an edge pair. Then we can iteratively apply the transformation shown in Fig. 2.4 until there is at most one crossing left. Due to the lower bound given above, this bound is tight if $|E| = \Theta(|V|^2)$.

2.2 Requirements of Drawings

The requirements of a drawing of a graph depend on different factors. Often, users want to illustrate combinatorial properties of a graph, e.g., if a graph is planar, then it should be drawn planar. Furthermore, different users may



(a) (b)

Figure 2.4: Multiple crossings between a pair of edges. The drawing in (a) can be transformed into the drawing in (b) by exchanging the segments between two successive crossings.

have different preferences and visual perceptions of a drawing and thus prefer different drawing styles. In order to describe the requirements for a "nice" drawing, Di Battista et al. [46] distinguish three different concepts, namely drawing conventions, drawing aesthetics and drawing constraints. These concepts define fundamental parameters for each graph drawing methodology. In the following we give a brief overview of them.

2.2.1 Drawing Conventions

Drawing Conventions can be seen as general constraints specifying the geometric representation of edges and vertices. Thus they provide the basic rules which have to be satisfied to yield an admissible drawing. Examples of drawing conventions are:

- **Polyline Drawings** where each edge is represented as a polygonal chain
- **Planar Drawings** where no two edges cross each other
- **Orthogonal Drawings** where each edge is drawn as a sequence of alternating horizontal and vertical line segments
- **Straight-Line Drawings** where each edge is represented as a straight-line segment

In this work we focus on orthogonal drawings (ORTHOGONAL) with vertices of arbitrary size (VERTEX_SIZE). This drawing convention is widely used in the area of software and database visualization as well as for wiring diagrams (e.g., for drawing UML-class diagrams, UML-activity diagrams or Entity-Relationship diagrams).

2.2.2 Drawing Aesthetics

Aesthetics are criteria for judging general graphical properties of a drawing that should be optimized to increase readability. The drawing of a graph

with a given set of aesthetic criteria can be seen as a multi-objective optimization problem. The set of applied criteria depends on the application domain and individual user preferences.

An overview of aesthetic criteria applying to abstract graphs is given in [36, 46]. The most common are:

- Minimize the number of edge crossings (CROSSING)
- Minimize the area of the drawing (AREA)
- Minimize the deviation from a given aspect ratio (ASPECT_RATIO)
- Minimize the maximum length of an edge (EDGE_LENGTH)
- Minimize the number of bends (BEND)
- Maximize the smallest angle between two edges incident to the same vertex (ANGLE)
- Minimize the number of overlapping vertices and edges (OVERLAP)
- Maximize symmetry (SYMMETRY)

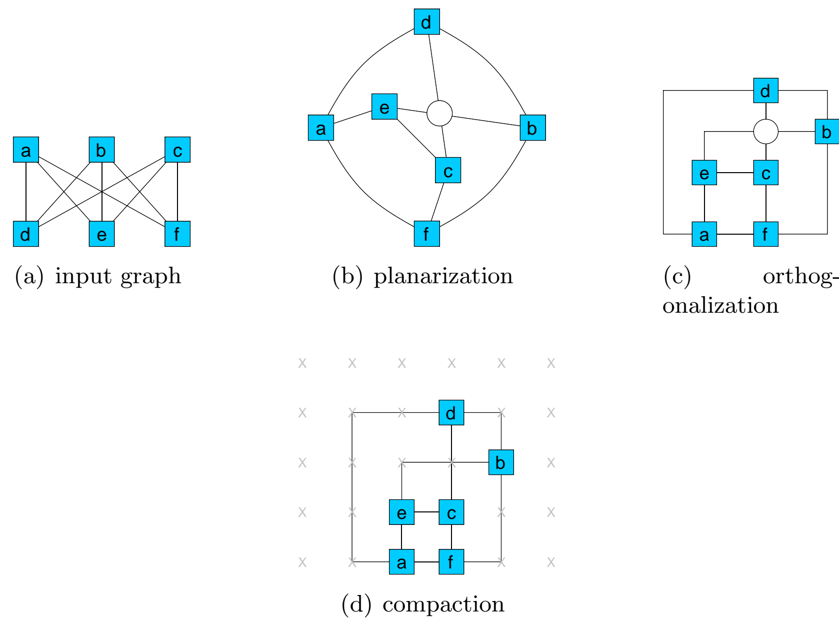
Note that it is often difficult to optimize a given set of aesthetics, since there are a lot of conflicting aesthetics like CROSSING and SYMMETRY as well as BEND and AREA.

The effects of BEND, CROSSING, ANGLE and SYMMETRY on the readability of graph drawings were empirically analyzed in [123, 124]. There, CROSSING was found to be the most important, followed by BEND and SYMMETRY. ANGLE had no significant effects.

2.2.3 Drawing Constraints

While drawing conventions and aesthetics apply to an entire drawing, drawing constraints only apply to specific parts. They usually emphasize semantic aspects of a graph (e.g., by placing related vertices close to each other) and are given as additional input to the layout algorithm. If all constraints are satisfied, a drawing is called *feasible*. Some examples of drawing constraints are the following:

- Place a given vertex to the left, right, top or bottom of another given vertex (RELATIVE_POS)
- Place a given vertex at a given position (ABSOLUTE_POS)
- Place a given vertex near the center of the drawing (CENTER)
- Place a given subset of vertices close to each other (CLUSTER)



(a) input graph (b) planarization (c) orthogonalization

(d) compaction

Figure 2.5: The three phases of the TSM approach. The white dummy vertex represents a crossing.

- Draw a given subset of edges monotonically in a prescribed direction (FLOW)

In the remainder we use the abbreviations given at the end of the above items (within the parentheses) to refer to the corresponding constraint or aesthetic criterion.

Besides the choice of the supported drawing conventions, aesthetics and constraints, runtime efficiency is another important design issue for layout algorithms. This is especially true if these algorithms are applied to application domains which require interactive usage.

In the next section, we introduce two different well-known drawing approaches for graphs, namely the TSM approach and Sugiyama's approach. Both approaches are relevant for this work.

2.3 The Topology-Shape-Metrics (TSM) Approach

The most effective concept for creating orthogonal grid drawings of undirected graph structures is the topology-shape-metrics (TSM) approach. It was introduced by Tamassia [147, 148] and later on refined and extended by several groups of authors [9, 50, 78, 105].

The TSM approach consists of three phases:

1. **Planarization**: The planarization phase determines the topology of a drawing which is described by a planar embedding. Non-planar graphs are made planar by introducing dummy vertices (also known as artificial vertices) that represent edge crossings (Fig. 2.5(b)). Common approaches try to minimize the number of edge crossings (CROSSING) during this phase.

2. **Orthogonalization**: The orthogonalization phase fixes the shapes of the edges in a drawing. It therefore determines edge bends and angles between adjacent edges. In orthogonal drawings each edge is drawn as an alternating sequence of horizontal and vertical line segments. Thus, each angle is a multiple of 90° (Fig. 2.5(c)) (ORTHOGONAL). Orthogonalization algorithms usually try to minimize the number of edge bends (BEND).

3. **Compaction**: The compaction phase determines the final coordinates of graph elements such that vertices and bends are placed on integer coordinates. Finally, the inserted dummy vertices are removed (Fig. 2.5(d)). During this phase most approaches try to minimize the area or the total edge length of the drawing (AREA, EDGE_LENGTH). Note that the placement of vertices and edge bends neither produces overlapping vertices/edges nor overlapping vertex-edge pairs (OVERLAP).

As presented above, the TSM approach incorporates several drawing aesthetics. Due to the high runtime complexity of its standard implementations, it is particularly used for medium-sized graphs (about 100 vertices and 200 edges). Note that because of the arrangement of the phases, the TSM approach is hardly applicable to constraints which fix the absolute position of vertices or the distance between them. In the following we have a closer look at the single phases.

2.3.1 Planarization

Instead of stating the cyclic ordering of edges around each vertex, the embedding of a planar graph $G = (V, E)$ can equivalently be expressed by a *planar representation* $\mathcal{P}$. For each edge $e \in E$ with endpoints v and w, the two possible orientations $\langle v, w \rangle$ and $\langle w, v \rangle$ are called *darts*. A planar representation encodes the embedding in the following way (see [65]): For each face $f \in F$ it contains a cyclic ordered list $\mathcal{P}(f)$ which contains the darts encountered when walking clockwise around f (f lies on the right-hand side when we traverse a dart of $\mathcal{P}(f)$ in the direction of its orientation). The first list of the planar representation always denotes the outer face. For the example in Fig. 2.3(a), we obtain the planar representation below. Note

that each dart is assigned to exactly one face.
$\mathcal{P}(f_0) = \{ \langle a,d\rangle; \langle d,f\rangle; \langle f,g\rangle; \langle g,b\rangle; \langle b,a\rangle \}$
$\mathcal{P}(f_1) = \{ \langle a,b\rangle; \langle b,c\rangle; \langle c,a\rangle \}$
$\mathcal{P}(f_2) = \{ \langle a,c\rangle; \langle c,f\rangle; \langle f,e\rangle; \langle e,f\rangle; \langle f,d\rangle; \langle d,a\rangle \}$
$\mathcal{P}(f_3) = \{ \langle b,g\rangle; \langle g,f\rangle; \langle f,c\rangle; \langle c,b\rangle \}$

The planarization of non-planar graphs $G = (V, E)$ is motivated by the availability of various efficient and well-analyzed drawing approaches for planar graphs. Since crossing minimization for general graphs is NP-hard, the planarization is usually done using heuristic approaches. The popular heuristic strategy described in [46] works as follows: First, a planar subgraph $G' = (V, E')$ of G is computed such that $|E'|$ is as large as possible. Recall that the calculation of a maximum planar subgraph is NP-hard. In the second step, the remaining edges $E \setminus E'$ are successively inserted one by one into G', minimizing the number of crossings caused at each insertion.

The following calculation of the planar subgraph is based on the two-phase heuristic of Goldschmidt and Takvorian [89] (GT heuristic). In an empirical comparison of different heuristics for tackling the maximum planar subgraph problem [35], only a branch-and-cut approach performed better. However, the GT heuristic has a shorter runtime and its randomized formulation, given in [127], often achieves better results than the branch-and-cut approach.

Let $G = (V, E)$ denote the undirected input graph. The GT heuristic consists of two phases: The first phase determines an ordering Π_V of the vertices of V. The vertices are placed on a fictitious vertical line according to Π_V. Let $\pi\colon V \to \mathbb{N}$ denote the function that maps each vertex $v \in V$ to its position within the sequence Π_V. Furthermore, let $e_1 = (v, w)$ and $e_2 = (x, y)$ denote two edges such that w.l.o.g. $\pi(v) < \pi(w)$ and $\pi(x) < \pi(y)$. Edge e_1 crosses edge e_2 with respect to Π_V if $\pi(v) < \pi(x) < \pi(w) < \pi(y)$ or $\pi(x) < \pi(v) < \pi(y) < \pi(w)$. The second phase partitions the edge set E into three subsets $\mathcal{L}$ (left of the line), $\mathcal{R}$ (right of the line), and $\mathcal{B}$ (the remainder) in such a way that $|\mathcal{L} \cup \mathcal{R}|$ is large (ideally maximum) and that no two edges both in $\mathcal{L}$ or both in $\mathcal{R}$ cross with respect to the sequence Π_V devised in the first phase. Therefore, we construct a conflict graph G_C that contains a vertex for each edge of G. Two vertices of G_C are connected if the corresponding edges in G cross each other with respect to Π_V. Each bipartite subgraph of G_C represents a valid assignment of the edges of E to the subsets $\mathcal{L}$, $\mathcal{R}$ and $\mathcal{B}$. However, the problem of finding a maximum bipartite subgraph of a graph is NP-complete [130]. Thus, the second phase uses the heuristic described in [4] to calculate two disjoint *independent sets* of G_C in time $O(|V||E|^2)$. An independent set of a graph $G = (V, E)$ is a subset $V' \subseteq V$ such that, for each pair $v, w \in V'$, v is not adjacent to w in G. Calculating a maximum independent set (MIS) is known to be NP-complete [83]. Note that in our application areas (software engineering, database modeling and VLSI design) the input graphs are almost always

sparse ($|E| = O(|V|)$) and the number of crossings is usually $O(|V|)$. In such a setting the runtime for this phase is $O(|V|^2)$.

If the vertex ordering calculated in the first phase corresponds to a Hamiltonian cycle in a maximum planar subgraph of G, then the number of edges of the planar subgraph obtained by the GT heuristic is at least three quarters of the number of edges of a maximum planar subgraph [89]. The following heuristic attempts to find an ordering Π_V which corresponds to a Hamiltonian cycle: The first vertex v_1 in the ordering is a vertex with minimum degree in G. Let $v_1, \ldots, v_i$ denote the first i vertices of the ordering and G_i the subgraph of G induced by the vertices of $V' = V \setminus \{v_1, \ldots, v_i\}$. The $i+1$-th vertex v_{i+1} is a vertex of V' which is adjacent to v_i in G and has minimum degree in G_i. If there is no such vertex adjacent to v_i, vertex v_{i+1} is a vertex of minimum degree in G_i. Algorithm 1 shows the corresponding pseudo code. The ordering can be calculated in $O(|V|^2)$ time.

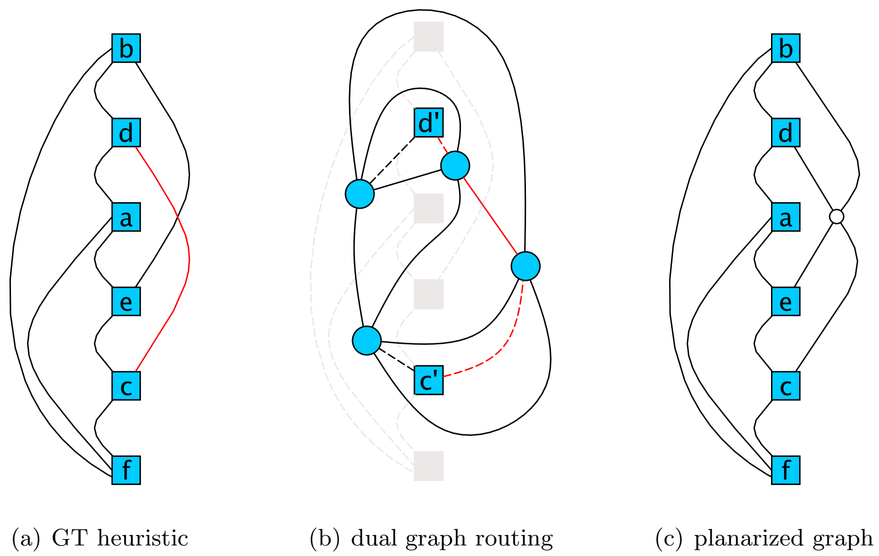
Algorithm 1: `calcGTOrdering`

Input: A graph $G = (V, E)$.
Output: The ordering function $\pi : V \to \mathbb{N}$.

```
V' ← V;
Neighbors ← ∅;
for i = 1 to |V| do
    G_i ← subgraph of G induced by V';
    Candidates ← Neighbors;
    if Candidates = ∅ then
        Candidates ← V';
    X ← {v ∈ Candidates | δ_{G_i}(v) ≤ δ_{G_i}(w) ∀ w ∈ Candidates};
    v ← randomly chosen element of X;
    π(v) ← i;
    Neighbors ← {w ∈ V' | w adjacent to v};
    V' ← V' \ v;
return π;
```

The randomized variant of the GT heuristic given in [127] calculates different vertex orderings using algorithm `calcGTOrdering`. For each ordering it applies the above independent set heuristic and then chooses the result leading to the largest planar subgraph. Since the number of considered vertex orderings is bounded by a constant factor, the overall runtime complexity does not change.

The insertion of the remaining edges is often done using shortest path computations in an extended dual graph. More precisely, if we want to insert an edge $e = (v, w)$ into a planar subgraph G' with given planar embedding, we first construct the corresponding dual graph $D_{G'}$ and insert two vertices v', w' into it (Fig. 2.6(b)). For each face f of G' whose corresponding list



(a) GT heuristic (b) dual graph routing (c) planarized graph

Figure 2.6: (a) shows the result when we apply the GT heuristic to the example of Fig. 2.5(a). The red edge (c, d) cannot be inserted without a crossing and thus is assigned to $\mathcal{B}$. In (b) we demonstrate the routing of edge (c, d) by means of the extended dual graph. Edges which were added to the dual graph are drawn using a dashed line (all edges incident to c' or d'). The red path denotes a shortest path from c' to d'. (c) presents the insertion of edge (c, d) into the planar subgraph. The white circle denotes a dummy vertex (crossing) inserted during the subdivision of edge (b, e).

$\mathcal{P}(f)$ contains a dart incident to v we insert an edge (v', u_f) into $D_{G'}$ (u_f denotes the vertex which represents f in $D_{G'}$). Analogously, we insert an edge (u_g, w') for each face g whose list $\mathcal{P}(g)$ contains a dart incident to w. A path $v' = u_1, u_2, \ldots, u_{k-1}, u_k = w'$ in $D_{G'}$ can be used to obtain a planarization of the graph $G' \cup e$ by subdividing those edges of G' which correspond to the edges (u_{i-1}, u_i), $3 \leq i \leq k-1$ of $D_{G'}$ (the first and the last edge of the path $v' \rightarrow^*_{D_{G'}} w'$ are only dummy edges). Let $x_1, \ldots, x_{k-3}$ denote the resulting dummy vertices that represent crossings. Edge $e = (v, w)$ is inserted as path $v, x_1, \ldots, x_{k-3}, w$ as shown in Fig. 2.6(c). Hence, if we insert an edge e into a graph G' with fixed embedding, the shortest path from v' to w' in $D_{G'}$ induces a planarization with minimal crossing number. Note that the number of crossings heavily depends on the chosen embedding of G'. The shortest path can be computed in linear time using a breadth-first search since the edges have unit weight [37]. Thus, the insertion of a single edge can be done in linear time. In the remainder of this work we refer to the above approach as "shortest path routing". Gutwenger et al. [93] presented a linear-time approach that determines an embedding of a planar graph G such that one additional edge can be inserted with the minimum number of crossings even among all embeddings of G.

2.3.2 Orthogonalization

The shape of an orthogonal drawing is encoded by the *orthogonal representation* $\mathcal{H}$. It extends a planar representation $\mathcal{P}$ with information about the bends along edges as well as angles between adjacent edges. More precisely, each element of a list $\mathcal{P}(f)$, $f \in F$ is extended to a tuple $(\langle v, w\rangle, s, a)$. The first entry $\langle v, w\rangle$ denotes the dart and the second entry s a bit string. The k-th bit of s represents the k-th bend that appears when going along the dart from v to w. A "1" represents a bend whose angle is 270° inside of f and a "0" a bend whose angle is 90°. If the dart has no bend, s is set to the empty string ϵ. The angle between a dart and its cyclic predecessor in list $\mathcal{P}(f)$ is specified by a. In orthogonal drawings, $a \in \{90, 180, 270, 360\}$. Thus, for the example in Fig. 2.7(a), we have

$\mathcal{H}(f_0) = \{\, (\langle a, d\rangle, \epsilon, 180); (\langle d, f\rangle, 11, 270); (\langle f, g\rangle, \epsilon, 90); (\langle g, b\rangle, \epsilon, 270);$
$(\langle b, a\rangle, 1, 180) \,\}$
$\mathcal{H}(f_1) = \{\, (\langle a, b\rangle, 0, 90); (\langle b, c\rangle, \epsilon, 90); (\langle c, a\rangle, \epsilon, 90) \,\}$
$\mathcal{H}(f_2) = \{\, (\langle a, c\rangle, \epsilon, 90); (\langle c, f\rangle, \epsilon, 180); (\langle f, e\rangle, \epsilon, 180); (\langle e, f\rangle, \epsilon, 360);$
$(\langle f, d\rangle, 00, 90); (\langle d, a\rangle, \epsilon, 90) \,\}$
$\mathcal{H}(f_3) = \{\, (\langle b, g\rangle, \epsilon, 90); (\langle g, f\rangle, \epsilon, 90); (\langle f, c\rangle, \epsilon, 90); (\langle c, b\rangle, \epsilon, 90) \,\}$

An orthogonal representation $\mathcal{H}$ of a plane graph is called *valid* if there exists a planar orthogonal point drawing inducing $\mathcal{H}$. The following theorem characterizes valid orthogonal representations:

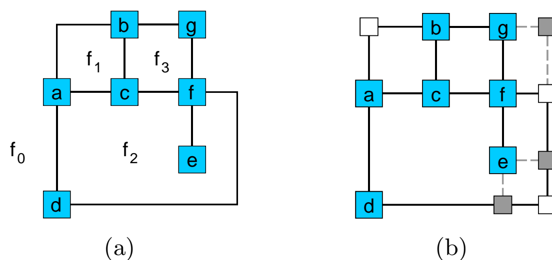


Figure 2.7: (a) shows an orthogonal drawing for the example of Fig. 2.3(a). After applying the rectangular decomposition, we obtain the result shown in (b). The smaller rectangles denote the inserted dummy vertices and the dashed lines the inserted dummy edges. White dummy vertices are those replacing bends.

Theorem 2.4 ([147]) *An orthogonal representation $\mathcal{H}$ of a plane 4-graph $G = (V, E)$ with given embedding and face set F is called* valid *if the following properties are satisfied:*

- *Let $(\langle v, w\rangle, s_1, a_1) \in \mathcal{H}(f_i)$ and $(\langle w, v\rangle, s_2, a_2) \in \mathcal{H}(f_j)$, $f_i, f_j \in F$ denote two distinct list elements whose darts represent the same edge. Then the bit string s_1 corresponds to that which we obtain when we first reverse and then negate bit string s_2, e.g., if $s_2 = 11001$ we have $s_1 = 01100$.*

- *Let $\#_0$ ($\#_1$) denote the function that states the number of 0's (1's) in a bit string. Furthermore, let $\delta(f)$ denote the number of darts defining a face f. Since each face $f \in F$ is a rectilinear polygon we have:*

$$\sum_{(\langle v,w\rangle,s,a)\in\mathcal{H}(f)} \#_0(s) - \#_1(s) + (2 - \frac{a}{90}) = \begin{cases} -4 & \text{if } f \text{ is the outer face,} \\ +4 & \text{otherwise.} \end{cases}$$

- *Let L_v denote the set of list elements with dart $\langle v, w\rangle$, $w \in V$. Then, for each $v \in V$ we have $\sum_{(\langle v,w\rangle,s,a)\in L_v} a = 360$.*

The number of bends of an orthogonal drawing is

$$\#bends = \frac{1}{2} \sum_{f\in F} \sum_{(\langle v,w\rangle,s,a)\in\mathcal{H}(f)} |s|.$$

In Section 5.1 we review the popular network flow-based algorithm of Tamassia [147] which computes bend-minimum orthogonal point drawings for plane 4-graphs with fixed embedding.

Up to now we have only considered planar 4-graphs. For planar graphs of higher degree we can no longer draw vertices as points without producing edge overlaps since there are only 4 different orthogonal directions. Thus, for those graphs, vertices are usually drawn as boxes. When two edges are incident to the same side of a vertex, the angle between them is $0°$. An orthogonal representation $\mathcal{H}$ that allows the angle values a to become 0 is called a *quasi-orthogonal representation.* A quasi-orthogonal representation is called *valid* if there exists a corresponding planar orthogonal box drawing. As shown in [77] Theorem 2.4 also holds for quasi-orthogonal representations.

Two approaches which are able to consider vertices of arbitrary degree are the GIOTTO [148] approach and the approach of Klau and Mutzel [105]. Both are based on a reduction of the input graph to a 4-graph. Their common drawback is that the resulting vertex boxes may be arbitrarily large, which is not suitable for most diagrams used in the area of software engineering or database modeling. Hence, we use the Kandinsky approach, which allows control over the vertex size.

2.3.2.1 The Kandinsky Model

The Kandinsky approach [77, 78] is based on Tamassia's network flow based algorithm and produces orthogonal drawings of general planar graphs with given embedding. In the original Kandinsky model all vertices are represented by squares of equal size. The model is also known as the ***podevsnef*** model (**p**lanar **o**rthogonal **d**rawing with **e**qual **v**ertex **s**ize and **n**on-**e**mpty **f**aces). Each vertex is placed on a coarse, rectilinear grid with uniform distance λ between the grid lines. The center of the vertices is placed on intersection points of those grid lines. Since the side length of the vertices is chosen to be $\frac{\lambda}{2}$ there are never overlapping vertices. To each coarse grid line a set of $2\kappa - 1$ fine grid lines is assigned where the edges are routed (Fig. 2.8). Recall that edges are not allowed to overlap/intersect with vertices. An intersection point between a fine grid line and the border of a vertex is called a *pin.* We always demand from a valid drawing that straight-line edges are centered at the corresponding vertex side (assigned to the κ-th fine grid line on the corresponding side). This is guaranteed if κ is chosen to be $\geq \max_{v \in V}(\delta(v))$. When we include port constraints, things are more complicated since in that case only a subset of the edges may be centered.

A valid quasi-orthogonal representation $\mathcal{H}$ of a plane graph G with face set F has a drawing in the Kandinsky model (also called a *Kandinsky drawing*) if it satisfies the following two properties:

Definition 2.5 (Bend-Or-End Property) *$\mathcal{H}$ satisfies the bend-or-end property if for every pair $\langle w, v\rangle$, $\langle v, u\rangle$ of darts following each other in a*

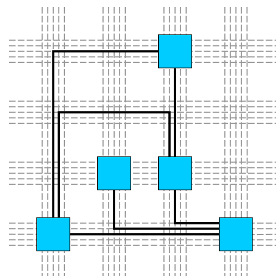
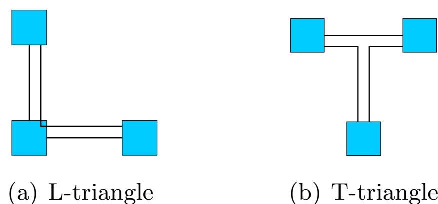


Figure 2.8: Example of a valid grid drawing in the Kandinsky model (for $\kappa = 3$). The underlying grid is illustrated by dashed lines.



(a) L-triangle (b) T-triangle

Figure 2.9: Example of an L- and a T-triangle.

cyclic ordered list $\mathcal{H}(f)$, $f \in F$ holds: either the last bend of $\langle w, v\rangle$ or the first bend of $\langle v, u\rangle$ is $270°$ inside of f.

The bend-or-end property implies that there is at most one straight-line edge per vertex side. All bends caused by the bend-or-end property are called *vertex-bends*, the remaining bends are called *face-bends*.

Definition 2.6 (Non-Empty Face Property) *Let $f \in F$ denote a triangular face ($\delta(f) = 3$) with $\mathcal{P}(f) = \{\, \langle v, w\rangle; \langle w, u\rangle; \langle u, v\rangle \,\}$. Then, f is called an* L-triangle *if $\mathcal{H}(f) = \{\, (\langle v, w\rangle, 1, 0); (\langle w, u\rangle, \epsilon, 0); (\langle u, v\rangle, \epsilon, 90) \,\}$ and a* T-triangle *if $\mathcal{H}(f) = \{\, (\langle v, w\rangle, 1, 0); (\langle w, u\rangle, 1, 0); (\langle u, v\rangle, \epsilon, 0) \,\}$ (Fig. 2.9). $\mathcal{H}$ satisfies the non-empty face property if it does not contain L- or T-triangles.*

Neither triangle can be drawn with positive area, which is often undesirable, e.g., if those faces should be labeled. As shown in [77], faces $f \in F$ with $\delta(f) > 3$ can always be drawn with positive area. The main motivation for excluding the L-triangle is that it cannot be drawn without overlap in the Kandinsky model (Fig. 2.9(a)). Furthermore, excluding L- and T-triangles leads to the following observation which is used to derive a network flow formulation for computing Kandinsky drawings.

Lemma 2.7 ([78]) *Every $0°$ angle in a Kandinsky drawing has a unique corresponding vertex-bend.*

A quasi-orthogonal representation $\mathcal{H}$ is said to be of *Kandinsky shape* if it satisfies the bend-or-end and the non-empty face properties. The bend-or-end property limits the overall number of straight-line edges as follows:

Lemma 2.8 *A Kandinsky drawing of a graph* $G = (V, E)$ *has at most* $\lceil 2(|V| - \sqrt{|V|}) \rceil$ *straight-line edges.*

Proof: Obviously, a grid graph has a maximum number of straight-line edges in a Kandinsky drawing. Each vertex at a corner of the grid has degree two, other vertices at the grid border have degree three, and the remaining inner vertices have degree four. Hence, a quadratic grid structure has a maximum number of straight-line edges. An $n \times n$ grid has $|V| = n^2$ vertices and each row/column contains $n - 1$ straight-line edges. Thus the overall number of straight-line edges is $2n(n-1) = 2(n^2-n) = 2(|V| - \sqrt{|V|})$. □

Since all other edges have at least one bend, this leads directly to:

Corollary 2.9 *A Kandinsky drawing of a graph* $G = (V, E)$ *has at least* $\lfloor |E| - 2(|V| - \sqrt{|V|}) \rfloor$ *bends.*

There are different variants derived from the original Kandinsky model (`podevsnef` model). In [44] the ***`podavsnef`*** model (**p**lanar **o**rthogonal **d**rawing with **a**rbitrary **v**ertex **s**ize and **n**on-**e**mpty **f**aces) was introduced. It allows vertices of prescribed size, i.e., the user specifies the width and height of each vertex. Since the bend-or-end property allows control over the vertex size, the changes for this model are restricted to the compaction step. The quasi-orthogonal representation corresponds to that of a `podevsnef` drawing.

Even though the complexity of finding a bend-minimum Kandinsky drawing is not known, there is an effective network flow formulation (Section 5.2) which calculates drawings with at most twice the number of bends of the optimal solution [64]. There are several extensions of the Kandinsky approach that are applicable to our purpose, e.g., the use of prescribed angles and bends [26] as well as prescribed edge shapes [64, 67].

2.3.3 Compaction

The compaction phase determines an orthogonal grid drawing for the given orthogonal representation $\mathcal{H}$, calculated in the preceding phase. It assigns lengths to the edge segments of $\mathcal{H}$, such that vertices and bends are placed on integer coordinates. Furthermore, the assignment guarantees that there are no intersections or overlaps among vertices and edges.

Several (two-dimensional) compaction algorithms for graphs are based on the so-called one-dimensional compaction approach, which has its origins in

VLSI design; see [109, 111]. In one-dimensional compaction approaches, only one dimension is changed at a time while the other dimension is fixed. Thus, a two-dimensional compaction can be performed by iteratively applying the one-dimensional compaction to the vertical and horizontal directions.

In the special case where each face of an orthogonal representation has a rectangular shape, the compaction problem can be formulated as a network flow problem. More precisely, two networks are constructed: one for the vertical and one for the horizontal edge segments. A minimum cost flow in these networks can be used to calculate a drawing with minimum width, height, area and total edge length [46]. The runtime of this approach is dominated by solving the minimum cost flow problem, which can be done in time $O(n^{\frac{7}{4}} \log n)$ as shown in [86] (n denotes the number of vertices in the network, which is linear to the number of vertices in the input graph). The compaction can be done in linear time if the minimization of the total edge length is omitted.

For general graphs the compaction problem is NP-hard [122]. In order to apply the above approaches to orthogonal representations of general graphs, we perform a so-called rectangular decomposition [147], where each face is made rectangular by introducing additional dummy vertices and edges (Fig. 2.7(b)). Bends are also replaced by dummy vertices. The rectangular decomposition can be performed efficiently by iteratively searching for certain patterns of the angles inside a face. Note that the quality of the resulting drawings depends heavily on the chosen decomposition.

Two known approaches which are not based on rectangular decomposition are the "turn-regularity" [28] and the "shape-graph" approach [106]. The second approach is based on a branch-and-cut method and calculates an optimal compaction regarding the total edge length. However, it may have exponential runtime.

The first compaction algorithm for the `podevsnef` model described in [77] was based on an adaptation of the rectangular decomposition approach. Di Battista et al. [44] presented a compaction approach for the `podavsnef` model. It starts with a `podevsnef` drawing and then expands the vertices by means of a minimum cost flow network.

Another approach which is able to handle vertices of prescribed size was presented in [64]. It consists of the following two steps: First a low quality, valid compaction is calculated and this is improved by a postprocessing algorithm in the second step. The approach is motivated by an experimental study which compares different orthogonal compaction algorithms [104]. The study shows that the results of different constructive compaction heuristics are almost the same after applying a flow-based one-dimensional compaction algorithm [111] as the postprocessing step. Thus, the first step is performed with a fast linear-time heuristic. It is based on the shape-graph compaction approach [106] and uses a variant of the rectangular decomposition.

2.4 Sugiyama's Approach

Now we turn to the abstract drawing framework of Sugiyama [146], which we use as an intermediate step during the planarization phase of our new layout approach. Sugiyama's framework is the most common approach for producing layered drawings of directed graphs. It is very popular and supported by almost all graph drawing libraries. Let $G = (V, E)$ denote a connected directed graph. The framework consists of four phases:

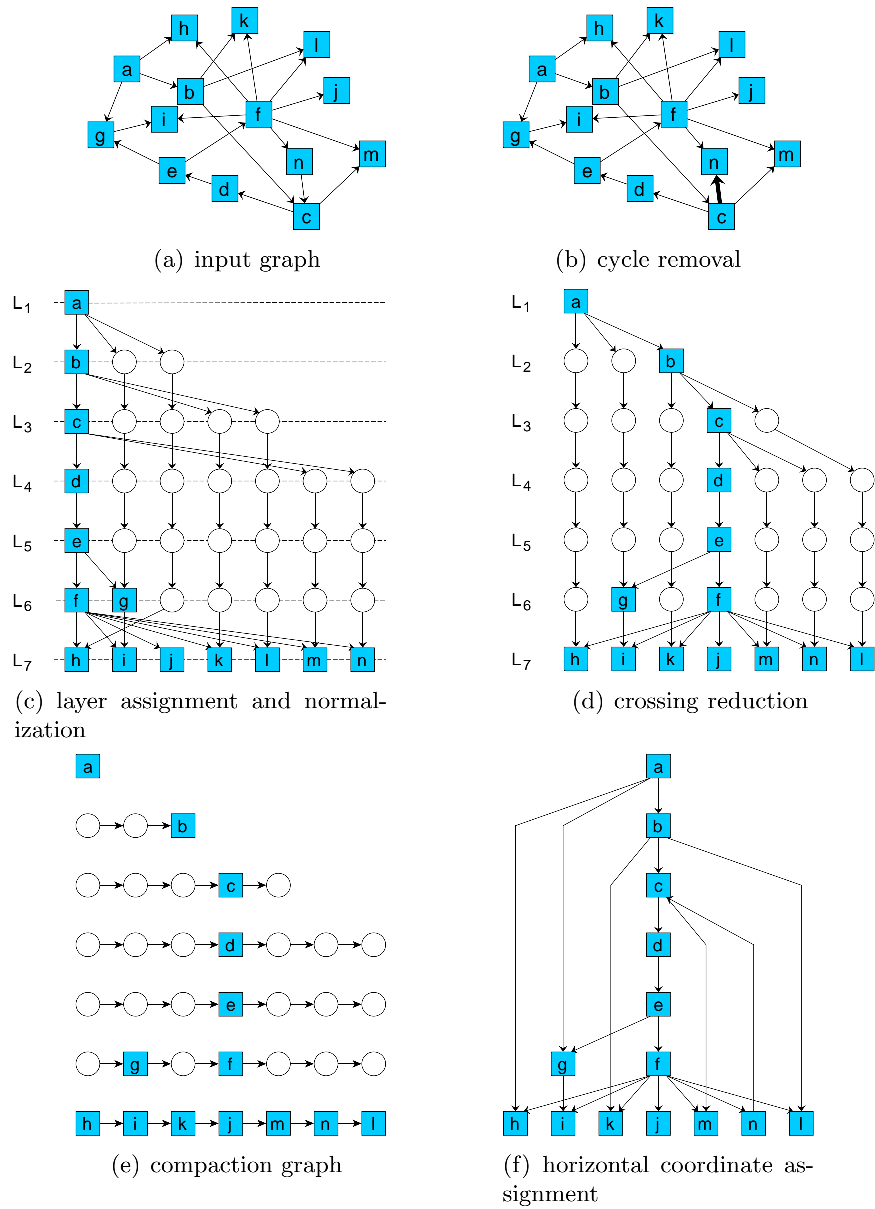
1. **Cycle Removal**: In the cycle removal phase, G is made acyclic by reversing appropriate edges (Fig. 2.10(b)).

2. **Layer Assignment**: During the layer assignment phase (also called rank assignment), the vertices of G are assigned to horizontal layers $L_1,\ldots,L_h$. L_1 represents the topmost layer. Let $\lambda\colon V \rightarrow 1,\ldots,h$ denote the function that maps each vertex to a layer number, i.e., $\lambda(v) = i$ if and only if v is in layer L_i. A valid layering has the property that for each edge $e = (v, w) \in E$ holds $\lambda(v) < \lambda(w)$. After the layer assignment, edges between vertices of non-adjacent layers (also called "long edges") are replaced by chains of dummy vertices and edges between the corresponding adjacent layers. This process is called *normalization* and the result is the normalized graph $G_N = (V_N, E_N)$ (Fig. 2.10(c)).

3. **Crossing Reduction**: In the crossing reduction phase, an ordering of the vertices within a layer is computed such that the number of edge crossings is reduced (Fig. 2.10(d)). The result is a directed acyclic compaction graph $G_a = (V_N, \{(a, b)\colon a, b \in V_N \text{ and } a, b \text{ consecutive in } L_i,\ 1 \leq i \leq h\})$. It gives the left-to-right order of the vertices in a layer and hence defines a total ordering for those vertices (Fig. 2.10(e)). Note that crossing minimization is NP-hard [84].

4. **Horizontal Coordinate Assignment**: Finally, the horizontal coordinate assignment phase uses the compaction graph to calculate an x-coordinate for each vertex. Long edges are represented by polygonal lines with the dummy vertices as intermediate points. Thus, in order to reduce the number of edge bends, all dummy vertices belonging to the same long edge should be aligned vertically. After the final layout, the reversed edges are restored to their original direction and the dummy vertices are removed (Fig. 2.10(f)).

Unfortunately, almost all problems occurring during the phases of this approach are NP-hard: Feedback Arc Set [102], Two-Layer Crossing Minimization [59], Optimal Linear Arrangement [85] etc. Nevertheless, for all these problems appropriate heuristics have been developed and nearly every practical graph drawing software uses this approach, mostly enriched by



(a) input graph

(b) cycle removal

(c) layer assignment and normalization

(d) crossing reduction

(e) compaction graph

(f) horizontal coordinate assignment

Figure 2.10: The different phases of Sugiyama's framework. Dummy vertices are drawn as white circles.

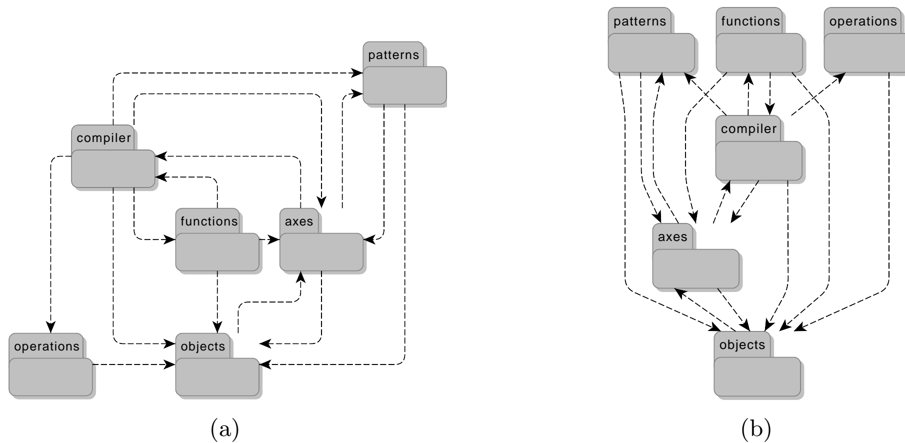


Figure 2.11: A package diagram laid out with the TSM approach (a) and with Sugiyama's approach (b). Both diagrams use rounded bends.

modifications required in practice like large vertices, same-layer-edges, clustering etc. An example of a layered graph drawing is given in Fig. 2.11(b).

The complexity of algorithms implementing Sugiyama's framework heavily depends on the number of dummy vertices inserted during the normalization. Although this number can be minimized efficiently, it may still be in the order of $\Theta(|V||E|)$ [79]. Assume that we are using an algorithm based on Sugiyama's framework which uses the fastest available algorithms for each phase. Then this algorithm has runtime $O(|V||E| \log |E|)$ and uses $O(|V||E|)$ memory.

In the following, we review Sugiyama's framework for drawing directed graphs in more detail and give the necessary definitions and algorithms.

2.4.1 Cycle Removal

In this phase the directed input graph G is made acyclic. This is necessary for the subsequent phases. Each graph can be made acyclic by reversing appropriate edges. Obviously, the number of reversed edges should be as small as possible, because in the final drawing those edges are drawn against the flow direction. Recall that the edges are only reversed internally. The problem of finding a small set of edges whose reversal makes G acyclic is closely related to the well-known feedback arc set problem for which there are several approved heuristics.

Let $G = (V, E)$ denote a directed graph. A feedback arc set $A \subset E$ is a set of edges such that the subgraph $G' = (V, E \setminus A)$ is acyclic. Finding a minimum cardinality feedback arc set is NP-hard [83]. A feedback arc set A

is called *minimal* if there is no edge of A which can be added to G' without introducing a cycle.

The greedy heuristic described in [58] determines a feedback arc set A of a directed simple graph $G = (V, E)$ such that $|E \setminus A| \geq \frac{|E|}{2} + \frac{|V|}{6}$. It runs in linear time.

A minimum weighted feedback arc set is a feedback arc set of minimum weight with respect to a nonnegative weight function $\omega\colon E \to \mathbb{R}^+$. A heuristic approach for this problem was described in [42]. For a directed graph $G = (V, E)$ it has runtime $O(|V||E|)$ and consists of two phases: First, it searches for a simple cycle $\mathcal{C}$ of G. If there is such a cycle it determines a minimum weight edge e of $\mathcal{C}$. Then, it decreases the weight of each edge of $\mathcal{C}$ by the weight of e. All edges whose weight becomes equal to 0 (true for at least e) are removed from G. If G is now acyclic, the first phase terminates, but otherwise the previous steps are repeated. Let A' denote the edges removed from G in the first phase. Obviously, A' is a feedback arc set, though not necessarily minimal. Thus, in the second phase the heuristic iteratively tries to reinsert edges of A' into G. An edge is only reinserted if this does not introduce a cycle in G. Hence, the result of the heuristic is always a minimal feedback arc set. We will exploit the following property of this heuristic:

Lemma 2.10 *Let $G' = (V', E')$ denote an acyclic subgraph of a directed graph $G = (V, E)$. If we assign weight ϕ to all edges of $E \setminus E'$ and weight $\phi \cdot |E|$ to all edges of E', the feedback arc set A returned by the heuristic never contains an edge of E'.*

Proof: Assume that there is an edge $e \in E'$ in A. Since E' is acyclic, each cycle $\mathcal{C}$ detected during the first phase of the heuristic contains at least one edge of $E \setminus E'$. Thus, the weight of e can only become equal to 0 (which is necessary to put it into A) if e is contained in at least $|E|$ cycles detected in the first phase. However, since in each step of this phase at least one edge is removed, the number of processed cycles is smaller than $|E|$. In the second phase no additional edges are inserted into A. □

Note that, in general, we cannot simply reverse all edges of a feedback arc set of G to make G acyclic; e.g., removing all the edges of a cycle makes it acyclic, but reversing those edges only reverses the cycle. However, reversing all edges of a minimal feedback arc set guarantees that G is acyclic. For heuristics which do not necessarily return a minimal feedback arc set A, we can proceed as follows: We calculate a topological ordering π of the graph $(V, E \setminus A)$ and reverse all edges $(v, w) \in A$ for which $\pi(v) > \pi(w)$. This can be done in linear time and guarantees that G is acyclic.

2.4.2 Layer Assignment and Normalization

We assume that $G = (V, E)$ is a directed acyclic graph. Let $L_1, \ldots, L_h$ be a partition of V with $L_i \subset V$, $1 \leq i \leq h$ and $\bigcup_{i=1}^{h} L_i = V$ (h denotes the number of layers). Such a partition is called a layering of G if for each edge $e = (v, w) \in E$ holds $\lambda(v) < \lambda(w)$. The *span* of an edge e is $\lambda(w) - \lambda(v)$. The number of vertices in a layer L_i is denoted with n_i. In a layered drawing, all vertices $v \in L_i$ are drawn on a horizontal line (same y-coordinate). So the layer assignment step assigns each vertex $v \in V$ a y-coordinate. We call the layering *proper* if $span(e) = 1$ for all edges $e \in E$. In most applications the layers of the vertices can be assigned arbitrarily and, in some cases, the layer assignment is even part of the input.

For edges $e = (u, v)$ with $span(e) > 1$ and for which the endpoints u and v lie on layers L_i and L_j, respectively, we replace e by a chain of dummy vertices $d_{i+1}, \ldots, d_{j-1}$ where vertex d_k, $i + 1 \leq k \leq j - 1$ is placed on layer L_k. The vertices are connected by edges (u, d_{i+1}), (d_{j-1}, v) as well as edges (d_k, d_{k+1}) for each $i+1 \leq k < j-1$. This process is called *normalization* and the result is the *normalized graph* $G_N = (V_N, E_N)$. With this construction, the next phase starts with a proper layering. Recall that in the worst case $|V_N|, |E_N| = \Theta(|V||E|)$. After the final layout of the modified graph, we replace the chains of dummy edges by polygonal chains in which the former dummy vertices become bends.

A simple layering approach is the so-called "longest path layering'". It first places all vertices $v \in V$ with zero in-degree ($\delta^-(v) = 0$) in layer L_1. Each remaining vertex v is placed in layer L_{l+1}, where l denotes the length of the longest path from v to a vertex in layer L_1. Since G is acyclic the layering can be computed in linear time using a topological ordering of the vertices. Furthermore, the layering produces a minimum number of layers.

The popular "simplex layering" approach introduced by Gansner et al. [82] calculates a layer assignment such that the total edge length, and thus the number of inserted dummy vertices, is minimized. The layering problem can be reformulated as the following integer program:

$$\text{minimize} \sum_{(v,w) \in E} (\lambda(w) - \lambda(v))$$

subject to

$$\begin{aligned} \lambda(v) &\geq 1 \qquad \forall\, v \in V \\ \lambda(w) - \lambda(v) &\geq 1 \qquad \forall\, (v, w) \in E. \end{aligned}$$

The linear program is solved by applying the network simplex method. Its time complexity has not been proven to be polynomial, but in practice it takes only a few iterations and runs quickly. Examples of other layering heuristics which work well in practice are given in [95, 129].

2.4.3 Crossing Reduction

The vertices within each layer L_i are stored in an ordered list which gives the left-to-right order of the vertices on the corresponding horizontal line. Such an ordering is called a *layer ordering.* We will often identify the layer with the corresponding list L_i. The ordering of the vertices within adjacent layers L_{i-1} and L_i determines the number of edge crossings with endpoints on both layers.

Crossing reduction is usually done using a layer-by-layer sweep where each step minimizes the number of edge crossings for a pair of adjacent layers. It is performed as follows: We start with an arbitrary vertex order of the first layer L_1 (we number the layers from top to bottom). Then iteratively, while the vertex ordering of layer L_{i-1} is kept fixed, we put the vertices of L_i in an order that minimizes crossings. This step is called *one-sided two-layer crossing minimization* and is repeated for $i = 2, \ldots, h$. After we have processed the bottommost layer, we reverse the sweep direction and go from bottom to top. These steps are repeated until no further crossings can be eliminated for a certain number of iterations. The one-sided two-layer crossing minimization problem is NP-hard [59]. Let $G' = (L_1 \cup L_2, E' \subseteq L_1 \times L_2)$ denote a two-layered (bipartite) graph where the ordering of vertices in L_1 is fixed. Many heuristics tackle this problem by first calculating a measure for each vertex of L_2 and then sorting the vertices according to their measure in ascending order.

Definition 2.11 *A* linear measure *m defines for each vertex $v \in L_2$ a non-negative value $m(v)$. If v has only one neighbor w in L_1, then $m(v) = pos(w)$, where $pos(w)$ is the position of w in layer L_1.*

In the following, we describe the two most established heuristics which are both based on a linear measure.

- **Barycenter Heuristic [146]**
  The barycenter heuristic calculates the measures as follows: the measure of a vertex $v \in L_2$ is the barycenter (average) of the positions of v's adjacent vertices in L_1. Hence, we have

  $$m(v) = \frac{1}{\delta^-_{G'}(v)} \sum_{(w,v) \in E'} pos(w)$$

  Calculating the barycenter values needs time $O(|E'|)$ and sorting the vertices according to this values needs time $O(|L_2| \log |L_2|)$.

- **Median Heuristic [59]**
  The median heuristic is related to the barycenter heuristic. Here, the measure of a vertex $v \in L_2$ is the median of the positions of v's adjacent

vertices in L_1. The median of n elements can be calculated in time $O(n)$, see, e.g., [37]. Thus, calculating the measures for the vertices has runtime $O(|E'|)$. We can sort the vertices in time $O(|L_1|+|L_2|)$ by using bucket sort [37] because the medians are integer values between 1 and $|L_1|$.

Only a few provable results on the quality of the above heuristics are known:

- For a graph G', the barycenter as well as the median heuristic give a solution without crossings, if one exists [46].
- Let x_{opt} denote the minimum number of crossings for a given one-sided two-layer crossing minimization problem. Then for the number of crossings x_{med} resulting from the median heuristic holds $x_{med} \leq 3x_{opt}$ [59].

Even if there is no such bound for the barycenter heuristic, various experiments show that it outperforms most other heuristics [100, 150]. In order to decide whether we have improved the number of crossings by a layer sweep, we have to count them. Therefore we sum the number of crossings produced by the one-sided two-layer crossing minimization steps. The problem of counting crossings of a two-layered graph G' is called the *bilayer cross counting* problem. The sweep-line approach proposed by Sander [129] solves this problem in time $O(|E'|+x)$ where x denotes the number of crossings. This bound has been improved to $O(|E'|\log(|L_1| + |L_2|))$ by Waddle and Malhotra [152]. Barth et al. [6] gave a much simpler description of the algorithm with the same running time.

It works as follows: Let L_1 and L_2 denote two adjacent layers with layer ordering $v_1, \dots, v_p$ and $w_1, \dots, w_q$, respectively. The edges between both layers are sorted lexicographically such that $(v_i, w_j) < (v_k, w_l)$ if and only if $i < k$ or $i = k \wedge j < l$. Let $e_1, \dots, e_r$ be the lexicographically sorted edge sequence, and $j_m \in \{1, \dots, q\}$ the position of the target vertex of edge e_m in L_2. An inversion in the sequence $j_1, \dots, j_r$ is a pair j_k, j_l with $k < l$ and $j_k > j_l$. Each inversion corresponds to an edge crossing between both layers. The number of inversions is counted by means of an efficient data structure, called the accumulator tree T [6]. The data structure can easily be extended to support cross counting of weighted edges.

Recall that the number of dummy vertices/edges is $O(|V||E|)$. Hence, the runtime of the crossing reduction phase is $O(|V||E|\log|E|)$ when we use the bilayer cross counting approach of Barth et al. together with the median or barycenter heuristic.

2.4.4 Horizontal Coordinate Assignment

The horizontal coordinate assignment computes the x-coordinate for each vertex with respect to the layer ordering computed during the crossing re-

duction phase. There are two objectives to consider to get nice drawings: First the drawings should be compact and second the edges should be "as vertical as possible" (with only few bends). Failure in the second objective can produce many unnecessary bends, which results in a "spaghetti effect" and reduces the readability.

Gansner et al. [82] model the horizontal coordinate assignment problem as an integer linear program:

$$\min \sum_{(v,w)\in E} \Omega(v,w)\cdot |x(v)-x(w)|$$

subject to

$$x(b)-x(a)\geq \rho(a,b) \quad \forall\, a,b\in V_N \text{ and } a,b \text{ consecutive in } L_i,\ 1\leq i\leq h$$

where $\Omega(v,w)$ denotes the priority to draw edge (v,w) vertical, $x(v)$ denotes the x-coordinate of v and $\rho(a,b)$ the minimum distance of consecutive vertices a and b. If Ω is chosen carefully, the "spaghetti effect" can be limited. The linear program can be interpreted as a layer assignment problem on a compaction graph $G_a = (V_N, \{(a,b)\colon a,b\in V_N$ and a,b consecutive in L_i, $1\leq i\leq h\})$ with length function ρ. Each valid rank assignment corresponds to a valid drawing. The above objective function can be modeled by adding vertices and edges to G_a [82].

The alternative approaches described in [27, 129] are motivated by the *linear segments model*, where each edge is drawn as a polyline with at most three segments. The first and the last segments are always proper (endpoints lie on adjacent layers) and the middle segment is drawn vertically (Fig. 2.12). Note that such drawings are only possible if the layer orderings produced during the crossing reduction phase guarantee that there are no crossing middle segments. As shown in Lemma 2.12, this is always the case if we use the traditional two-layer crossing minimization with a linear measure. In general, drawings in the linear segments model have less bends but need more area than drawings in other models.

Lemma 2.12 *Using a linear measure m, there are no middle segments crossing each other.*

Proof: A middle segment represents a chain of dummy vertices. Each dummy vertex v on a layer L_i has exactly one neighbor w in layer L_{i-1}. Hence, when we use a linear measure m, $m(v) = pos(w)$. Thus two middle segments never change their relative ordering and thus never produce a crossing with each other. □

The approach of Brandes and Köpf [27] is a longest path-based heuristic which produces nicely balanced drawings and runs in time linear to the size of the compaction graph G_a. If the crossing reduction phase uses a

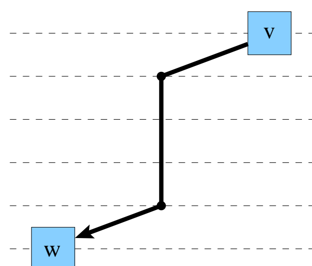


Figure 2.12: The linear segments model. Edge (v, w) is drawn as a polyline, where the first and the last segments are proper and the middle segment is drawn vertically. Dashed lines denote the layers.

linear measure, the resulting drawings always conform to the linear segments model. The heuristic first tries to align a vertex with either its median upper or its median lower neighbor. Aligned vertices share the same vertex in the compaction graph and thus get the same x-coordinate. There are three kinds of alignment conflicts: type 0 conflicts arise between two non-inner segments (a non-inner segment has at least one non-dummy vertex as an endpoint), type 1 conflicts arise between a non-inner segment and an inner segment and type 2 conflicts between two inner segments. Note that type 2 conflicts do not appear if the crossing reduction uses a linear measure. Type 1 conflicts are always resolved in favor of the inner segment.

Regardless of whether the vertices are aligned with their median upper or median lower neighbor, alignment conflicts can be resolved either in a leftmost or a rightmost fashion. Thus, there are four possible combinations for aligning the vertices. For each combination, the horizontal coordinates are calculated by a longest path algorithm. Finally, the four resulting coordinate assignments are combined to get a balanced drawing.

Chapter 3

Orthogonal Graph Drawing with Constraints

Although there are a lot of publications on orthogonal graph drawing (see [46] for an overview), very few of these publications consider drawing constraints that arise in diagram types used for practical applications like software engineering, database modeling and VLSI (Very Large Scale Integration) design. Handling such constraints is very important for producing adequate visualizations for these applications.

In the first section, we present five different drawing constraints which often appear in graph-based diagrams. The new approach presented in this work is able to simultaneously include all these constraints. For the design of our approach, we assume that the underlying graphs are sparse (density ≤ 2) and of medium size (≤ 100 vertices). This assumption applies to most diagram types which are used to visualize the applications mentioned above. After the presentation of the single constraints, we take a look at issues arising when combining them. We also review related work dealing with combinations of multiple constraints. In the last section, we present the interface of our new approach for the automatic layout of graphs with constraints and give an overview of its single steps. In the remainder of this work, we assume that vertices are represented as boxes.

3.1 Drawing Constraints

In the following we will introduce different important drawing constraints. Besides theoretical results, like the complexity of planarity testing as well as crossing and bend minimization, we will also state relevant practical approaches done so far. Note that we focus on TSM-based approaches here, i.e., approaches which realize one or more phases of the TSM approach.

3.1.1 Bimodal Drawings (BIMODAL)

Let $G = (V, E_D)$ denote a directed graph. For a given drawing/embedding of G, a vertex $v \in V$ is called *bimodal* if the incoming edges (and thus the

outgoing edges) around v are consecutive. A drawing of G is called a *bimodal drawing* if each vertex $v \in V$ is bimodal. In practice, incoming and outgoing edges of a vertex are often placed on opposite sides. Note that for any graph G there is always a bimodal drawing. G is called *bimodally planar* if it has both a bimodal and a planar drawing at the same time. Note that there are graphs which have a bimodal and a planar drawing but are not bimodally planar (Fig. 3.1). A *bimodal embedding* of a graph is given by a clockwise cyclic ordering of the edges around each vertex in which the incoming and outgoing edges form separate, non-intersecting intervals. Such an ordering of the edges around a vertex is called *bimodal ordering*.

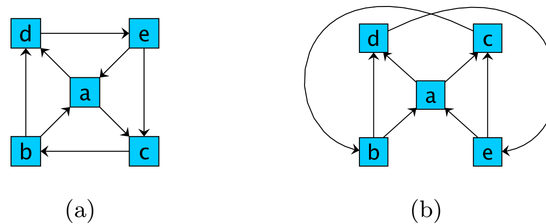


Figure 3.1: Example of a graph which has a planar (a) and a bimodal drawing (b) but not a planar bimodal drawing.

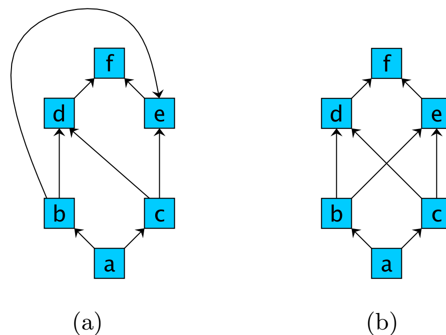
A *mixed graph* is a tuple $G = (V, E_D \cup E_U)$, where V denotes the set of vertices, E_D the set of directed edges and E_U the set of undirected edges. A drawing of a mixed graph is called bimodal if the sub-drawing induced by (V, E_D) is bimodal. We also allow specifying only a subset $V' \subseteq V$ of vertices that have to be bimodal. In that case a drawing is called bimodal if the sub-drawing induced by (V', E_D) is bimodal.

Bimodal drawings are helpful for graphs where the edge direction should be emphasized, e.g., for UML class and entity-relationship diagrams. They ensure a better recognition of the edge direction by separating incoming and outgoing edges of vertices.

Testing for whether a directed graph $G = (V, E_D)$ is bimodally planar can be done in time $O(|V|)$ [10]. Obviously, each planar graph with maximum vertex degree 3 is bimodally planar, because any order of the edges around the vertices is bimodal. As shown in [72], calculating a maximum planar subgraph for cubic graphs (all vertices have degree 3) is NP-hard. Hence, calculating a maximum bimodally planar subgraph is NP-hard, too. Furthermore, in [98] it is shown that crossing minimization is NP-hard for cubic graphs and thus it is also NP-hard when we demand a bimodal ordering of edges around vertices. Buchheim et al. [32] show how to adapt the traditional planarization method in order to produce bimodal embeddings of directed graphs.

3.1.2 (Mixed) Upward Drawings (FLOW)

An *upward drawing* of a directed graph $G = (V, E_D)$ is a drawing of G such that all edges are represented by monotonically increasing curves in the vertical direction. Furthermore, we add the restriction that incoming edges enter a vertex at the bottom and that outgoing edges leave a vertex at the top. An upward drawing exists if and only if G is acyclic. Note that an upward drawing always induces a bimodal drawing. G is called *upward planar* if it has an upward and a planar drawing at the same time. Note that there are graphs which have an upward and a planar drawing but are not upward planar (Fig. 3.2). An *upward embedding* of a graph is given by a clockwise cyclic, bimodal ordering of the adjacent edges around each vertex.



(a) (b)

Figure 3.2: Example of a graph which has a planar (a) and an upward (b) drawing but not a planar upward drawing.

A *mixed-upward drawing* of a mixed graph $G = (V, E_D \cup E_U)$ is a drawing such that the edges of E_D are represented by monotonically increasing curves in the vertical direction. G is called *mixed-upward planar* if it has a mixed-upward and a planar drawing at the same time.

Mixed upward drawings arise in applications where the edges of a graph can be partitioned into a set which denotes a hierarchical structure and a set which does not have such a structure. Well-known examples are data flow diagrams as well as UML class diagrams where we have generalization and association edges.

In [87], Garg and Tamassia showed that testing directed graphs for upward planarity is NP-hard in general. It follows directly that crossing minimization as well as the calculation of a maximum upward planar subgraph is also NP-hard. The same results apply to mixed-upward graphs which are a generalization of upward graphs. Note that for a fixed planar embedding of a directed graph, testing upward planarity can be done in time $O(|V|^2)$ [11].

An approach for mixed-upward planarization was given by Eiglsperger et al. [69]. The approach is based on a modification of the GT heuristic (Section 2.3.1) which allows preserving the given direction of directed

edges and thus produces a mixed-upward planar subgraph. This subgraph is augmented to an upward planar st-graph by inserting appropriate dummy edges. A special routing graph, which guarantees monotone edge routes, is used to insert the remaining directed edges. Finally, the dummy edges are removed and the undirected edges not yet in the subgraph are inserted using shortest path routing. The overall runtime of this approach is $O(|V||E|^2 + (|V| + c)^2|E|)$.

An integer linear program (ILP) formulation for calculating bend-minimum orthogonal upward drawings of graphs with given upward embedding can be found in [66]. In [70], we presented the "UML-Kandinsky" approach for drawing UML class diagrams. More precisely, we showed how to calculate a mixed-upward drawing for a given mixed-upward planar embedding of a connected mixed input graph $G = (V, E_D, E_U)$. The algorithm works as follows: First, we only consider directed (upward) edges, i.e., the subgraph induced by the edges of E_D. For each vertex v, we assign a "head-shape" to the incoming edges and a "tail-shape" to the outgoing edges. The assigned shape depends on the given embedding and guarantees monotonically increasing edge routes complying with the Kandinsky model. We obtain the final shape for directed edges by concatenating the head- and tail-shapes. In the second step, we calculate the shape of undirected edges by solving a minimum cost flow problem. In order to produce a mixed-upward drawing with this approach, we have to guarantee that the subgraph induced by the directed edges is connected. Hence, we calculate a spanning tree of G and, if necessary, temporarily add some edges of E_U to E_D. In Section 5.3 we will review this approach in more detail. A similar approach (called the "GoVisual" approach) was independently described in [90].

3.1.3 Cluster Drawings (CLUSTER)

A *cluster* of a graph is a non-empty subset of vertices. A *clustered graph* $G_C = (G, T)$ consists of a graph G (called the *underlying graph*) and a directed rooted tree T (called the *inclusion tree*) that describes the hierarchical clustering structure. The leaves of T are exactly the vertices of G. Edges of T are directed from the root to the leaves. Each internal (non-leaf) vertex c of T has at least two children and represents the cluster of G whose vertices are the leaves of the subtree rooted at c. The induced subgraph of G on those vertices is denoted by $G(c)$. The root of T is a special vertex which represents the whole drawing area and thus includes each element. An example of a clustered graph is given in Fig. 3.3.

In a *cluster drawing* of a clustered graph G_C, the vertices of G are drawn as points and the edges as simple curves. Each internal vertex c of T is drawn as a simple closed region enclosed by a simple closed curve. In this work each internal vertex (cluster) is drawn as a rectangle. The region of c contains a vertex v of G (leaf of T) if and only if there is a path $c \rightarrow_T^* v$. Analogously,

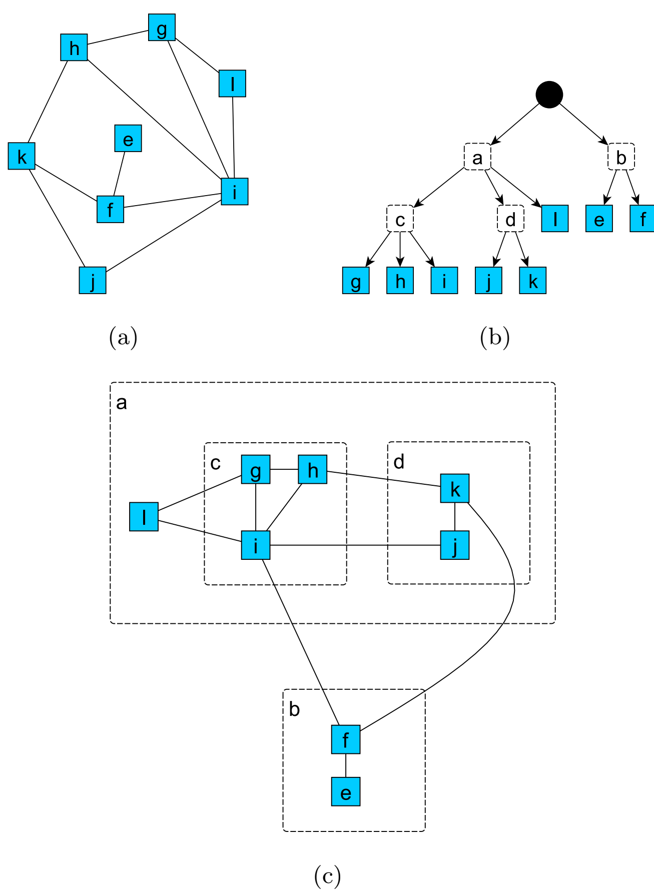


Figure 3.3: Example of a clustered graph, where vertices representing clusters are drawn as rounded rectangles with a dashed border. (a) shows the underlying graph, (b) the corresponding inclusion tree and (c) the resulting cluster drawing.

it contains the region of another internal vertex c' if and only if there is a path $c \to_T^* c'$. Furthermore, in a cluster drawing, each edge of G crosses the boundary of a region at most once.

A clustered graph is called *c-planar* if it has a planar and a cluster drawing at the same time. Note that there are clustered graphs which have a cluster drawing as well as a planar drawing but are not c-planar (Fig. 3.4). A clustered graph is called *c-connected* if for each internal vertex c of T the induced subgraph $G(c)$ is connected. Let $G_C = (G, T)$ and $G'_C = (G', T')$ denote two clustered graphs. G'_C is called a *sub-clustered graph* of G_C if T' is a subtree of T and for each vertex c of T' (and thus of T) $G'(c)$ is a subgraph of $G(c)$.

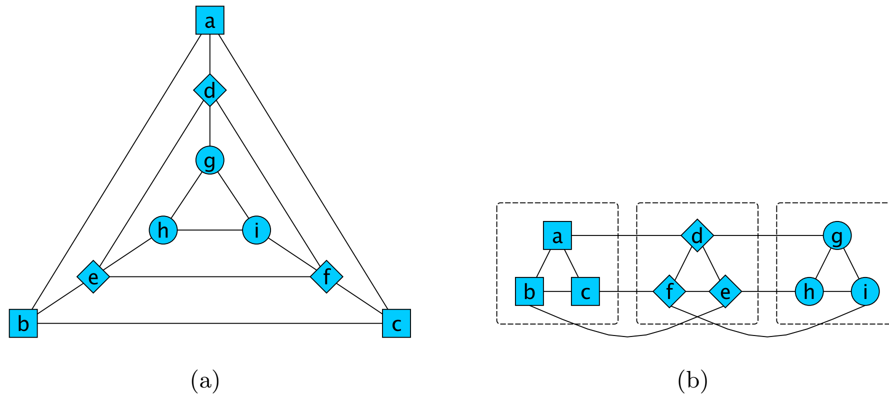


Figure 3.4: A graph which has a planar (a) and a cluster (b) drawing but not a planar cluster drawing (different vertex shapes denote different clusters).

Cluster drawings of graphs are required for several application domains where vertices are grouped together into clusters. Examples are UML class diagrams where clusters denote the package structure, large computer networks where clusters represent local area networks and wiring diagrams where complex building blocks are mapped to clusters.

There has been a lot of research on clustered graphs (see [29] for an overview). However, the complexity of the c-planarity testing problem for general clustered graphs is still unknown. For c-connected clustered graphs, the following characterization of c-planarity was given by Feng et al. [73]:

Theorem 3.1 ([73]) *A c-connected clustered graph $G_C = (G, T)$ is c-planar if and only if G is planar and there exists a planar drawing of G such that for each vertex c of T, all the vertices and edges of $G \setminus G(c)$ are in the outer face of the drawing of $G(c)$.*

In the same work it is also shown that for a c-connected clustered graph with n vertices, testing planarity and computing the corresponding c-planar embedding can be done in time $O(n^2)$. This bound was improved to $O(n)$

by Dahlhaus [40]. Another, more precise description of an algorithm with the same runtime was given by Cortese et al. [39]. Some other important results for c-planarity testing of special subclasses of clustered graphs are given in [38, 47, 91].

The problems of calculating a maximum c-planar subgraph and finding a crossing minimum c-planarization for a clustered graph are both NP-hard, since for the special case of having only one cluster which contains all vertices both problems correspond to the related planarization problems for common graphs.

A planarization algorithm for clustered graphs is given in [43]. For a clustered graph $G_C = (G, T)$, $G = (V, E)$ the algorithm proceeds as follows: First a c-connected c-planar sub-clustered graph $G'_C = (G', T)$ is computed such that $G' = (V, E')$ is a spanning tree of G. Then G'_C is extended to a c-connected maximal c-planar sub-clustered graph by successively testing whether the insertion of an edge $e \in E \setminus E'$ into G' would still leave G'_C c-planar. If so, e is inserted. Recall that for c-connected clustered graphs, testing c-planarity can be done in linear time. Finally, the remaining edges are inserted using shortest path routing in a modified dual graph. Let n, m, c and x denote the number of vertices of G, edges of G, non-leaf vertices of T and crossing vertices, respectively. The overall runtime of this approach is $O(mx + m^2c + nmc)$

A linear time orthogonalization algorithm for clustered graphs was given in [55]. Its input is an n vertex c-connected clustered 4-graph with a c-planar embedding. The output is a c-planar orthogonal grid drawing with rectangular cluster regions, $O(n^2)$ area and at most 3 bends per edge. The algorithm first constructs a special visibility representation and uses local operations to transform each horizontal segment into a point. Note that the resulting edge routes have at most 4 bends. Finally, a rotation procedure eliminates all 4 bend edges.

An extension of Tamassia's orthogonalization approach which is able to include clustered graphs was independently described in [24] and [112]. The rectangular shape of clusters is realized by putting additional constraints on the flow of Tamassia's network formulation. Note that the applied modifications still lead to a bend-minimum solution for 4-graphs. The same modifications can be applied to the Kandinsky network [43]. More details are given in Section 5.3.

For our approach we will use *compound graphs* [144], which are an extension of clustered graphs. While in a clustered graph the vertices of the underlying graph are exactly the leaves of the inclusion tree, the underlying graph and inclusion tree of a compound graph are defined on the same set of vertices. This allows modeling edges connecting two clusters or a cluster and a (common) vertex. In the following we will refer to the vertices representing clusters as *compound vertices* and to the remaining vertices as *base vertices*.

3.1.4 Partitioned Drawings (PARTITION)

In the last section, we presented clustered/compound graphs. In a cluster drawing, all vertices of a cluster are placed inside the same rectangular region. Regions can be nested, and their positions are not given as input. In this section, we consider the problem of placing each vertex inside a predetermined partition cell of a rectangular partitioned drawing area. Partition cells have fixed relative positions and do not overlap. Their size is not given.

Even if such partitions have several applications in practice, e.g., for UML activity diagrams, to our knowledge, our work described in [135] was the first one dealing with this kind of drawing constraint. The following definitions and results are based on the results described there.

Let A_R denote the (rectangular) drawing area. A *(rectangular) partition* P_R of A_R is a partition of A_R into a set $R = r_1, \ldots, r_k$ of non-overlapping rectangles (called *partition cells*); see Fig. 3.5(a) for an example. The corresponding *partition grid graph* P_G is constructed from P_R by placing a vertex on each point where a horizontal segment touches or intersects a vertical segment. The underlying structure of P_G is a rectilinear grid graph which enables us to assign grid coordinates to the vertices as shown in Fig. 3.5(b). For each partition cell $r \in R$, let r^t, r^b, r^l and r^r represent the grid coordinate of the top, bottom, left and right border, respectively (e.g., for partition cell r_4 in Fig. 3.5, $r_4^t = 2$, $r_4^b = 3$, $r_4^l = 3$ and $r_4^r = 4$). Recall that these coordinates do not indicate distances. Only the topology and shape of P_G has to be preserved – the size of the partition cells is not fixed. A partition P_R is called a *regular partition* if the associated partition grid graph P_G corresponds to a rectilinear grid. Otherwise, P_R is called an *irregular partition.*

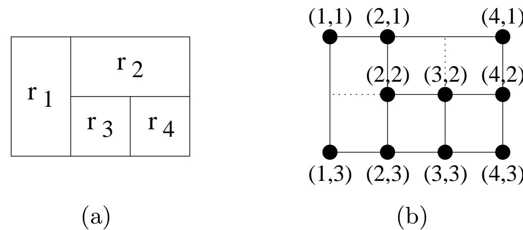


Figure 3.5: An irregular partition P_R (a) and the corresponding partition grid graph P_G (b).

Definition 3.2 *Let $G = (V, E)$ denote a graph, P_R a partition and $p\colon V \rightarrow R$ a function that maps each vertex to a partition cell. A drawing of G is called a* partitioned drawing *if each vertex $v \in V$ is drawn inside $p(v)$. G is called* p-planar *if it has a partitioned and a planar drawing at the same time.*

UML activity diagrams often use regular partitions to divide a diagram into logical areas, e.g., organizational units in a business model. Typically, the drawing area is subdivided into vertical or horizontal swimlanes (stripes). Such a partitioning is especially useful to emphasize a logical flow or time flow in a drawing. Activity diagrams also offer more complex, grid-like partitions which are a combination of horizontal and vertical swimlanes (Fig. 3.6(a)).

Irregular partitions are often used to indicate geometric information or positions, e.g., for wiring schematics as shown in Fig. 3.6(b). Statechart diagrams employ a similar structure but determine the partitions during the layout process. Here, the partition structure is already given as input.

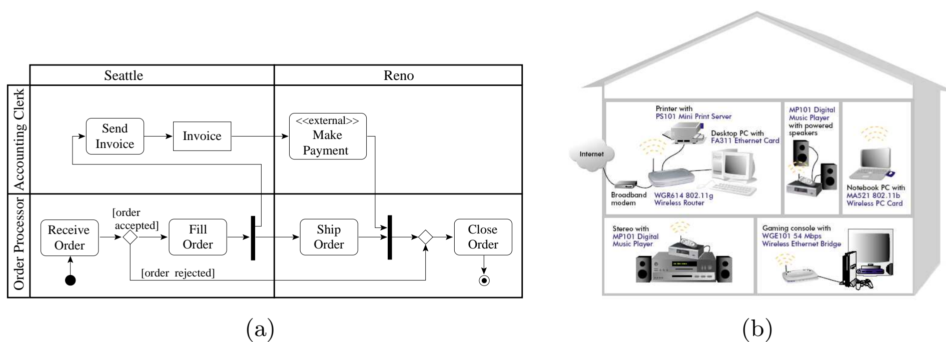


Figure 3.6: Partitioned drawings: (a) shows a UML activity diagram taken from the UML 2.2 Superstructure specification [118] and (b) a wiring schematic taken from www.netgear.de.

The following theorem gives a characterization of p-planar graphs.

Theorem 3.3 *A graph G is p-planar if and only if it is planar. A p-planar embedding of a p-planar graph can be constructed in time $O(|V|^2)$.*

Proof: A p-planar graph is planar by definition. Let us assume that each vertex $v \in V$ is assigned to an arbitrarily distinct location inside of $p(v)$. Pach and Wenger [121] showed that every planar graph admits a planar embedding which maps each vertex to an arbitrarily prescribed distinct location and each edge to a polygonal curve with $O(|V|)$ bends. Such an embedding can be found in $O(|V|^2)$ time and implies that each planar graph is p-planar. □

An example of such an embedding is given in Fig. 3.7(b). Theorem 3.3 has several consequences: since testing a graph for planarity can be done in time $O(|V|)$ [99], the same is true for testing p-planarity. Furthermore, the problems of calculating a maximum p-planar subgraph and finding a crossing minimum p-planarization are both NP-hard, because the related planarization problems are also NP-hard.

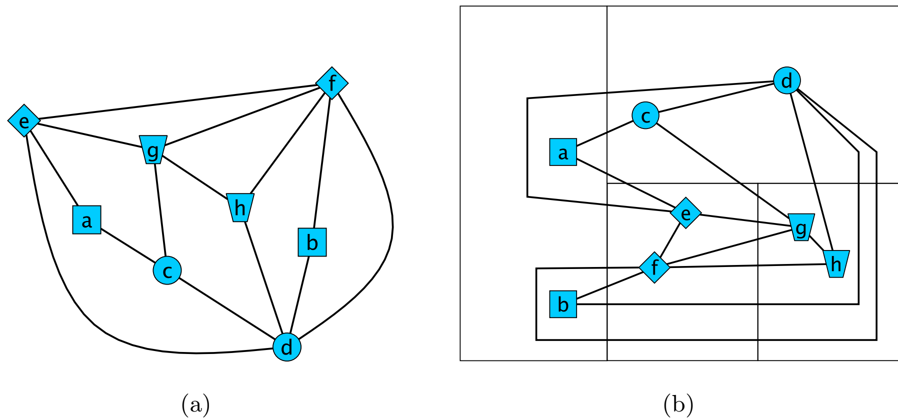


Figure 3.7: (a) shows a planar graph and (b) the corresponding p-planar embedding using the partition of Fig. 3.5(a) (different vertex shapes denote different partitions).

3.1.5 Port/Side Preserving Drawings (PORT/SIDE)

In this section we consider two related types of edge constraints, namely side and port constraints.

The location where an edge joins an incident vertex is called a *port*. Using rectangular vertices, the two ports of an edge can lie at the **t**op, **r**ight side, **b**ottom or **l**eft side of a vertex. Let $dir = \{t, r, b, l\}$ denote the set of different sides. If we restrict an edge e to leave/enter its incident vertex v at a prescribed side $s \in dir$, we call this a *side constraint* and denote it with sc_e^v. The set containing all side constraints of a graph G is denoted by SC_G. The function $side\colon SC_G \rightarrow dir$ maps each side constraint to the prescribed side.

Definition 3.4 *A drawing of a graph $G = (V, E)$ is called a* side constraint preserving drawing *if all side constraints are fulfilled, i.e., for each side constraint $sc_e^v \in SC_G$ edge e leaves/enters vertex v at side $side(sc_e^v)$. G is called* side constraint preserving planar *if it has a side constraint preserving and a planar drawing at the same time.*

There are graphs which have a side constraint preserving and a planar drawing but are not side constraint preserving planar. When we assign all incoming edges of a vertex to the same side and all outgoing edges to another side, we can reuse Fig. 3.1 to give a corresponding example.

Recall that in the Kandinsky (`podevsnef`) model there are $2\kappa - 1$ fine grid lines on each vertex side which are used for routing the edges (see Section 2.3.2.1). Hence, each port of an edge has to lie on a pin (intersection point between a fine grid line and the vertex border). Fig. 3.8 shows a

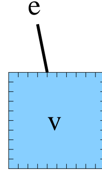


Figure 3.8: A vertex v whose pins are denoted by small dashes running orthogonal to v's border. Edge e has a port constraint pc_e^v with $side(pc_e^v) = t$ and $location(pc_e^v) = 4$.

vertex with 9 pins on each side ($\kappa = 5$). Let P_v denote the set of pins of a vertex v. If we restrict an edge e to leave/enter its incident vertex v at a prescribed pin $p \in P_v$, we call this a *port constraint* and denote it with pc_e^v. PC_G denotes the set containing all port constraints of graph G. The injective function $pin\colon PC_G \rightarrow P$ where $P = \bigcup_{v \in V} P_v$ maps each port constraint to the corresponding pin. Port constraints can be seen as a specialization of side constraints; they do not only specify the side but also the exact pin where an edge leaves/enters a vertex. We extend the function $side$ to $side\colon \{PC_G \cup SC_G\} \rightarrow dir$, accordingly. Furthermore, the function $location\colon PC_G \rightarrow \{1, \dots, 2\kappa - 1\}$ maps a port constraint to the number of the corresponding pin (the pins are numbered separately for each side and the numbers increase clockwise).

Definition 3.5 *A drawing of a graph $G = (V, E)$ is called a* port constraint preserving drawing *if it is a side constraint preserving drawing and all port constraints are fulfilled, i.e., for each port constraint $pc_e^v \in PC_G$, edge e leaves/enters vertex v at pin $pin(pc_e^v)$. G is called* port constraint preserving planar *if it has a port constraint preserving and a planar drawing at the same time.*

In this work we consider the following three scenarios separately:

- `sc` **scenario:** all edges have side constraints on both endpoints
- `pc` **scenario:** all edges have port constraints on both endpoints
- `mc` **scenario:** edges may have mixed constraints, i.e., port, side and no constraints

Port and side constraints arise in graph-based diagrams where the sides of the represented objects or the positions where the relation edges enter/leave an object have different semantics.

In electrical circuit schematics the building blocks are connected by a set of wires which join the blocks at prescribed ports. Furthermore, there are several electric components with fixed wirings like multiplexer and integrated circuits. These components can be modeled by means of port constraints.

In UML class diagrams, inheritance edges are often expected to start at the top and end at the bottom of the respective class vertices. Association edges are preferably placed on the left or right side of a vertex.

In [45], an orthogonal layout algorithm for drawing database schemas is presented where database tables are displayed as boxes containing a vertically ordered sequence of attributes. Edges denoting links between table attributes are only allowed to leave/enter tables at the left or right side and must be attached to the corresponding attributes (as in the example shown in Fig. 1.1(b) on page 2). The planarization approach presented there is tailored to this specific scenario and cannot be adapted to our port/side constraint model.

Integer linear program (ILP) formulations for calculating bend-minimum orthogonal drawings of graphs with fixed embeddings and various constraints including port and side constraints are given in [66].

Below we investigate planarization issues arising for port and side constraints. Let $id\colon dir \rightarrow \{0, 1, 2, 3\}$ denote a function that maps vertex sides to integers as follows: $id(t) = 0$, $id(r) = 1$, $id(b) = 2$ and $id(l) = 3$. Let E_v^{sc} (E_v^{pc}) denote the set of edges incident to vertex $v \in V$ and with side (port) constraints on v.

Definition 3.6 *An embedding is called* side constraint preserving *if the clockwise order of the edges $e \in \{E_v^{sc} \cup E_v^{pc}\}$ around each vertex $v \in V$ induces a monotonic function regarding $id(side(sc_e^v))$ and $id(side(pc_e^v))$, respectively.*

A side constraint preserving embedding ensures that we can calculate a side constraint preserving drawing. For two port constraints $pc_v^{e_1}, pc_v^{e_2}$ with $e_1, e_2 \in E_v^{pc}$, the function $\rho\colon PC_G \times PC_G \rightarrow \mathbb{N}$ gives the number of pins lying between pin $pin(pc_v^{e_1})$ and pin $pin(pc_v^{e_2})$ in cyclic clockwise order (not including $pin(pc_v^{e_1})$ and $pin(pc_v^{e_2})$). Furthermore, let $E_v^{e_1,e_2}$ denote the set of edges lying between edge e_1 and e_2 (not including e_1 and e_2) regarding the cyclic clockwise edge order around v.

Definition 3.7 *An embedding is called* port constraint preserving *if the following conditions are satisfied:*

1. *The embedding is side constraint preserving.*
2. *For each side $s \in dir$ of a vertex v, the clockwise order of all edges in $\{e \mid e \in E_v^{pc} \wedge side(pc_v^e) = s\}$ induces a strictly monotonic function with respect to $location(pc_e^v)$.*
3. *All edges lying between two edges with port constraints can be assigned to a pin. More precisely, for any pair of edges e_1, e_2 with port constraints $pc_v^{e_1}$, $pc_v^{e_2}$ on a vertex v is $\rho(pc_v^{e_1}, pc_v^{e_2}) \geq |E_v^{e_1,e_2}|$.*

4. *An edge with side constraint can always be assigned to a pin at the corresponding side. Among all edges with port constraints on side $s \in dir$ of a vertex v, let e denote the edge assigned to the port with lowest pin number and f the edge assigned to the port with highest pin number. Let us further assume that there is at least one edge g with port or side constraint on another side. Then, for each edge h with side constraint on side s must hold: $location(pc_v^f) + |E_v^{f,h}| < 2\kappa - 1$ if h appears in the (cyclic clockwise) edge order between f and g, and $location(pc_v^e) - |E_v^{h,e}| > 1$, otherwise. If there is no such edge g, at least one of the above conditions must be satisfied.*

Note that items 3 and 4 are only relevant for the `mc` scenario. Crossing minimization in the `mc` scenario is NP-hard since it is a generalization of the common crossing minimization problem. The same applies to the `sc` scenario, since it can be reduced to the common crossing minimization problem by assigning all edges to the same vertex side. The time complexity of the planarization problem in the `pc` scenario is unknown (we do not allow edges to share a pin).

A linear-time approach for testing planarity of graphs with embedding constraints was recently described in Gutwenger et al. [92]. The authors introduce three different types of embedding constraints – grouping, mirror and oriented constraints – which can be hierarchically nested. More precisely, 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

The oriented constraints can be used to obtain a linear-time planarity test for the `pc` scenario. For the `sc` scenario we need to nest oriented and grouping constraints. Note that the above approach does not allow the modeling of vertices which have both incident edges with and without constraints. Thus, we can not apply it to the `mc` scenario.

3.2 Combining Drawing Constraints

In this section we look at issues arising when we combine the five aforementioned constraints. We state some theoretical results which can be directly

derived from the results of the single constraints, investigate the compatibility of constraints and also review related work.

Definition 3.8 *A mixed compound graph G_C has a* port constraint preserving bimodal partitioned mixed-upward cluster drawing *if it has a bimodal, a partitioned, a mixed-upward, a cluster and a port constraint preserving drawing at the same time. Furthermore, G_C is called* port constraint preserving bimodally mixed-upward p,c-planar*, if it has a planar and a port constraint preserving bimodal partitioned mixed-upward cluster drawing at the same time.*

Let G_C denote a mixed compound graph. A port constraint preserving bimodal partitioned mixed-upward cluster drawing is a generalization of each drawing with only a subset of these constraints. Hence, the following results can already be derived from upward drawings:

Corollary 3.9 *Testing whether G_C is port constraint preserving bimodally mixed-upward p,c-planar is NP-hard.*

Corollary 3.10 *The calculation of a port constraint preserving bimodal partitioned mixed-upward cluster drawing for G_C with the minimum number of crossings is NP-hard.*

Corollary 3.11 *The calculation of a maximum port constraint preserving bimodally mixed-upward p,c-planar subgraph of G_C is NP-hard.*

3.2.1 Drawing Compatibilities

More important than minimizing crossings is the preservation of the drawing convention given by the different constraints. To satisfy constraint FLOW the directed input graph has to be acyclic. The other constraints do not have special requirements for the input graph. However, the combination of multiple constraints can produce drawing conflicts.

While constraints BIMODAL and PORT/SIDE affect the cyclic order of edges around vertices, constraints CLUSTER and PARTITION affect the geometric position of vertices. Since both effects do not interact with each other, combining constraints CLUSTER and PARTITION with constraints BIMODAL and PORT/SIDE is always possible without generating drawing conflicts. Constraint FLOW can be seen as a hybrid because, on the one hand, it has the same effect on the cyclic order as BIMODAL but, on the other hand, it also ensures that the source vertex of a directed edge is always placed below the target vertex thus having an impact on the relative geometric positions of vertices. Recall that each upward drawing is also a bimodal drawing. Hence, constraints BIMODAL and FLOW can be combined without any problems. When constraints BIMODAL and PORT/SIDE are

applied to different edge sets, they can also be combined. The same holds for constraints PORT/SIDE and FLOW.

Combining constraints FLOW and CLUSTER might introduce drawing conflicts because there are graphs which have an upward drawing and a cluster drawing but not an upward cluster drawing (Fig. 3.9). The user can decide which of the two constraints is more important. If CLUSTER is chosen to be more important, we always produce a cluster drawing. However, the number of edges which can be drawn upward might decrease in that case.

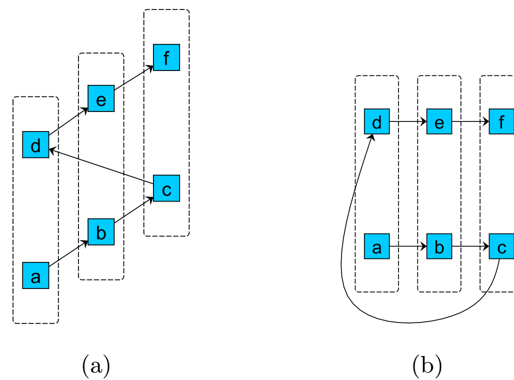


Figure 3.9: Example of a graph which has an upward drawing (a) and a cluster drawing (b) but not both at the same time (dashed rectangles represent clusters). Note that in the upward drawing (a) the vertical ordering of the vertices is fixed. The clusters have to be arranged horizontally because they should be drawn as rectangles and are not allowed to overlap each other. Hence, each upward drawing contains an edge which crosses the boundary of the middle cluster twice, which is not allowed in a cluster drawing.

Fig. 3.10(a) shows an example of incompatibility when we combine constraints PARTITION and FLOW. Edge (a, b) could not be drawn upward because the partition restricts the source vertex a to be above the target vertex b. Note that the partition is given as input and thus is not allowed to be changed or rotated. We assume that if partitions are used, they are more important than constraint FLOW.

Fig. 3.10(b) shows a conflict which could arise when we combine constraints PARTITION and CLUSTER. Cluster a contains vertex c and f, cluster b contains vertex e and d, and the assignment of the vertices to partitions is as shown in the figure. If the clusters should not be drawn nested, we cannot realize a valid partitioned cluster drawing. Our approach avoids those conflicts in the following way: cluster regions are not allowed to cross partition cells. Thus, each compound vertex can be uniquely assigned to a partition cell. Under this assumption, each graph has a partitioned cluster drawing. Table 3.1 summarizes the drawing compatibilities.

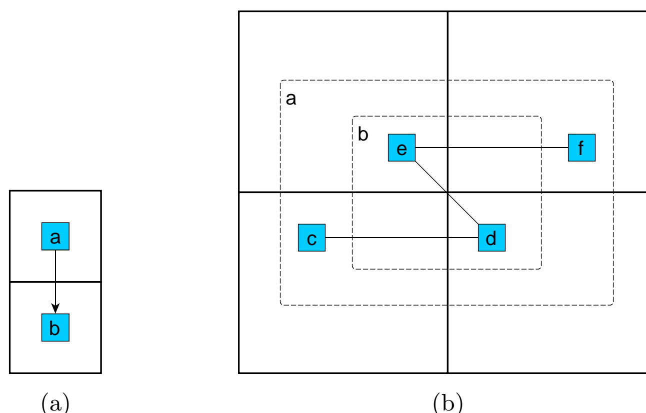


Figure 3.10: Possible conflicts when we combine constraints PARTITION and FLOW (a) and constraints PARTITION and CLUSTER (b).

	BIMODAL	FLOW	CLUSTER	PARTITION	PORT/SIDE
BIMODAL	-	YES	YES	YES	YES[4]
FLOW	YES	-	NO[1]	NO[2]	YES[4]
CLUSTER	YES	NO[1]	-	NO[3]	YES
PARTITION	YES	NO[2]	NO[3]	-	YES
PORT/SIDE	YES[4]	YES[4]	YES	YES	-

[1] see Fig. 3.9; [2] see Fig. 3.10(a); [3] see Fig. 3.10(b);
[4] when applied to different edge sets

Table 3.1: Drawing compatibilities.

In order to simplify the description of our algorithm we only consider regular partitions here. Irregular partitions lead to some special cases during the planarization phase. However, the challenges arising are only minor technical issues and do not require new concepts.

3.2.2 Related Work

In the preceding sections we introduced five important drawing constraints which arise in several practical applications. While for some of them there are already planarization and orthogonalization approaches, there is still no approach which is able to incorporate multiple constraints at the same time. The only exception is the integer linear program (ILP) based orthogonalization approach described in [66], which is able to include various constraints including FLOW, SIDE and PORT. The drawback of this approach is that solving the corresponding ILP is NP-complete and thus it cannot be efficiently applied to larger graphs.

Up to now we have only considered approaches which implement phases of the TSM approach. In the following, we take a look at the constraint-handling capabilities of two other popular drawing approaches, namely Sugiyama's approach introduced in Section 2.4, and the so-called force-directed approaches.

Due to its conceptual design, Sugiyama's approach is especially suited to satisfy constraint FLOW. Furthermore, it allows a natural modeling of geometrical constraints since the layering phase allows control over the y-coordinates and the crossing minimization and horizontal coordinate assignment phase control over the x-coordinates. Note that Sugiyama's approach never produces overlapping graph elements. It is highly adaptable and a lot of variants/extensions have been introduced to handle application-specific requirements like compound graphs [128, 144] and constraints on the vertex order [76, 151]. It has been successfully applied to different areas like visualization of UML class diagrams [61, 62, 132], statecharts [33, 34], biochemical pathways [7, 25] and compiler graphs [129].

"SugiBib" [62, 63, 132] is a sophisticated, Sugiyama-based layout algorithm for drawing UML class diagrams. It is able to simultaneously consider constraints CLUSTER and FLOW. However, it is rather weak for basic properties like VERTEX_SIZE or ORTHOGONAL, which can be realized in a more natural way by a TSM-based approach. As shown in [64] for mixed-upward drawings of mixed graphs, TSM-based methods produce considerably fewer crossings than Sugiyama-based approaches as well as fewer bends. In our new approach we will take advantage of the capabilities of Sugiyama's approach. More precisely, we will use Sugiyama's approach for an intermediate step during the planarization phase.

Another class of drawing approaches are the so-called force-directed approaches (see [23] for a comprehensive overview). In those approaches, a graph is seen as a physical system in which the graph elements represent interacting physical objects. The objects are subject to forces (e.g., spring, gravitational or magnetic forces) acting on or between them.

The spring-embedder approach [53] is based on the following physical model: edges are associated with mechanical springs which attract the edges' endpoints. Furthermore, vertices are associated with charged particles and thus repel each other if they get too close together. A minimum energy (equilibrium) state of the physical system corresponds to a readable layout where adjacent vertices are placed close together.

There are a lot of refinements and variants of force-directed approaches [41, 80, 81, 101]. An experimental comparison of the different methods is presented in [22]. Force-directed approaches are particularly used to produce straight-line drawings of undirected graphs with unknown structure. They are especially suited to identify and display symmetries.

Force-directed approaches are popular for the following reasons:

- They are quite intuitive because they are based on physical analogies. Thus, the behavior of the algorithms is relatively easy to predict and understand.
- Compared to other drawing approaches, force-directed algorithms are typically simple and easy to implement.
- Force-directed approaches are highly flexible and can easily be extended to satisfy advanced requirements, like FLOW [145], CLUSTER [54, 154], 3D layouts [30] as well as positioning constraints [94] (e.g., RELATIVE_POS and ABSOLUTE_POS).

If CROSSING is the main requirement, classical planarization approaches seem to be more appropriate. Furthermore, realizing requirements ORTHOGONAL and BEND is quite difficult here.

3.3 Constraint-Kandinsky

In this section, we present the interface of `Constraint-Kandinsky`, our new algorithm for the automatic layout of graphs with constraints. Furthermore, we give an overview of its individual steps. `Constraint-Kandinsky` generates orthogonal drawings (ORTHOGONAL) in the `podavsnef` model, which allows vertices of arbitrary size (VERTEX_SIZE) as well as vertices of arbitrary degree. Besides constraints BIMODAL, FLOW, CLUSTER, (regular) PARTITION and SIDE/PORT presented in this chapter, the approach incorporates the aesthetics CROSSING, BEND, AREA and OVERLAP. Furthermore, it also satisfies the following requirements:

- Several diagram types contain elements which correspond to vertices whose incident edges can only be connected on two opposing sides as shown in Fig. 3.11(a) (TWO_SIDED_VERTICES). Examples of those elements are buses in wiring diagrams, transitions in Petri nets and join/fork nodes in activity diagrams. Here, we add the restriction that all edges incident to a two-sided vertex v have to be directed and that all incoming edges are attached to one side and the outgoing edges to the other side (v is bimodal). Furthermore, the edges are not allowed to have port/side constraints on v.
- An important requirement is a suitable handling of labels of graph elements (LABEL). While we assume that vertex labels are placed inside the corresponding vertices and thus do not need special consideration, handling edge labels is a challenging task. An edge label can be placed near the corresponding source/target vertex or the center of the corresponding edge. Our algorithm also supports labeling of clusters and partitions. In order to increase readability we prefer a horizontal alignment of labels (HORIZONTAL_LABELS).

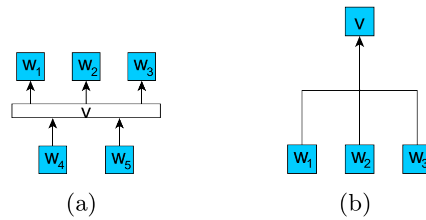


Figure 3.11: In (a) vertex v represents a bus where edges can only be attached to the top and bottom border. (b) shows a hyperedge structure.

- A notation element which appears in several diagrams is a note (NOTE). Notes are often represented as rectangles and are used to comment on diagram elements. Unlike labels, a note can be seen as a special kind of vertex which is often attached to the corresponding element by an edge. Notes are, for example, available in all kinds of UML diagrams. For our approach, we use the restriction that each note has only one corresponding graph element. Of course, a graph element can have multiple notes.

- Some diagrams contain edge structures which correspond to so-called hyperedges (HYPEREDGES). A hyperedge is an edge connecting more than two vertices and thus allows specifying non-binary relations. In our approach, we only consider directed hyperedges with exactly one source vertex and multiple target vertices as well as one target vertex and multiple source vertices. This kind of hyperedge is often presented as in Fig. 3.11(b), e.g., for generalization relations in class diagrams.

To our knowledge there is no existing layout algorithm that is able to handle such a complex combination of requirements.

3.3.1 Input and Interface

The input of our layout algorithm is:

- A mixed compound graph $G_C = (G, T)$, $G = (V, E)$, $T = (V, E_T)$, $E = E_D \cup E_U$, $V = B \cup C$ (B denotes the set of base vertices and C the set of compound vertices). We assume that for each vertex $v \in V$ is $\delta(v) = O(|V|)$.

- A set of directed edges $E_\uparrow \subseteq \{(v, w) \in E_D \mid v, w \in B\}$ that should be drawn upward.

- A regular partition P_R of the drawing area with R, the set of partition cells.

- A function $p: V \to R$ assigning each vertex to a partition cell. Furthermore, we use two functions $px: V \to \mathbb{N}$ and $py: V \to \mathbb{N}$ to map each vertex to the column/row index of the corresponding partition cell. The first (leftmost) column and the first (topmost) row both have index 0. Since we assume that each column/row contains at least one vertex we have $|R| = O(|V|^2)$.
- A set of side constraints SC_G and a set of port constraints PC_G. We do not allow specifying port/side constraints on compound vertices.
- Sets $H_v^- \subseteq E_D$ ($H_v^+ \subseteq E_D$) of directed incoming (outgoing) edges with target (source) $v \in B$ that should be represented as hyperedges. We assume that $|H_v^-|$ ($|H_v^+|$) ≥ 2.
- A function *type*: $B \to \{$`common`, `note`, `two_sided`, `bimodal`, `note_dummy`, `hyper_dummy`$\}$, denoting the type of the base vertices (we immediately explain the vertex types `hyper_dummy` and `note_dummy`). In the resulting drawing all vertices of type `two_sided` and `bimodal` have to be bimodal.
- A set of labels $L = L_E \cup L_B \cup L_C \cup L_R$ where L_E denotes the set of edge labels, L_B the set of vertex labels, L_C the set of cluster labels (i.e., labels of compound vertices) and L_R the set of partition cell labels.
- A function *refer*: $L \to E \cup B \cup C \cup R$ assigning each label to a graph element.
- A function *pos*: $L_E \to \{$`center`, `source`, `target`$\}$ denoting the preferred position of an edge label along the corresponding edge.
- A function *size* : $B \cup L \to \mathbb{N} \times \mathbb{N}$ denoting the size of the base vertices and labels in the drawing.

We use edge set $E_\uparrow$ to distinguish between directed edges that should be drawn upward and those that need not be drawn upward. While constraint FLOW only applies to edges of $E_\uparrow$, BIMODAL includes all edges of E_D. The algorithm can be customized to fit individual user preferences and different views of a diagram. The user can, for example, decide if PARTITION should be applied, if CLUSTER is more important than FLOW or if FLOW should not be considered at all.

In the remainder of this work we also use the following vertex/edge sets which can be derived from the above input:

- A vertex set $V^b = \{v \in B \mid type(v) = \texttt{bimodal} \lor type(v) = \texttt{two_sided}\}$
- An edge set $E^b = \{(v, w) \in E_D \mid v \in V^b \lor w \in V^b\}$

- An edge set $E^c = \{e \in E \mid e$ has a port/side constraint$\}$
- An edge set $E^* = E \setminus E_\uparrow$
- An edge set $E^*_D = E_D \setminus E_\uparrow$

According to the drawing compatibilities shown in Table 3.1, we assume that $E^c \cap E_\uparrow = \emptyset$ as well as $E^c \cap E^b = \emptyset$.

3.3.2 Overview

We conclude this chapter with an overview of the individual steps of algorithm `Constraint-Kandinsky`:

1. **Preprocessing**

   (a) We remove all self-loops from the input graph G.

   (b) We use common edges to model hyperedges by altering the input graph as follows: for each set H^-_v of incoming hyperedges of a vertex $v \in B$ we insert a dummy vertex d into B and replace the edges of H^-_v by an edge (d, v) and edges $\{(w, d) \mid (w, v) \in H^-_v\}$ (Fig. 3.12(a)). Analogously, for each set H^+_v of outgoing hyperedges, we insert a dummy vertex d and replace the edges of H^+_v by an edge (v, d) and edges $\{(d, w) \mid (v, w) \in H^+_v\}$. The type of vertex d is set to `hyper_dummy`. Furthermore, we assign d to the same cluster and partition as v (this implies that d gets the same parent in T as v).

   (c) For each vertex $u \in B$ of type `note` which is not connected to its corresponding graph element q, we temporarily insert an undirected edge (u, q). Note that q is either a vertex or an edge element. In order to obtain a valid graph structure, we have to transform the graph such that it only contains edges connecting vertices. Hence, for each edge $e = (u, q)$, where $q = (v, w)$ also denotes an edge, we insert a dummy vertex d into B and replace e and q by edges (u, d), (v, d) and (d, w) as shown in Fig. 3.12(b). Furthermore, we set $type(d)$ to `note_dummy` and assign d to the same cluster and partition as v.

   (d) If the subgraph induced by the vertices of B is not connected, we make it so by adding additional undirected edges between its connected components.

   (e) For each edge $e = (v, w) \in E_\uparrow$, we compare the row indices of the corresponding endpoints. If $py(v) < py(w)$, e cannot be drawn upward and thus is removed from $E_\uparrow$. Note that e is still in E_D.

(f) If the subgraph $(V, E_\uparrow)$ contains cycles, we calculate a feedback arc set $A \subset E_\uparrow$ as described in Section 2.4.1. All edges of A are removed from $E_\uparrow$ (they are still in E_D).

2. **Planarization**

   We apply the planarization approach described in Chapter 4.

3. **Orthogonalization**

   We apply the orthogonalization approach described in Chapter 5.

4. **Compaction**

   (a) All self-loops removed during the preprocessing step are reinserted as described in Section 5.5.

   (b) All labels except edge labels $l \in L_E$ with $pos(l) = \texttt{source}$ or `target` are inserted into the orthogonalized graph as described in Section 5.6.

   (c) The compaction phase uses the fast constructive compaction algorithm, described in [68], which is able to handle vertices of prescribed size (VERTEX_SIZE). Furthermore, we use the flow-based visibility postprocessing strategy described in [111] to further optimize aesthetics AREA. During the compaction all dummy vertices, except those representing edge labels, are assigned a size of one.

5. **Postprocessing**

   (a) All dummy vertices and edges inserted during the previous steps are removed.

   (b) Edge labels $l \in L_E$ with $pos(l) = \texttt{source}$ or `target` are placed by an efficient map-labeling algorithm [153].

   (c) Rectangles denoting cluster regions as well as the grid partition are inserted into the drawing.

Let x denote the number of crossings and b the number of bends in the final drawing. Furthermore, let N denote the input size of the layout algorithm. All steps of the preprocessing phase can be performed in time linear in N. The runtime of the compaction phase is dominated by the flow-based postprocessing algorithm which can be performed in time $O((N+x+b)^2 \log(N+x+b))$. The runtime of the postprocessing phase is as follows: We assume that the number of label candidates (places where a single label can be placed) is constant. Under this assumption, we can perform the map labeling in time $O(|L_E|^2)$ (see [153]). The remaining steps can be performed in time $O(N+x+b)$.

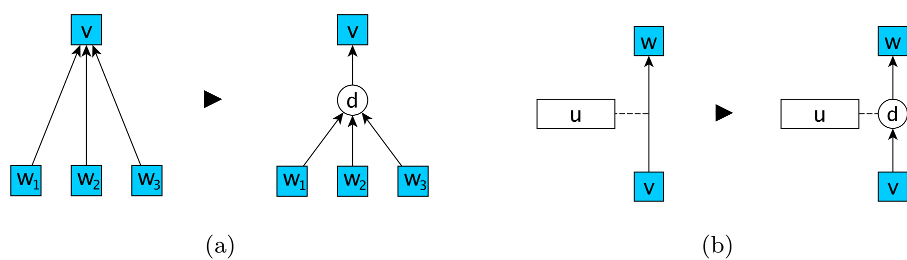


Figure 3.12: (a) shows the transformation of hyperedges and (b) the transformation of notes attached to edges.

In the above overview we presented the individual steps of the compaction phase as well as the pre- and postprocessing. The planarization and orthogonalization phase both require substantial modifications and extensions to handle our set of constraints. They are addressed in detail in the next two chapters.

Chapter 4

Planarization with Constraints

In this chapter we introduce our new generic planarization framework, which is capable of including all drawing constraints described in the last chapter. It combines the layered approach of Sugiyama with a rerouting strategy known from TSM-based approaches. The result of the planarization is a port constraint preserving bimodally mixed-upward p,c-planar embedding of the input graph. This chapter is partly based on our previous work described in [71, 135, 138].

First, we introduce the so-called *orientation problem* – an issue that arises during the layout process. Afterwards, we describe the basic concept behind our planarization approach. Since it was originally designed to produce mixed-upward planar embeddings of mixed graphs, we introduce it by means of mixed-upward planarization. Then we show how to incorporate the remaining constraints BIMODAL, CLUSTER, PARTITION and PORT/SIDE into this approach.

In the last section of this chapter, we present a fast implementation of Sugiyama's approach. It provides a significant runtime improvement over existing implementations and, hence, also speeds up our Sugiyama-based planarization step.

4.1 The Orientation Problem

Before we apply our planarization framework to the input graph, we have to consider the following issue: our orthogonalization approach is based on a traditional minimum cost flow formulation. A problem that arises when we use such an approach is that, up to now, it has not been possible to directly include the orientation of the vertices into the flow formulation, i.e., we cannot specify on which side an edge should enter/leave a vertex. Hence, in order to include constraints FLOW and PORT/SIDE, we have to unify the orientation of each vertex such that all edge segments restricted to the same side point in the same direction for each vertex. We call this issue the *orientation problem*.

Let $G = (V, E)$ denote the connected mixed input graph and $E_\uparrow \subseteq E$ the set of edges that should be drawn upward. Furthermore $G' = (B, E')$ denotes the subgraph of G induced by the vertices of B. Recall that our preprocessing step guarantees that G' is always connected. Only vertices of B require a uniform orientation since they are the only vertices incident to upward edges or edges with port/side constraints. Our handling of the orientation problem is based on that described by Eiglsperger et al. [67]. First we have to calculate a spanning tree $T^s = (B, E^s)$ of G'. All edges of E^s are treated as upward edges, i.e., they are embedded upward planar. During the orthogonalization phase, we assign a fixed upward shape to these edges which is not allowed to change during the orthogonalization phase. The edges of E^s are called *skeleton edges* since they define a fixed skeleton of the drawing and thus allow us to determine the orientation of the vertices. More details on how to fix the shapes of these edges are given in Section 5.3.

Since skeleton edges have to be embedded upward planar, we already have to consider the orientation problem during the planarization phase. The choice of edges for the spanning tree T^s is affected by the handling of skeleton edges. Edges of $E_\uparrow$ are especially suited to be in T^s because such edges have to be embedded upward planar in any case. Edges of E^c must not be handled like upward edges since the shape assignment of upward edges is not compliant with that of edges with port/side constraints. The same holds for directed edges $(v, w) \in E_D$ with $py(w) > py(v)$. These edges could not be drawn upward without violating the geometrical constraints given by the partition. Let E^p denote the set of such edges. We include the different preferences by assigning different weights to the edges and use a minimum spanning tree algorithm to calculate $T^s = (B, E^s)$. Let $\omega_s \colon E' \to \mathcal{N}$ denote the weight function. For each edge $e \in E'$,

$$\omega_s(e) = \begin{cases} 1 & \text{if } e \in E_\uparrow, \\ 2 & \text{if } e \in E' \setminus (E_\uparrow \cup E^c \cup E^p), \\ 3 & \text{if } e \in E^c \cup E^p. \end{cases}$$

With Prim's algorithm [37], the calculation of T^s can be done in time $O(|B| \log |B| + |E'|)$. Let G^+ denote the induced subgraph of G' on the edges of $E_\uparrow \cup E^c$. Since uniform orientation is required only for vertices incident to upward edges or edges with port/side constraints, we can iteratively remove edges (v, w) from E^s if $(\delta_{T^s}(v) = 1 \wedge \delta_{G^+}(v) = 0)$ or $(\delta_{T^s}(w) = 1 \wedge \delta_{G^+}(w) = 0)$. This may reduce the number of required skeleton edges. Now we add all edges of E^s to $E_\uparrow$. Recall that we have to guarantee that $E_\uparrow \cap E^p = \emptyset$ and $E_\uparrow \cap E^c = \emptyset$. Hence, for each edge $e \in E^s \cap (E^c \cup E^p)$ we add a clone of e into $E_\uparrow$. If $e \in E^p$, we additionally have to reverse the cloned edge. The inserted clones can be removed after the orthogonalization phase since the edge shapes and thus the orientation is fixed then. For undirected edges added to $E_\uparrow$ we have to assign a valid direction which conforms to the given partition. Due to the chosen weights and the properties of a minimum

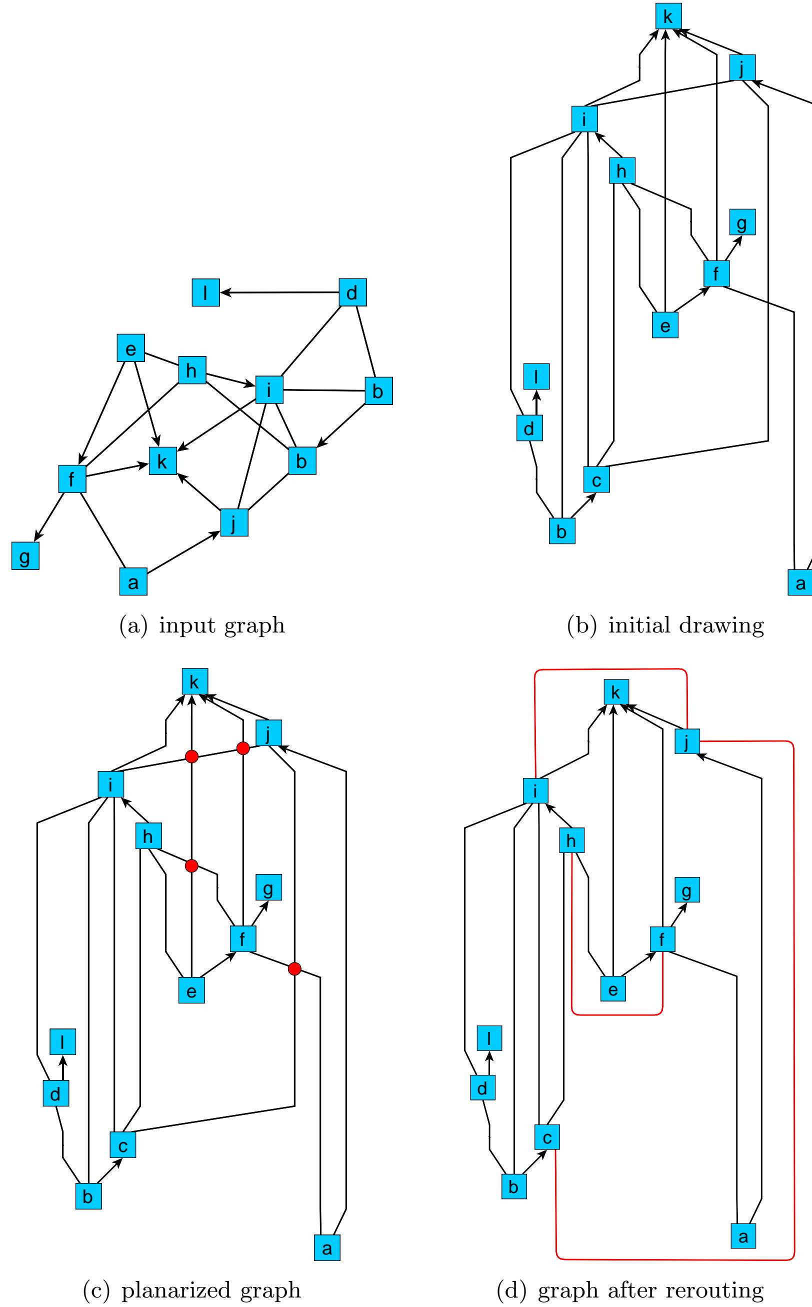
spanning tree, the resulting subgraph $(B, E_{\uparrow})$ is always acyclic. Let δ_G^{max} denote the maximum degree of a vertex v of G after inserting the cloned edges. We assume that κ is chosen such that $\kappa \geq 2\delta_G^{max} - 1$. Note that the number of cloned edges incident to a vertex v is bounded by v's original degree.

4.2 Mixed Upward Planarization

In the following, we introduce our generic planarization framework and show how to apply it to calculate a mixed-upward planar embedding of a connected mixed input graph G. Basically, our planarization framework consists of the following phases:

1. **Construction of an initial drawing:** We use a Sugiyama-based layout approach to construct a preliminary drawing of G which includes the specified constraints.
2. **Construction of a planar embedding:** We use a sweep-line algorithm to determine the embedding given by the initial drawing.
3. **Rerouting of edges:** In order to reduce the number of crossings, we successively reroute edges.

Fig. 4.1 illustrates our framework for mixed-upward planarization. Unlike previous planarization approaches we derive the planar embedding of G by means of an initial drawing. This offers a more suitable way to include additional constraints like PARTITION and CLUSTER into the planarization phase. To construct the initial drawing, we have to take a drawing approach which is highly adaptable and allows us to realize the different drawing constraints. Hence, we use Sugiyama's approach, which is especially suited to satisfy constraint FLOW. For acyclic input graphs the resulting drawing is always an upward drawing. There are sophisticated approaches for handling constraint CLUSTER. Furthermore, constraints PARTITION and BIMODAL can be included in a natural way. Note that after the first phase we already have a drawing that satisfies our set of drawing constraints. However, when we further reduce the number of crossings and take the resulting embedding as input for a TSM-based approach, we can also easily include requirements like VERTEX_SIZE, LABEL, ORTHOGONAL and ANGLE. Recall that in TSM-based approaches, vertices are not placed on fixed layers. The experimental evaluation given in [64] shows that for mixed-upward drawings of graphs the results produced by TSM-based approaches clearly outperform those of Sugiyama-based approaches with respect to aesthetics CROSSING and BEND. However, Sugiyama-based approaches usually require significantly less area. This is not surprising, since, in general, aesthetic criteria BEND and AREA as well as CROSSING and AREA are conflicting



(a) input graph

(b) initial drawing

(c) planarized graph

(d) graph after rerouting

Figure 4.1: Illustration of the three phases of our planarization framework for mixed-upward planarization. All directed edges should be drawn upward. The small red circles denote crossings and the red edges mark rerouted edges.

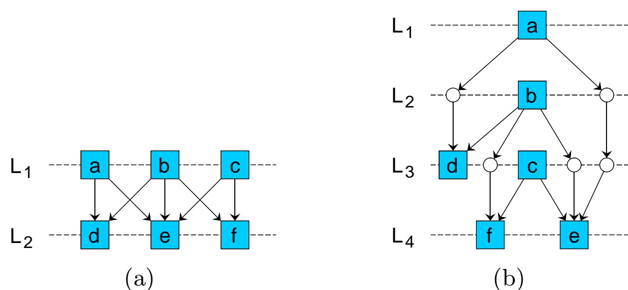


Figure 4.2: Example of two different layerings of the same graph. While the layering in (a) minimizes edge length, the two edge crossings cannot be avoided. The layering in (b) uses two additional layers and offers a crossing-free drawing. However, the overall edge length increases so that we need 5 additional dummy vertices to normalize the graph.

aesthetics. In the next subsections we describe the different phases in more detail.

4.2.1 Construction of an Initial Drawing

We use Sugiyama's approach with a customized layering strategy to create an initial mixed-upward drawing of the mixed input graph G. Recall that the subgraph induced by the edges of $E_\uparrow$ is acyclic. The layering strategy has a great impact on the number of edge crossings in the resulting drawing. Common layering strategies usually cause a high number of crossings since they are mainly optimized to produce short edges and not for minimizing the number of crossings. While aesthetic criteria EDGE_LENGTH is possibly more important for layered drawings, this is not true when we use the initial drawing just to derive a planar embedding. Hence, we introduce a new layering strategy which ignores criteria EDGE_LENGTH and leads to drawings with less crossings. An example of how to save crossings by adding additional layers is shown in Fig. 4.2.

Recall that in our approach the flow direction of edges is upward. Thus, unlike for common layerings where the flow direction of edges is top-down, it is bottom-up, here. Let $\lambda\colon V \to \mathbb{N}$ denote the function that maps vertices to layers. For each edge $(v, w) \in E_\uparrow$, we have to guarantee that $\lambda(v) > \lambda(w)$ (layers are still numbered from top to bottom). For edges $(v, w) \in E^*$ it is sufficient that $\lambda(v) \neq \lambda(w)$. Algorithm 2 gives the pseudo code for our layering strategy. In each iteration of the for-loop, we choose a vertex $v \in V$ and assign it to a new layer, i.e., for a vertex v chosen in the i-th step we set $\lambda(v) = i$. Basically, our algorithm uses a strategy similar to the first phase of the GT heuristic (see algorithm `calcGTOrdering` on page 17), i.e., it tries to place adjacent vertices on adjacent layers. In order to realize constraint FLOW, we use a

directed graph $C_G = (V, E_C)$ called the *layering constraint graph*. An edge (w, v) of C_G models the requirement that $\lambda(w)$ should be smaller than $\lambda(v)$. The function *etype*: $E_C \rightarrow \{\texttt{upward}, \texttt{bimodal}, \texttt{cluster}, \texttt{partition}, \texttt{pc/sc}\}$ maps each edge $e \in E_C$ to a type that indicates the drawing constraint which caused the insertion of e into C_G. For each edge $(v, w) \in E_\uparrow$, we insert an edge $e' = (w, v)$ into C_G and set $etype(e') = \texttt{upward}$. In order to include such layering constraints into our algorithm we use a strategy similar to the one described by Eiglsperger et al. [69], i.e., we can only choose a vertex v if all its predecessors in C_G have already been chosen. Since $(V, E_\uparrow)$ is acyclic, we can always find a feasible vertex. We store those vertices in a list *Cand* (lines 6-8). The vertex set V' contains vertices not yet placed.

Algorithm 2: `calcLayering`

Input: A mixed graph $G = (V, E)$, $E = E_D \cup E_U$ and the acyclic directed layering constraint graph $C_G = (V, E_C)$.
Output: The layering function $\lambda : V \rightarrow \mathbb{N}$.

1 $V' \leftarrow V$;
2 $G' \leftarrow G$;
3 $C'_G \leftarrow C_G$;
4 $Neighbors \leftarrow \emptyset$;
5 **for** $i = 1$ **to** $|V|$ **do**
6 $\quad Cand \leftarrow \{v \in Neighbors \mid \delta^-_{C'_G}(v) = 0\}$;
7 $\quad$ **if** $Cand = \emptyset$ **then**
8 $\quad\quad Cand \leftarrow \{v \in V' \mid \delta^-_{C'_G}(v) = 0\}$;
9 $\quad \mathcal{X} \leftarrow \{v \in Cand \mid \delta_G(v) - \delta_{G'}(v) \geq \delta_G(w) - \delta_{G'}(w) \quad \forall\, w \in Cand\}$;
10 $\quad \mathcal{Y} \leftarrow \{v \in \mathcal{X} \mid \delta_{G'}(v) - \delta^+_{C'_G}(v) \leq \delta_{G'}(w) - \delta^+_{C'_G}(w) \quad \forall\, w \in \mathcal{X}\}$;
11 $\quad v \leftarrow$ randomly chosen element of $\mathcal{Y}$;
12 $\quad \lambda(v) \leftarrow i$;
13 $\quad Neighbors \leftarrow \{w \in V' \mid w$ adjacent to $v\}$;
14 $\quad V' \leftarrow V' \setminus v$;
15 $\quad G' \leftarrow$ subgraph of G induced by V';
16 $\quad C'_G \leftarrow$ subgraph of C_G induced by V';
17 **return** λ;

Note that during a two-layer crossing minimization step, the number of crossings correlates with the number of edges between both layers. We use the following two refinements to keep the number of such edges small (and thus the number of crossings):

- Among all vertices of *Cand* we prefer those with the highest number of neighbors n with $n \in V \setminus V'$ and store them in a set $\mathcal{X}$ (line 9).

- Among all vertices of $\mathcal{X}$ we prefer those with the fewest number of neighbors $n \in V'$ as well as those blocking a lot of other vertices, i.e., vertices v with large out-degree $\delta^{+}_{C'_G}(v)$ (line 10).

Before we apply algorithm `calcLayering`, we temporarily remove all degree-one vertices, i.e., vertices $v \in V$ with $\delta_G(v) = 1$, and reinsert them afterwards as follows: if $(u, v) \in E_D$, we insert v in a new layer lying between layer $\lambda(u) - 1$ and layer $\lambda(u)$. Otherwise, we insert v in a new layer lying between layer $\lambda(u)$ and $\lambda(u) + 1$. Thus, as long as we do not include further constraints, all edges incident to degree-one vertices never have crossings.

The experiments in Section 7.3 verify that our layering strategy induces drawings which have significantly fewer crossings than those resulting from common layering approaches. They also show that this behavior does not solely depend on the sparse layering (only one vertex per layer). Note that we can still use a common layering strategy as follows: First, we make C_G acyclic. Then, for each edge (v, w) of C_G, we insert an edge (w, v) into G (we have to reverse the edges because the flow direction is upward here). Let E' denote the set of inserted edges. We assign weight $|E|$ to edges of E' and weight 1 to the remaining edges. Then we use the weighted feedback arc set heuristic described in Section 2.4.1 to identify an edge set A of edges that have to be reversed to make G acyclic. From Lemma 2.10 and the chosen weights, we always have $A \cap E' = \emptyset$. Now we apply a common layering strategy to G. Obviously, the resulting layering observes all constraints given by edges of E'. Note that after the layer assignment we remove those edges.

After the normalization of the layered graph, we apply the iterative layer-by-layer sweep to reduce the number of edge crossings. During the one-sided two-layer crossing minimization, we use a linear measure, e.g., the median or barycenter heuristic. For the horizontal coordinate assignment we apply the approach of Brandes and Köpf [27] described in Section 2.4.4 which produces drawings in the linear segments model. Note that due to our layering strategy all edges of $E_\uparrow$ are drawn upward.

The runtime for constructing the initial drawing is as follows: Making C_G acyclic can be done in time $O(|V| + |E_\uparrow|)$. The for-loop of our layering algorithm is iterated $|V|$ times and each step inside the loop can be performed in $O(|V|)$ time. Thus, the runtime of the layering step is $O(|V|^2 + |E_\uparrow|)$. The crossing minimization has runtime $O(|V||E| \log |E|)$ and requires $O(|V||E|)$ space (see Section 2.4). For the horizontal coordinate assignment, the time and space complexity is linear to the size of the normalized graph, i.e., $O(|V||E|)$. Hence, the overall runtime for calculating the initial drawing is $O(|V||E| \log |E|)$ and the space requirement is $O(|V||E|)$.

4.2.2 Construction of a Planar Embedding

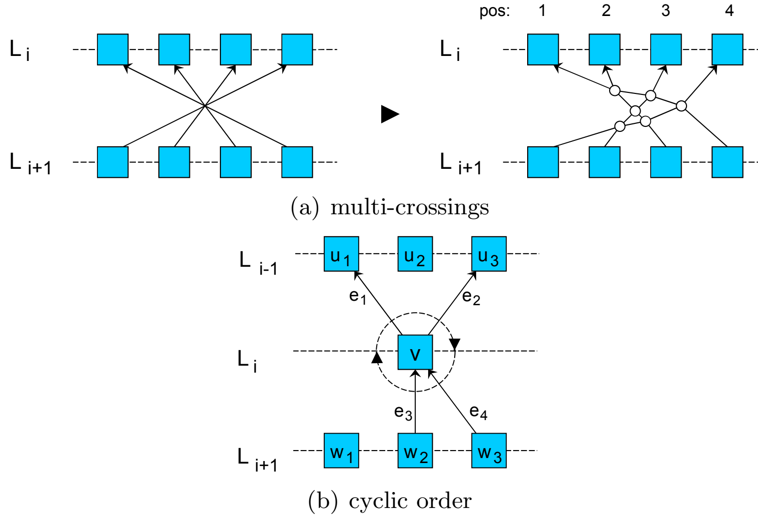
In this phase, we use the initial mixed-upward drawing to derive a mixed-upward planar embedding of G. Therefore, we have to replace crossings by dummy vertices and specify the cyclic order of the edges around each vertex.

We detect crossings by means of a sweep-line algorithm [8] which we apply to the initial drawing of G. This can be done in time $O(|S| \log |S| + x)$, where S denotes the set of segments and x the number of crossings. Since the initial drawing conforms to the linear segments model, the number of segments $|S|$ is $O(|E|)$. For each crossing, we store its coordinates and the two related edges. A crossing is modeled by splitting the corresponding edges with a dummy vertex. For an edge $e = (u, v)$, let $x_1^e, \dots, x_k^e$ denote the ordered sequence of crossings as they appear when we traverse e from u to v. If we split e in this order, we can easily identify the segment to split (the i-th crossing of (u, v) splits segment (x_{i-1}^e, v)). If the crossings are processed arbitrarily, the identification of these segments demands a more complex data structure. Note that in the initial drawing of the graph all edges are routed monotone. Hence, we sort the crossings according to their y-coordinate. Crossings with the same y-coordinate are additionally sorted according to their x-coordinate. Thus, crossings of horizontal edge segments are also processed in the ordered sequence. This will be relevant during the realization of constraints CLUSTER and PARTITION, where we also have to cope with horizontal edge segments.

A problem arising in this phase is that of multi-crossings, i.e., points where more than two edges cross. While those crossings are allowed in Sugiyama's approach they are not allowed in the TSM approach. Let $e_1,\dots,e_k$ denote edges participating at a multi-crossing. The sweep-line detects a crossing for each of the $\frac{k(k-1)}{2}$ edge pairs. In order to obtain a valid embedding, those crossings have to be processed in a suitable order. Hence, crossings with the same x- and y-coordinate are additionally sorted as follows: First we sort them according to the smaller layer position of the endpoints of the two participating edges in the upper layer. Crossings with the same value are additionally sorted according to the larger layer position of these endpoints. This processing order always leads to a valid elimination of multi-crossings as shown in Fig. 4.3(a).

After inserting crossing vertices, the cyclic order of edges around a vertex v can easily be determined using the layer positions of v's neighbors. This is illustrated in Fig. 4.3(b), where the cyclic order around vertex v is e_1, e_2, e_4, e_3.

If G contains multi-edges, we also have to consider the following issue: Let $E_{v,w}^m$ denote the set of multi-edges between two vertices v and w. If v and w lie on adjacent layers, all edges of $E_{v,w}^m$ overlap each other (in the initial drawing) since they are drawn straight-line. We detect those cases



(a) multi-crossings

(b) cyclic order

Figure 4.3: Planarization issues: (a) illustrates how to eliminate multi-crossings, and (b) how to determine the cyclic order of edges around vertices.

and embed all edges of $E^m_{v,w}$ in parallel, i.e., such that they all cross exactly the same edges in the same order.

Sorting crossings according to their coordinates can be done in time $O(x \log x)$ and sorting edges around vertices in time $O(|E| \log |E|)$. Thus, the overall runtime of this phase is $O(|E| \log |E| + x \log x)$.

4.2.3 Rerouting of Edges

The initial drawing produced by Sugiyama's approach is too restrictive because each edge is routed monotonically, which is not required for edges of E^*. Thus, we perform a rerouting step to further reduce the number of crossings. For each edge of E^* with at least one crossing, we perform a shortest path routing (see Section 2.3.1) and if we find a route with fewer crossings we take this route and discard the old one. Note that we always reroute a complete edge and not just segments. To further improve quality, we iteratively repeat the rerouting step and process the edges in randomized order. The size of the planarized graph is $O(|V| + x)$ and thus the runtime of the rerouting is $O((|V| + x)|E^*|)$.

Summing the runtime and space complexity of the single planarization phases we obtain the following theorem:

Theorem 4.1 *Let $G = (V, E)$, $E = E_D \cup E_U$ denote a connected, mixed graph where the subgraph induced by the edges of $E_\uparrow$ is acyclic. The above approach calculates a mixed-upward planar embedding of G in time $O(|V||E| \log |E| + x \log x + x|E^*|)$ where x denotes the number of crossings in the initial drawing. The space complexity is $O(|V| + x)$.*

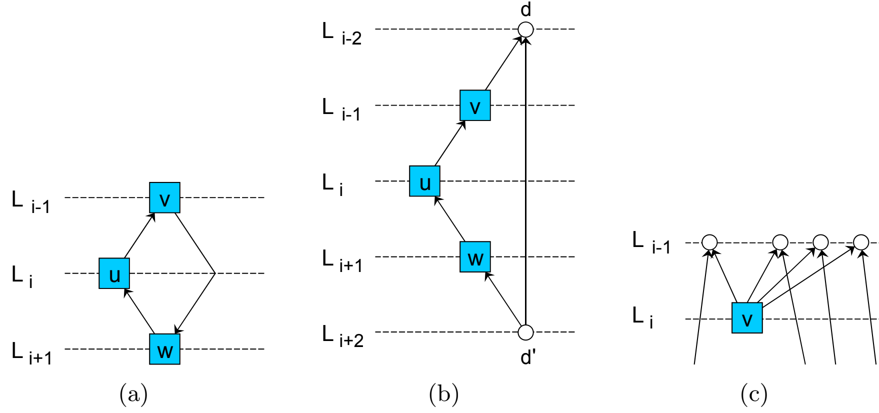


Figure 4.4: Handling of back edges to maintain bimodality. (a) shows the original graph that contains a back edge $e = (v, w)$. Instead of reversing e we apply the transformation shown in (b), which guarantees a bimodal handling of e, i.e., e enters w at the bottom and leaves v at the top. (c) demonstrates that this transformation may cause additional crossings.

4.3 Bimodal Planarization

All directed edges $e \in E_D$ around vertices of V^b have to be embedded bimodally. In order to incorporate constraint BIMODAL into Sugiyama's approach we have to adapt the layering phase as follows: For a given layer assignment λ let $A_D \subset E_D$ denote the set of *back edges*, i.e., directed edges (v, w) for which $\lambda(v) < \lambda(w)$ (Fig. 4.4). For each back edge $(v, w) \in A_D$ with $v \in V^b$, we insert a dummy vertex d into layer $\lambda(v) - 1$ and replace (v, w) by two directed edges (v, d) and (w, d). Analogously, if $w \in V^b$ we insert a dummy vertex d' into layer $\lambda(w) + 1$ and replace (v, w) (or (w, d)) by two edges (d', w) and (d', v) (or (d', d)). This transformation is sufficient to guarantee that the resulting drawing is bimodal. There are no changes required for the crossing minimization or horizontal coordinate assignment phase.

Until now, the layering constraint graph C_G has only contained edges induced by upward edges $e \in E_{\uparrow}$. To reduce the number of back edges, we insert additional edges into it. Minimizing the number of back edges is advantageous since they often cause additional crossings, e.g., if there are many such edges incident to the same vertex as shown in Fig. 4.4(c). Furthermore, the handling of those edges produces additional vertices with zero in- or out-degree. For such vertices d it is more difficult to find a suitable layer position since the measure $m(d)$ cannot be derived from the positions of d's neighbors. Hence, for each edge $(v, w) \in E_D^*$ we additionally insert an edge $e' = (w, v)$ into C_G if v or $w \in V^b$. Furthermore, we set $etype(e') =$ `bimodal`. If C_G is cyclic, we remove its cycles by applying a

weighted feedback arc set heuristic (see Section 2.4.1). Therefore, we assign the following weights to edges of C_G ($\veebar$ denotes the XOR operator):

$$\omega_c(v,w) = \begin{cases} 2|E_C| & \text{if } etype(v,w) = \texttt{upward}, \\ 2 & \text{if } etype(v,w) = \texttt{bimodal} \ \wedge \ v,w \in V^b, \\ 1 & \text{if } etype(v,w) = \texttt{bimodal} \ \wedge \ (v \in V^b \ \veebar \ w \in V^b). \end{cases}$$

Let $A \subset E_C$ denote the resulting feedback arc set. From Lemma 2.10 and the chosen weights, we have $A \cap \{e \in E_C \mid etype(e) = \texttt{upward}\} = \emptyset$. Thus, we still obtain a proper layering for the upward edges.

Dummy vertices inserted during the transformation step are removed after the calculation of the planar embedding of the initial drawing. This guarantees that, during the embedding phase, edge routes can still be considered to be monotonic. In the example of Fig. 4.4(b), we restore the transformed edge (v,w) by concatenating segment (v,d), the reverse of segment (d',d) and segment (d',w).

During the rerouting phase we reroute edges of E_D^*. We only have to guarantee that the resulting edge order around the vertices of V^b remains bimodal. The valid positions of an edge (v,w) in the cyclic edge order around the vertices v as well as w can be calculated in time $O(\delta_G(v) + \delta_G(w))$. We temporarily remove edges from the dual routing graph that might lead to invalid edge orders.

The number of additional dummy vertices inserted after the layering phase is $O(|E_D|)$. Hence, the runtime for the first two planarization phases stays the same as for the mixed-upward planarization described in the previous section. Removing cycles from C_G requires time $O(|V||E_D|)$ [42]. Since all edges except those of $E_\uparrow$ can be rerouted, the rerouting step runs in time $O((|V| + x)|E^*|)$. Summing, the overall runtime is $O(|V||E|\log|V| + x\log x + x|E^*|)$ and the space requirement $O(|V| + x)$.

4.4 Planarization of Clusters and Partitions

In this section, we show how to adapt our planarization framework to incorporate constraints CLUSTER and PARTITION. We consider both constraints simultaneously because they demand a similar treatment. Recall that the input is a mixed compound graph $G_C = (G,T)$ with $G = (V,E)$, $T = (V, E_T)$, $E = E_D \cup E_U$ and $V = B \cup C$. Furthermore, let $r \in C$ denote the root of T.

4.4.1 Construction of an Initial Drawing

For the construction of an initial drawing that includes constraints CLUSTER and PARTITION we have to modify the single phases of Sugiyama's approach.

Similar to the approach described in [128], we first transform G as follows: Let $B' = \{c^t, c^b \mid c \in C\}$ denote the vertex set that contains, for each compound vertex $c \in C$, two vertices c^t and c^b that represent the top and bottom boundaries of the corresponding cluster region. The graph G now consists of the vertex set $V' = B \cup B'$. All incoming edges $(v, c) \in E$, $v \in V$ are replaced by edges (v, c^t) if there is a path $c \rightarrow_T^* v$ (the cluster c contains v) and by edges (v, c^b) otherwise. Analogously, all outgoing edges $(c, v) \in E$, $v \in V$ are replaced by edges (c^b, v) if there is a path $c \rightarrow_T^* v$ and by edges (c^t, v) otherwise. Furthermore, we replace each edge $(c, d) \in E$, $c, d \in C$ with an edge (c^b, d^b) if $c \rightarrow_T^* d$, with an edge (c^t, d^t) if $d \rightarrow_T^* c$ and with an edge (c^t, d^b) otherwise. For each vertex $c^{\{t,b\}} \in B'$ we set $type(c^{\{t,b\}}) = \texttt{bimodal}$ and insert an edge $(c, c^{\{t,b\}})$ into T.

4.4.1.1 Layer Assignment and Cycle Removal

For a compound vertex $c \in C$, the layering has to satisfy the following condition: for each base vertex $w \in B$ with $c \rightarrow_T^* w$, $\lambda(c^t) < \lambda(w) < \lambda(c^b)$ and for each compound vertex $d \in C$ with $c \rightarrow_T^* d$, $\lambda(c^t) < \lambda(d^t) < \lambda(d^b) < \lambda(c^b)$.

Sander [128] guarantees this by introducing the concept of a *nesting graph* $G_n = (V', E_n)$. The edge set E_n contains the following edges:

- Edges (c^t, v) and (v, c^b) for each edge $(c, v) \in E_T$ with $c \in C$ and $v \in B$.
- Edges (c^t, d^t) and (d^b, c^b) for each edge $(c, d) \in E_T$ with $c, d \in C$.

Note that G_n is acyclic by construction (Fig. 4.5(b)). Each layer assignment on G_n complies with the above condition. We adopt this concept into our planarization approach by inserting edges of E_n into the layering constraint graph C_G and setting their type to `cluster`.

Let $py^{\max}$ ($px^{\max}$) denote the number of rows (columns) of the partition. We obtain a layering that preserves the partition by inserting $py^{\max} + 1$ additional vertices p_j^y into G. Let P_y denote the set of those vertices. A vertex p_j^y denotes the horizontal grid line separating grid row $j-1$ and j. The layering has to satisfy the following condition: For each base vertex $v \in B$ with $py(v) = j$, $\lambda(p_j^y) < \lambda(v) < \lambda(p_{j+1}^y)$ and for each compound vertex $c \in C$ with $py(c) = j$, $\lambda(p_j^y) < \lambda(c^t), \lambda(c^b) < \lambda(p_{j+1}^y)$. We realize this by inserting two edges, (p_j^y, u) and (u, p_{j+1}^y), into C_G for each vertex $u \in B$ with $j = py(u)$. Furthermore, we insert edges (p_{j-1}^y, p_j^y), $1 \leq j \leq py^{\max}$ into C_G to guarantee that $\lambda(p_i^y) < \lambda(p_j^y)$, if $i < j$ (Fig.4.6(b)). The type of these edges is set to `partition`.

Let $E_C^c = \{e \in E_C \mid etype(e) = \texttt{cluster}\}$, $E_C^p = \{e \in E_C \mid etype(e) = \texttt{partition}\}$ and $E_C^u = \{e \in E_C \mid etype(e) = \texttt{upward}\}$.

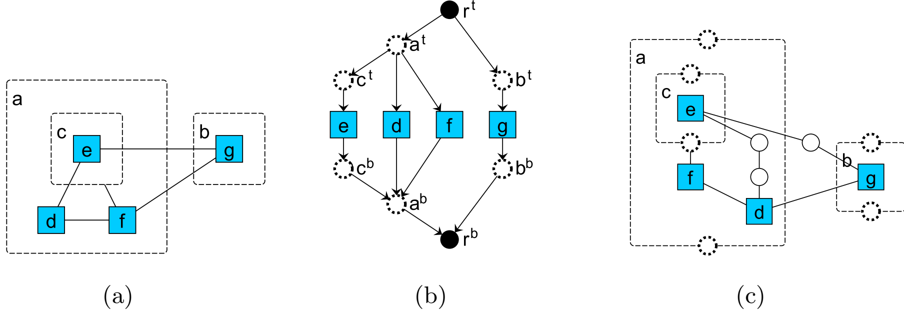


Figure 4.5: A cluster drawing (a) and the edges induced in the constraint graph C_G (b). The vertices r^t, r^b represent the top and bottom borders of the whole drawing. The assignment of the dummy vertices is shown in (c).

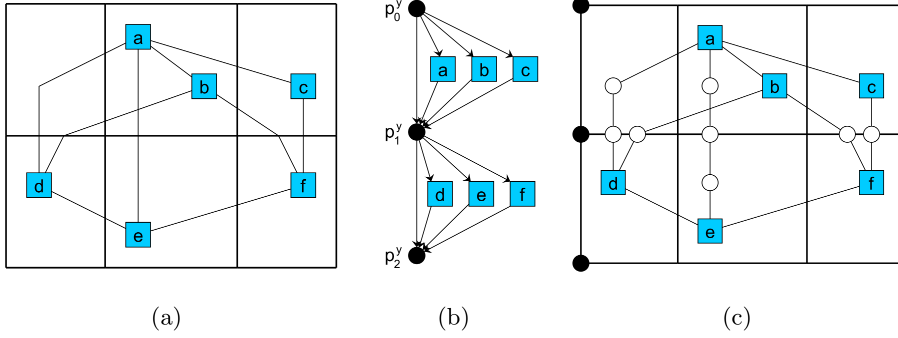


Figure 4.6: A partitioned drawing (a) and the edges induced in the constraint graph C_G (b). The insertion of the dummy vertices is shown in (c).

Lemma 4.2 *The graph $C'_G = (V', E'_C)$ with $V' = B \cup B' \cup P_y$ and $E'_C = E^c_C \cup E^p_C \cup E^u_C$ is acyclic.*

Proof: The graph C'_G is acyclic if and only if there is a permutation $\Pi_{V'}$ of the vertices of V' that induces a topological ordering. Let $V_i = \{v \in B \mid py(v) = i\}$ denote the subset of vertices of B assigned to the i-th partition row and Π_{V_i} a permutation of the vertices of V_i. Analogously, we define $V^t_i = \{v \in B' \mid py(v) = i \ \wedge \ v \text{ represents the top boundary of a cluster region}\}$ and $V^b_i = \{v \in B' \mid py(v) = i \wedge v \text{ represents the bottom boundary of a cluster region}\}$. In the following, we show that the permutation $\Pi_{V'} = r^t, p^y_0, \Pi_{V^t_1}, \Pi_{V_1}, \Pi_{V^b_1}, p^y_1, \Pi_{V^t_2}, \Pi_{V_2}, \Pi_{V^b_2}, p^y_2, \ldots, p^y_k, r^b$ with $k = py^{\max}$ induces a topological ordering, i.e., each edge $e \in E'_C$ is directed from left to right with respect to $\Pi_{V'}$. The vertices r^t and r^b are associated with the root vertex $r \in C$ and thus represent the top/bottom boundary of the whole drawing.

Obviously, $\Pi_{V'}$ induces a topological ordering for the edges of E^p_C since each vertex $v \in B \cup B'$ lies between vertex p^y_i and p^y_{i+1} if and only if $py(v) =$

i. Furthermore, in $\Pi_{V'}$ the vertices p_i^y appear in increasing order of their indices. Note that up to now we have not made assumptions about the order of the subsequences Π_{V_i}, $\Pi_{V_i^t}$ and $\Pi_{V_i^b}$.

Now we take a look at edge set E_C^c. Recall that E_C^c consists of edges (c^t, d^t), (d^b, c^b), (c^t, v), (v, c^b) with $v \in B$ and $c^t, c^b, d^t, d^b \in B'$. For vertex r^t we have $\delta^-_{C'_G}(r^t) = 0$ and for vertex r^b we have $\delta^+_{C'_G}(r^b) = 0$. Hence, due to the positioning of both vertices, edges incident to r^t or r^b are always directed from left to right. For all remaining edges $(v, w) \in E_C^c$, it is $p(v) = p(w)$ because we do not allow clusters to cross partition cells. Since E_C^c is acyclic, we can always find a valid permutation $\Pi_{V_i^t}$ that induces a topological order for the edges $(c^t, d^t) \in E_C^c$ with $py(c^t) = py(d^t) = i$. Analogously, we can find a valid permutation $\Pi_{V_i^b}$ for vertices of V_i^b. Furthermore, in $\Pi_{V'}$ the vertices of V_i are placed to the right of the vertices V_i^t and to the left of vertices V_i^b such that each edge (c^t, v) and (v, c^b) with $py(v) = py(c^t) = py(c^b) = i$ runs from left to right. Note that the order of the vertices of V_i is still not yet relevant.

Since $E_C^u \subseteq B \times B$ is acyclic there is a permutation Π_{V_i} that induces a topological ordering for the edges $(v, w) \in E_C^u$ with $p(v) = p(w) = i$. Furthermore, the permutation $\Pi_{V'}$ guarantees that each edge $(v, w) \in E_C^u$ with $p(v) < p(w)$ runs from left to right. Note that there are no edges $(v, w) \in E_C^u$ with $p(v) > p(w)$ (those edges are removed during the postprocessing step). Summing up, it follows that there is a permutation such that each edge of E'_C runs from left to right. □

We assign the following weights to the edges of E_C:

$$\omega_c(v,w) = \begin{cases} 2|E_C| & \text{if } (v,w) \in E'_C, \\ 2 & \text{if } \mathit{etype}(v,w) = \texttt{bimodal} \ \wedge \ v, w \in V^b, \\ 1 & \text{if } \mathit{etype}(v,w) = \texttt{bimodal} \ \wedge \ (v \in V^b \ \veebar \ w \in V^b). \end{cases}$$

Note that due to Lemma 4.2 and Lemma 2.10, the edge weights guarantee that $A \cap E'_C = \emptyset$ and thus we obtain a proper layering.

After the layer assignment, we normalize the graph by replacing long edges by chains of dummy vertices and edges (see Section 2.4). We have to extend the function px as follows: For each dummy vertex d_e of an edge $e = (u, v)$, we set $px(d_e) = px(v)$ if $\lambda(u) < \lambda(v)$ and $px(d_e) = px(u)$ otherwise (Fig. 4.6(c)). We also have to include the dummy vertices into the inclusion tree T. For each dummy vertex d_e of an edge $e = (u, v)$, we first determine the lowest common ancestor $w \in C$ of u and v in T. Then we assign d_e to cluster w, i.e., we insert an edge (w, d_e) into E_T. Note that this assignment is consistent with the assignment of the partition cells because if u and v are in different partition cells, the first common predecessor is always the root of the inclusion tree (recall that cluster regions are not allowed to cross partition cells).

Let p^x_j, $0 \le j \le px^{\max}$ denote the vertical grid line which separates partition column $j-1$ and j. For each p^x_j, we insert a dummy vertex $u_i^{p^x_j}$ on each layer L_i, $1 \le i \le h$. Furthermore, for each compound vertex $c \in C$, we insert two dummy vertices $u_i^{c^l}$ and $u_i^{c^r}$ on each layer L_i, $\lambda(c^t) \le i \le \lambda(c^b)$ representing the left/right boundary of the cluster region. The number of dummy vertices and edges inserted during normalization is $O(|V'||E|)$.

4.4.1.2 Crossing Reduction

We realize constraint CLUSTER by using the clustered crossing reduction approach described by Forster [74, 75]. First, a *layer cluster tree* T_{L_i} is computed for each layer L_i, $1 \le i \le h$. T_{L_i} is the subgraph of T that is induced by all vertices relevant for layer L_i, i.e., all base vertices $v \in B$ with $\lambda(v) = i$ and all compound vertices $c \in C$ with $\lambda(c^t) \le i \le \lambda(c^b)$. Note that we can contract a layer cluster tree by removing every single-child compound vertex and connecting the child directly to its grandparent. This guarantees that the size of a layer cluster tree T_{L_i} is linear to the size of L_i.

For compound graphs, there are two new restrictions which have to be considered during the crossing minimization:

Cluster-Layer Restriction: The cluster-layer restriction is satisfied, if for any compound vertex $c \in C$ of T_{L_i}, $1 \le i \le h$, the vertices lying between $u_i^{c^l}$ and $u_i^{c^r}$ in L_i are exactly the successors of c in T_{L_i}.

Cluster-Cluster Restriction: The cluster-cluster restriction is satisfied if, for any two compound vertices $c, d \in C$ which are not nested, the relative position of c and d is the same on each common layer, i.e., $pos(u_i^{c^r}) < pos(u_i^{d^l})$ for $\max(\lambda(c^t), \lambda(d^t)) \le i \le \min(\lambda(c^b), \lambda(d^b))$ or $pos(u_i^{d^r}) < pos(u_i^{c^l})$ for $\max(\lambda(c^t), \lambda(d^t)) \le i \le \min(\lambda(c^b), \lambda(d^b))$.

The crossing reduction of Forster is based on the following lemma:

Lemma 4.3 *[75] Let $G' = (L_1 \cup L_2, E' \subseteq L_1 \times L_2)$ denote a two-layered (bipartite) graph where the ordering of vertices in L_1 is fixed. An order of the vertices in L_2 has a minimum number of crossings with respect to L_1 if and only if the child order of each compound vertex c in T_{L_2} induces a minimal number of crossings.*

Thus, during a one-sided two-layer crossing minimization step, the number of crossings can be minimized without losing quality by independently computing an order of the children for each compound vertex. More precisely, for each compound vertex c of the layer cluster tree T_{L_2}, we construct a weighted two-layered *crossing reduction graph* $G'_c = (V'_c, E'_c)$ with $V'_c = L_1 \cup L'_2$. The upper layer is the same as for G', the lower layer L'_2 consists of the children of c in T_{L_2}. The relevant edges of E' are then transferred to E'_c as follows (Fig. 4.7):

- Edges $(v, w) \in E'$ ending in a vertex w that is not the successor of c in T_{L_2} are ignored.
- For each remaining edge $(v, w) \in E'$, let y denote the unique child of c, which is a predecessor of w. If E'_c does not already contain edge (v, y), we add it and set its weight to one. Otherwise, we increase the weight of (v, y) by one.

Note that for a two-layered graph G' each single edge may be transferred to $O(|C_2|)$ crossing reduction graphs where C_2 denotes the set of compound vertices of T_{L_2}. The runtime for building the crossing reduction graphs for G' is $O(|L_2| + |C_2||E'|)$. Now, for each crossing reduction graph we use a weighted one-sided two-layer crossing minimization to determine the vertex order of its lower layer. The results are used to derive an order of the vertices of L_2 that satisfies the cluster-layer restriction. Vertices $u_i^{c^l}$ ($u_i^{c^r}$) representing the left (right) border of a compound vertex $c \in C_2$ are placed accordingly. Finally, we use the approach of Barth et al. described in Section 2.4.3 to count crossings.

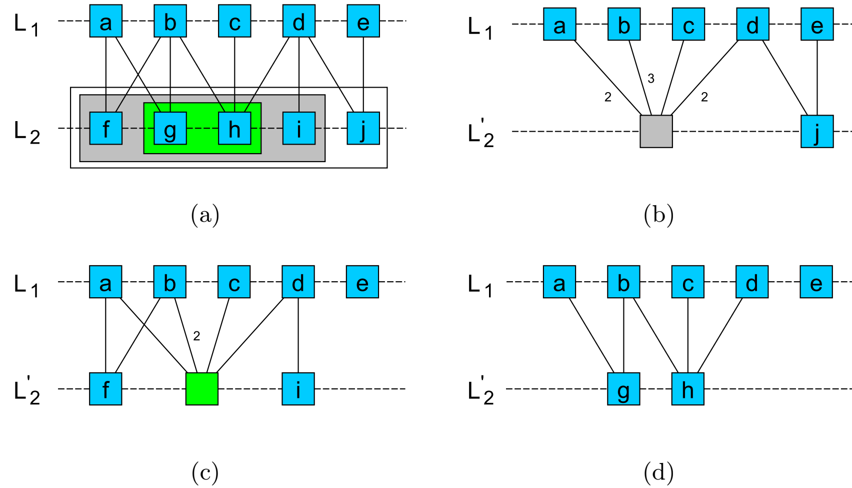


Figure 4.7: Example of creating crossing reduction graphs (taken from [74]). (a) shows the underlying two-layered graph where the colored rectangles sketch the cluster regions. Note that vertices representing the left/right border of a compound vertex are omitted here. (b),(c) and (d) show the crossing reduction graph for the white, gray and green cluster regions respectively. Edges with weight > 1 are labeled accordingly.

To satisfy the cluster-cluster restriction, we use the constrained one-sided two-layer crossing minimization described in [75, 76]. For a crossing reduction graph $G'_c = (L_1 \cup L'_2, E'_c)$ it allows us to specify constraints on the relative order for some vertex pairs of L'_2. We store these constraints in an

edge set $R \subseteq L'_2 \times L'_2$. An edge $(v, w) \in R$ denotes that v should be placed to the left of w. The given constraints are always satisfied if the graph (L'_2, R) is acyclic. We preserve the relative ordering of two compound vertices by inserting an appropriate constraint (edge) into R. Using this approach, the layer ordering of the vertices is no longer based on a linear measure and, hence, crossing middle segments may occur. For a single crossing reduction graph G'_c, the algorithm has runtime $O(|L'_2| \log |L'_2| + |E'_c| + |R|^2)$. Note that we have to apply the constrained crossing minimization to each crossing reduction graph of G'. As shown in [75], it is always sufficient to add constraints between compound vertices having the same parent in T_{L_2}. Thus, summed over all crossing reduction graphs of G', the overall number of constraints is bounded by $O(|C_2|)$. Furthermore, the overall number of edges is $O(|C_2||E'|)$ and the overall number of vertices is linear to the size of T_{L_2}. Hence, for a two-layered graph G' the clustered crossing minimization has runtime $O(|L'_2| \log |L'_2| + |C_2||E'| + |C_2|^2)$

Recall that the normalized mixed compound graph G_C may contain $\Theta(|V||E|)$ vertices and edges. More precisely, the number of layers may be $\Theta(|V|)$, the number of vertices per layer $\Theta(|V| + |E|)$ and the number of edges per layer $\Theta(|E|)$. Furthermore, we know that the number of compound vertices of a layer cluster tree is bounded by $O(|V|)$. Hence, the overall runtime complexity for applying the above crossing reduction approach to G_C is $O(|V|^2|E|)$.

Below, we show how to realize constraint PARTITION. Let $G' = (L_1 \cup L_2, E' \subseteq L_1 \times L_2)$ denote a two-layered graph and $L_2^i \subseteq L_2$ the set of vertices assigned to the i-th partition column, i.e., those vertices $v \in L_2$ with $px(v) = i$. We apply the above clustered crossing reduction approach separately to the subgraphs G'_i induced on G' by the vertices of $L_1 \cup L_2^i$, $0 \leq i < px^{\max}$. Recall that cluster regions never intersect partition cells and, thus, the cluster-layer as well as the cluster-cluster restriction are still maintained. We process the subgraphs G'_i in increasing order of i and concatenate the resulting vertex orders to obtain the order of L_2. The vertices representing the vertical grid lines are placed accordingly, e.g., vertex $u_2^{p_i^x}$ is placed between the vertices of L_2^{i-1} and L_2^i. Regarding the quality of the separate handling of the subgraphs G'_i, we can state the following theorem:

Lemma 4.4 *During the one-sided two-layer crossing minimization of G', we can separately calculate the vertex order for each subgraph G'_i without losing quality.*

Proof: Let $e_1 = (u, v)$ and $e_2 = (w, z)$ denote two edges of G' with $px(v) \neq px(z)$, i.e., both edges are assigned to different subgraphs. Furthermore, we assume w.l.o.g. that in L_1, vertex u is placed to the left of w. Since the vertex order of L_1 as well as the order of the partition columns is fixed, e_1 crosses e_2 if and only if $px(z) < px(v)$. Hence, the crossing number of

such edge pairs cannot be influenced and, thus, we do not lose quality if we separately minimize the number of crossings for each subgraph. □

Building the subgraphs of G' requires $O(|L_2| \log |L_2| + |E'|)$ overall runtime, i.e., we first sort the vertices of L_2 according to their px values and then assign the vertices and edges of G' to the different subgraphs. We only need to consider subgraphs G'_i with $|L_2^i| > 0$. The overall size of the subgraphs is linear to the size of G'. Placing the dummy vertices $u_2^{p_i^x}$ needs time $O(|L_2| + px^{\max})$.

Note that our crossing reduction approach does not depend on a specific constrained one-sided two-layer crossing minimization strategy. Due to Lemmas 4.3 and 4.4, an optimal strategy would also imply optimality with respect to constraints CLUSTER and PARTITION. However, this does not imply global optimality because the layer-by-layer sweep may introduce unnecessary bends. The overall running time of the above crossing reduction is $O(|V|^2|E|)$. If we only consider constraint PARTITION, the runtime is $O(|V||E| \log |E|)$.

4.4.1.3 Horizontal Coordinate Assignment

Let G_a denote the directed acyclic (Sugiyama-based) compaction graph resulting from the layer ordering calculated during the crossing reduction phase (see Section 2.4). Each cluster should be represented by an enclosing rectangle. Therefore, we have to vertically align the left (right) boundary vertices $u_i^{c^l}$ ($u_i^{c^r}$), $\lambda(c^t) \leq i \leq \lambda(c^b)$ for each compound vertex $c \in C$. Furthermore, we have to align the vertices $u_i^{p_j^x}$, $1 \leq i \leq h$ for each p_j^x, $0 \leq j \leq px^{\max}$ to obtain the vertical grid lines of the partition. Both are realized by mapping all vertices that should be aligned to the same vertex of G_a. Due to the adapted crossing minimization, this never introduces cycles in G_a and, hence, all aligned vertices are assigned to the same horizontal coordinate.

We use G_a to perform the horizontal coordinate assignment with the algorithm of Brandes and Köpf introduced in Section 2.4.4. Note that we no longer use a linear measure during the crossing minimization and, hence, the resulting drawing is not guaranteed to be in the linear segments model. Finally, all dummy vertices inserted during normalization are removed.

4.4.2 Construction of a Planar Embedding

In our approach, we model cluster regions and partitions using vertices and edges. More precisely, before we calculate the planar embedding, we "materialize" them in the following way: Let $x(v)$ denote the x-coordinate and $y(v)$ the y-coordinate of a vertex v. For each compound vertex $c \in C$, we know the coordinates of the vertices c^t and c^b representing the top and

bottom boundaries of the cluster. In addition, we have stored the former coordinates of the already removed dummy vertices $u_i^{c^l}$ and $u_i^{c^r}$ representing c's left and right boundaries. The cluster region of c is represented by inserting two edges e_1, e_2 between c^t and c^b into the drawing. For the route of e_1, we use point $(x(u_i^{c^l}), y(c^t))$ and $(x(u_i^{c^l}), y(c^b))$ as intermediate points and for the route of e_2, point $(x(u_i^{c^r}), y(c^t))$ and $(x(u_i^{c^r}), y(c^b))$; see Fig. 4.8(a). We materialize the partition by inserting the partition grid graph P_G into the drawing. The position of the vertices of P_G is derived from the former coordinates of the dummy vertices $u_i^{p_j^x}$ and p_i^y; see Fig. 4.8(b). Note that the overall number of additional elements inserted into the drawing is linear in the number of clusters and partition cells.

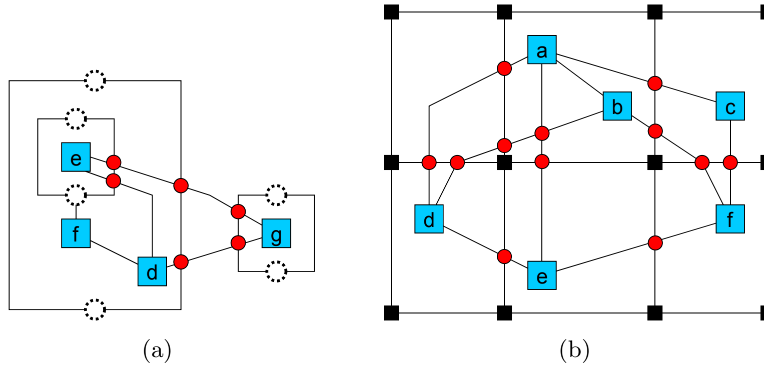


Figure 4.8: Materializing cluster regions (a) for the example shown in Fig. 4.5. Small red circles denote dummy vertices which represent crossings. They are inserted during the construction of the planar embedding. Partitions are materialized by inserting the partition grid graph as shown in (b). Small rectangular vertices denote the vertices of the partition grid graph. Fig. 4.6 shows the underlying graph.

Now we construct the planar embedding induced by the modified drawing as described in Section 4.2.2. For clustered graphs the runtime of this step increases to $O(|V||E|\log|E| + x\log x)$ since the underlying drawing is not in the linear segments model and, thus, may contain up to $O(|V||E|)$ bends. When we only consider constraint PARTITION the runtime is $O((|E| + |P|)\log(|E| + |P|) + x\log x)$, where $|P| = py^{\max} \cdot px^{\max}$ denotes the number of partition cells. For the overall number of crossings x we have $x = x_E + x_C + x_P$ where x_E denotes the crossing number of edges of E, x_C the crossing number of edges of E with edges modeling cluster regions and x_P the crossing number of edges of E with edges of the partition grid graph. Due to the monotonic edge routes produced by Sugiyama's approach, an edge of E crosses at most $py^{\max} + px^{\max}$ partition cells. Hence, we have $x_P = O(|E| \cdot (py^{\max} + px^{\max}))$. Moreover, an edge of E crosses at most $O(|C|)$ clusters, with the result that $x_C = O(|E||C|)$.

The orthogonalization phase demands a connected planarized input graph. Recall that the induced subgraph G^* of G on the base vertices B is connected. Hence, if there are vertices of B both inside and outside of a cluster region/partition cell, there is at least one crossing vertex that connects an edge of G^* with one representing the boundary of that cluster region/partition cell. Thus, the graph $G^* \cup P_G$ is always connected since using partitions with only one non-empty partition cell is useless. If we do not use partitions, there could be a compound vertex $c \in C$ containing all base vertices. If there is such a compound vertex c, we simply add an edge from c to one of its children in T into G, which then guarantees that G is connected.

Recall the orientation problem described in Section 4.1. The shape assignment during the orthogonalization phase guarantees correct orientation of the partition grid graph if there is at least one upward edge $(v, w) \in E_\uparrow$ that crosses the boundary of a partition cell, i.e., $p(v) \neq p(w)$. Hence, we add a preprocessing step that checks this condition and, if necessary, adds such an edge to $E_\uparrow$.

4.4.3 Rerouting of Edges

To obtain a c-planar embedding of G_C, we have to modify the rerouting step such that the calculated edge routes guarantee that each edge crosses the boundary of a region at most once. Note that up to now the embedding may have contained edges that do not satisfy this property. Let E_x denote the set of such edges. Since our rerouting approach cannot guarantee that edges are inserted upward planar, we only reroute edges $e \in E_x \cap E_\uparrow$ if requirement CLUSTER is more important than FLOW. Furthermore, we never reroute skeleton edges.

A modification of the dual graph routing approach that guarantees that each edge crosses a region boundary at most once is described in [43]. Let G' denote the planarized graph. First, a common dual graph $D_{G'}$ of G' is computed. As illustrated in Fig. 4.9, each vertex of $D_{G'}$ is enclosed by a certain set of boundary rectangles of clusters. Let c denote the innermost cluster that encloses a vertex u_f of $D_{G'}$. Then each vertex on the path $r \rightarrow_T^* c$ also encloses u_f (r denotes the root of the inclusion tree T). When we route an edge $e = (v, w) \in E$, we first calculate the undirected path $v \rightarrow_T^* w = v, c_1, \ldots, c_k, w$. Each edge (u_f, u_g) of $D_{G'}$ is handled as follows: Let c_i denote the innermost cluster that encloses u_f and c_j the innermost cluster that encloses u_g. If c_i or c_j is not contained in $\{c_1, \ldots, c_k\}$, (u_f, u_g) is temporarily removed. Otherwise, if $i < j$ we orient (u_f, u_g) from u_f to u_g (the edge can only be passed from u_f to u_g) and if $i > j$ we orient it from u_g to u_f. If $i = j$, then edge (u_f, u_g) is bidirectional, i.e., it can be traversed in both directions. Now, as in the common dual graph routing approach, we compute a directed shortest path between vertex v' and w'. Edge e is

inserted by following the shortest path and replacing crossings with dummy vertices. The old route of e is discarded. After each step, the dual graph has to be updated and the temporarily removed and oriented edges are restored. The running time of this step is $O((|V| + x_E + x_C + x_P)|E|)$ [43].

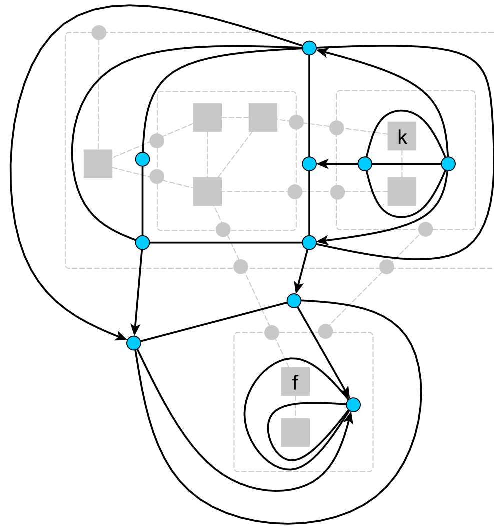


Figure 4.9: The modified dual graph for the example shown in Fig. 3.3(c) when we route edge (k, f). Edges without arrows are bidirectional. The embedding of the underlying graph is shown in the background (gray vertices and dashed edges).

To satisfy constraint PARTITION, the route of an edge is not allowed to leave the outer boundary of the partition grid graph. Furthermore, in practice we obtain more readable drawings if we use the following restrictions on edge routes:

- Each route of an edge $(v, w) \in E$ is completely contained inside the smallest rectangle containing partition cell $p(v)$ and $p(w)$.
- No edge reenters a partition cell after leaving it.

Both restrictions lead to fewer crossings between edges of E and edges of the partition grid graph. Since the shape of the partition grid graph is fixed, such crossings often lead to unsatisfying edge routes of the crossing edges, i.e., to lurching edge routes with a lot of bends. Using these restrictions, an edge route never crosses a partition cell if $p(v) = p(w)$. For an edge $e = (v, w) \in E$, let R^e denote the smallest rectangle containing partition cell $p(v)$ and $p(w)$. In order to include the first restriction, we remove each vertex of $D_{G'}$ that represents a face lying outside of R^e. Let *dist* denote the function that returns the Manhattan distance between two partition cells p_1

and p_2, i.e., $dist(p_1, p_2) = |p_1^x - p_2^x| + |p_1^y - p_2^y|$. Furthermore, for a vertex u_f of $D_{G'}$ let p' denote the function that maps u_f to its enclosing partition cell. We incorporate the second restriction by orienting edges $e' = (u_f, u_g)$ of $D_{G'}$ towards the partition cell of e's target, as shown in Fig. 4.10. More precisely, if $dist(p'(u_f), p(w)) < dist(p'(u_g), p(w))$ we orient e' from u_g to u_f and if $dist(p'(u_f), p(w)) > dist(p'(u_g), p(w))$ we orient it from u_f to u_g. If $p'(u_f) = p'(u_g)$, then e' is bidirectional. Note that this is sufficient to meet the second restriction. The time needed for rerouting a single edge is linear in the size of the planarized graph and, hence, the overall runtime of the rerouting step is $O((|V| + x_E + x_C + x_P)|E|)$. Since cluster regions are not allowed to cross partition cells, we can easily combine both approaches described in this subsection.

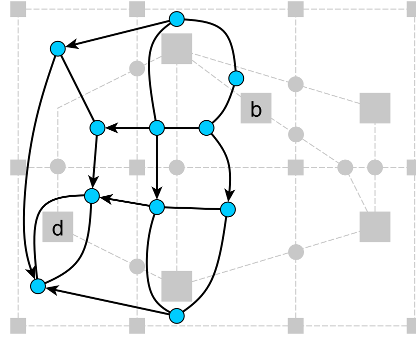


Figure 4.10: The modified dual graph for the example shown in Fig. 4.8 when we route edge (b, d). Edges without arrows can be traversed in both directions. The embedding of the underlying graph is shown in the background (gray vertices and dashed edges).

After rerouting edges, we replace the vertices $c^{\{t,b\}}$ representing the top and bottom borders of a cluster $c \in C$ as follows: if $c^{\{t,b\}}$ is only incident to the two edges representing the border of c, we remove $c^{\{t,b\}}$ and join these edges. Otherwise, as shown in Fig. 4.11, we replace $c^{\{t,b\}}$ by a chain of vertices each connected to a non-border edge incident to $c^{\{t,b\}}$. This step can be done in time $O(|C| + |E|)$.

4.4.4 Optimal Swimlane Order

When partitions are only one-dimensional, i.e., traditional swimlanes, and the order of the lanes is not prescribed, we can use the following strategy to improve the readability of the resulting drawing: since each vertex is constrained to stay in its swimlane, the number of crossings as well as the total edge length heavily depend on the order of the swimlanes. If there are many edges between two swimlanes it is advantageous to place them near

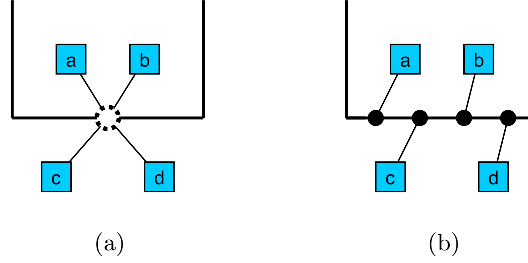


Figure 4.11: Splitting a vertex representing the bottom border of a cluster into several vertices, one for each incident edge.

each other. Hence, we construct an undirected graph $G_P = (V_P, E_P)$ as follows (w.l.o.g. we assume a vertical alignment of partitions): V_P contains a vertex p_i, $0 \leq i < px^{max}$ for each partition column (swimlane) and E_P an edge (p_k, p_l) if there is at least one edge $e = (u, v) \in E$ with $px(u) = k \wedge px(v) = l$ or $px(u) = l \wedge px(v) = k$. The weight of an edge $e \in E_P$ is equal to the number of edges in E that meet this condition. Now we can formulate the corresponding optimization problem: Find a permutation π of the vertices such that

$$z = \sum_{e=(v,w)\in E_P} weight(e) \cdot |\pi(v) - \pi(w)|$$

is minimized ($\pi(v)$ denotes the position of v in π). For uniform edge weights this problem is known as the *Optimal Linear Arrangement Problem* [85] and is NP-hard.

We tackle this optimization problem with a variant of the greedy switch heuristic [56]. It works as follows: Starting with some order π of the vertices of V_P, consecutive pairs of vertices are exchanged if this reduces the value of the objective function z. We repeat this step with the resulting order until there is an iteration without an improvement of z. Algorithm 3 gives the corresponding pseudo-code. Note that the maximum number of iterations is bounded by $|V_P|$. A single iteration has runtime $O(|V_P| + |E_P|)$ since we have to check $|V_P| - 1$ vertex pairs and updating z for a vertex pair v_i, v_{i-1} needs $O(\delta_{G_P}(v_i) + \delta_{G_P}(v_{i+1}))$ time. Hence, the overall runtime is $O(|V_P|^2 + |V_P||E_P|)$.

The heuristic performs a lot of local changes, but it does not recompute the complete ordering. Hence, to further improve the results, we apply it to a certain number of randomized initial vertex orders of V_P and take the result of the best iteration. Finally, we arrange the partitions according to the resulting order. Note that for two-dimensional partitions we can simply apply the same heuristic for each dimension separately.

Algorithm 3: `greedySwitchHeuristic`

$changed \leftarrow true$;
while $changed$ **do**
    $changed \leftarrow false$;
    for $i \leftarrow 1$ **to** $|V_P| - 1$ **do**
        if *exchange of* v_i *and* v_{i+1} *reduces* z **then**
            switch position of v_i and v_{i+1};
            $changed \leftarrow true$;

4.5 Including Port and Side Constraints

In this section, we show how to include port/side constraints into our planarization framework. We first give the modifications needed to realize port/side constraints within Sugiyama's approach followed by some notes about the remaining steps.

4.5.1 Construction of an Initial Drawing

An approach for incorporating port constraints into Sugiyama's approach was sketched in [131]. It models ports as dummy vertices which are placed inside the layer above or below the corresponding vertex (depending on the side associated with the port constraint). The edge ordering given by the port constraints is realized by inserting appropriate constraints between these dummy vertices and then applying a constraint crossing minimization approach. Note that this approach is not sufficient for our port/side constraint model.

Let $G = (V, E)$ denote a simple input graph. Recall that in Sugiyama's approach each edge is either attached to the top or bottom of its endpoints. Hence, for each vertex $v \in V$ we first map all pins of the left and right sides to the top and bottom. Therefore, we use an imaginary horizontal line that crosses v below the κ-th pin of its left/right side (see the dashed red line in Fig. 4.12(a)). As shown in Fig. 4.12(b), we increase the width of v and assign all pins above this separation line to the top and all pins below that line to the bottom. Let P_v^t denote the ordered list of pins mapped to the top and P_v^b the order list of pins mapped to the bottom of v. The pins of P_v^t are ordered as follows (from left to right): first we have pin $\kappa, \ldots, 2\kappa - 1$ of the left side followed by pin $1, \ldots, 2\kappa - 1$ of the top and finally pin $1, \ldots, \kappa$ of the right side. Pins of P_v^b are ordered as follows: first we have pin $\kappa - 1, \ldots, 1$ of the left side followed by pin $2\kappa - 1, \ldots, 1$ of the bottom and finally pin $2\kappa - 1, \ldots, \kappa + 1$ of the right side.

Below we show the required modifications for the layering and crossing minimization phase. Due to the above mapping of the pins to the top and

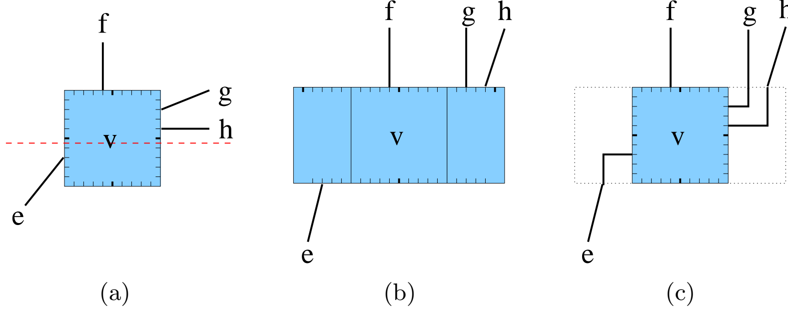


Figure 4.12: (a) shows a vertex v with some incident edges assigned to pins of v. The dashed red line denotes the separation line that determines which of the pins are mapped to the top or bottom. (b) illustrates the result after mapping pins to these sides. To obtain valid routes for edges originally assigned to pins of the left/right side, we extend the routes as shown in (c).

bottom of the vertices, our layering strategy is similar to that for constraint BIMODAL. Note that in the following we consider edges to be undirected, i.e., $(v, w) = (w, v)$.

4.5.1.1 Layering

For each vertex $v \in V$, let $E_v^t \subseteq E^c$ denote the set of edges which have to be connected to a pin of P_v^t, i.e., those edges e with side constraint sc_v^e, $side(sc_v^e) = t$ or those with port constraint pc_v^e with $pin(pc_v^e) \in P_v^t$. Analogously, we define E_v^b to be the set of edges which have to be connected to a pin of P_v^b (edges e with side constraint sc_v^e, $side(sc_v^e) = b$ or those with port constraint pc_v^e and $pin(pc_v^e) \in P_v^b$).

For a given layer assignment λ, we proceed as follows: for each edge $(v, w) \in E_v^t$ of a vertex v with $\lambda(v) < \lambda(w)$, we insert a dummy vertex d into layer $\lambda(v) - 1$ and replace (v, w) by two edges (v, d) and (w, d). Analogously, for each edge $(w, v) \in E_w^b$, we insert a dummy vertex d' into layer $\lambda(w) + 1$ and replace (v, w) (or (w, d)) by two edges (d', w) and (d', v) (or (d', d)). This construction corresponds to that for back edges shown in Fig. 4.4. Note that we have to transfer the port/side constraints of (v, w) to the resulting dummy edges.

We try to reduce the number of inserted dummy vertices by adding appropriate edges into the layering constraint graph $C_G = (V, E_C)$. For each edge $e = (v, w) \in E^c$, we add a directed edge $e' = (v, w)$ into E_C if $e \in E_w^t$ or $e \in E_v^b$. If $e \in E_w^b$ or $e \in E_v^t$, we add a directed edge $e' = (w, v)$ into E_C. For each edge e' added to E_C, we set $etype(e') = \texttt{pc/sc}$. We extend the weight function ω_c as follows: For each edge $e = (v, w) \in E_C$,

$$\omega_c(e) = \begin{cases} 2|E_C| & \text{if } e \in E'_C, \\ 2 & \text{if } etype(e) = \texttt{bimodal} \ \wedge \ v, w \in V^b, \\ 2 & \text{if } etype(e) = \texttt{pc/sc} \ \wedge \ e \in E_v^t \ \wedge \ e \in E_v^b, \\ 1 & \text{if } etype(e) = \texttt{bimodal} \ \wedge \ (v \in V^b \ \veebar \ w \in V^b), \\ 1 & \text{if } etype(e) = \texttt{pc/sc} \ \wedge \ (e \in E_v^t \ \veebar \ e \in E_v^b). \end{cases}$$

There are no further changes for the layer assignment and normalization. The removal of the dummy vertices inserted during the above transformation is done as for constraint BIMODAL.

4.5.1.2 Crossing Minimization

The following crossing minimization approach only supports port constraints. Hence, before we apply it, we temporarily transform each side constraint sc_v^e of an edge e into a port constraint pc_v^e. Therefore, we assign e to a non-occupied pin on side $side(sc_v^e)$ of v. For side constraints on the left and right side we choose a pin that conforms to the given layering, i.e., for an edge (v, w) with $\lambda(v) < \lambda(w)$, the pin has to be in P_v^b and if $\lambda(v) > \lambda(w)$ it has to be in P_v^t. Obviously, a port constraint preserving drawing of the transformed graph is also a port constraint preserving drawing of the original graph. After the crossing minimization we restore the original constraints.

In the following, we consider the one-sided two-layered crossing minimization for a two-layered graph $G' = (L_1 \cup L_2, E' \subseteq L_1 \times L_2)$ with port constraints.

Port Constraints on Vertices of L_2 First, we show how to handle port constraints of edges on vertices of L_2. For each vertex $v \in L_2$, let $pos(v)$ denote the layer position of v in L_2 and $E_v \subseteq E'$ the edges incident to v. While up to now there have never been crossings between edges of E_v, this is no longer true when we include port constraints since such constraints restrict the valid orderings of edges around vertices. An ordering of the edges E_v around v is called *valid* if it satisfies the items of Definition 3.7 (page 46). Since the positions of vertices of L_1 are fixed, we have the following property:

Property 4.5 *During a one-sided two-layered crossing minimization, the number of crossings between edges of E_v, $v \in L_2$ does not depend on the position $pos(v)$ of v.*

It follows that we cannot affect the crossing number between these edges. Since common cross counting approaches do not include port constraints, we have to additionally count and add the number of such crossings.

Let $E_v^{pc} \subseteq E_v$ denote the edges with port constraint on v and $E_v^{free} \subseteq E_v$ the edges without constraints. Then, there are three different kinds of crossings between edges of E_v: crossings between two edges with port constraints

on v (called `pc-pc` crossings), crossings between two edges without constraints on v (called `free-free` crossings) as well as crossings between an edge without constraint and one with port constraint on v (called `pc-free` crossings). Below, we show how to find a valid ordering of the edges E_v on a vertex $v \in L_2$ such that the number of crossings between these edges is minimized.

For edges of E_v^{pc}, the relative positions of the endpoints are already fixed. Hence, the number of `pc-pc` crossings is also fixed and can be calculated in time $O(|E_v^{pc}|^2)$.

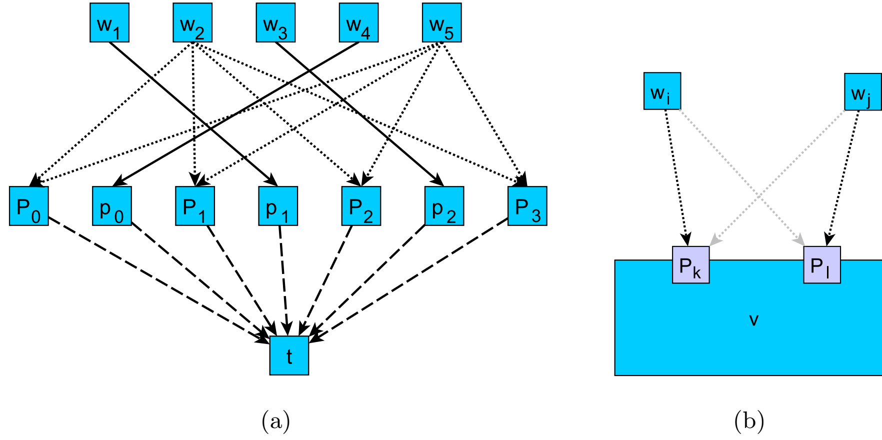
In the following, we present a network flow problem which calculates a valid ordering of the edges of E_v on v such that the number of `pc-free` crossings is minimized. Let $\mathcal{N}_v = (U_1 \cup U_2 \cup t, A)$ denote the network used for minimizing crossings between edges incident to a vertex $v \in L_2$. Vertex set U_1 contains the neighbors of v in layer L_1, i.e., $U_1 = \{w \mid w \in L_1 \wedge w \text{ adjacent to } v\}$. Each vertex $w \in U_1$ has supply $b(w) = 1$. Vertex set U_2 consists of vertices $P_0, p_0, P_1, p_1, \ldots, P_{k-1}, p_{k-1}, P_k$, $k = |E_v^{pc}|$ where vertex p_i represents the i-th pin of P_v^t that is associated with an edge of E_v^{pc}. Vertex P_i represents the set of pins lying between pin p_{i-1} and p_i (not including p_{i-1} and p_i). Vertices of U_2 have zero supply. Vertex t has supply $b(t) = -|U_1|$. The edge set A consists of the following subsets:

- An edge set A_p that contains an edge (w, p_i) for each edge $e = (w, v) \in E_v^{pc}$, $w \in L_1$ with port constraint pc_e^v associated with the pin represented by vertex p_i (i.e., p_i represents $pin(pc_e^v)$). Edges of A_p have zero costs and capacity one.
- An edge set A_P that contains, for each edge $e' = (w, v) \in E_v^{free}$, $w \in L_1$, an edge (w, P_i) $\forall\, 0 \le i \le k$, i.e., w is connected to each vertex P_i. The capacity of these edges is one and the costs are set to the number of edges of E_v^{pc} that are crossed by e' if e' is assigned to a pin represented by vertex P_i.

  For an edge $e \in A_P$, we can determine the corresponding number of crossings with edges of E_v^{pc} in time $O(|E_v^{pc}|)$ (by simply comparing the relative positions of the endpoints). Hence, the overall time for calculating the costs for a network $\mathcal{N}_v$ is $O(|E_v^{pc}||A_P|) = O(\delta_G(v)^3)$.
- An edge set A_t that contains an edge (u, t) for each vertex $u \in U_2$. Let $|P_i|$ denote the number of pins represented by a vertex $P_i \in U_2$. The cost of an edge $(u, t) \in A_t$ is zero and its capacity is $|P_i|$ if u represents a vertex P_i and one if it represents a vertex p_i.

Fig. 4.13(a) shows an example of a network N_v and illustrates its geometric interpretation. Our transformation of the crossing minimization problem into a network flow problem has the following intuitive interpretation: Due to the chosen supplies, each neighbor $w \in U_1$ of v has exactly one incident

edge $e = (w, u) \in A$ with $f(e) > 0$. The flow on this edge is interpreted as an assignment of edge $(w, v) \in E$ to a pin represented by $u \in U_2$. For an edge $e \in A_P$, the cost of this flow corresponds to the number of crossings of e with edges of E_v^{pc}. The capacity of edges (P_i, t) guarantees that the number of edges assigned to pins represented by vertex P_i is valid, i.e., there are enough free pins for these edges. Since the number of available pins is larger than the number of neighbors of v, there is always a feasible flow in $\mathcal{N}_v$. We can derive a valid ordering of the edges of E_v by sorting them according to the position of the associated pin in P_v^t. Edges assigned to pins represented by the same vertex P_i can be ordered arbitrarily. Using the above observations we can directly derive the following lemma:



(a) (b)

Figure 4.13: (a) illustrates the minimum cost flow network $\mathcal{N}_v$ for a vertex v. Edges of A_p are denoted by solid lines, edges of A_P by dotted lines and edges of A_t by dashed lines. To demonstrate the geometric interpretation of the construction of $\mathcal{N}_v$, the neighbors w_i of v are ordered from left to right according to their position in L_1. Furthermore, edges (w_1, v), (w_3, v), $(w_4, v) \in E'$ have port constraints on v. Now if, for example, edge $e = (w_5, v)$ is assigned to a pin represented by vertex P_0 ($f(w_5, P_0) = 1$) this would induce a cost of 3 in $\mathcal{N}_v$ which corresponds to the number of crossings of e with edges of E_v^{pc}. (b) demonstrates how to resolve `free-free` crossings. The crossing between edges (w_i, v) and (w_j, v) (gray-colored) can be removed by changing the pins of both edges.

Lemma 4.6 *A minimum cost flow in the network $\mathcal{N}_v$ induces a valid ordering of the edges of E_v on v with a minimum number of* `pc-free` *crossings.*

We store the induced edge order in an ordered list L^{E_v}. Let $x_{pc-free}$ denote the corresponding number of `pc-free` crossings, x_{pc-pc} the corresponding number of `pc-pc` crossings and $x_{free-free}$ the corresponding number of `free-free` crossings. Since x_{pc-pc} is fixed and due to Lemma 4.6,

$x_{pc-free} + x_{pc-pc}$ is a lower bound on the number of crossings between edges of E_v.

Lemma 4.7 *There is always a valid ordering of the edges of E_v on v such that the number of crossings between these edges is $x = x_{pc-free} + x_{pc-pc}$, i.e., $x_{free-free} = 0$. This number is minimal.*

Proof: The minimum cost flow induces a valid ordering of the edges of E_v on v. If, for this ordering, $x_{free-free} = 0$, we have $x = x_{pc-free} + x_{pc-pc}$. Assume that $x_{free-free} > 0$. Then there are crossing edges $e = (w_i, v)$ and $e' = (w_j, v)$, both without port constraint on v. As illustrated in Fig. 4.13(b), we can remove such a crossing by exchanging the pins of both edges. Obviously, neither $x_{pc-free}$ nor x_{pc-pc} can be increased by this transformation (each edge that crosses (w_i, P_k) or (w_j, P_l) also has to cross edge (w_i, P_l) or (w_j, P_k)). Furthermore, this transformation does not change the number of edges assigned to pins represented by a vertex P_i. Hence, after successively removing all `free-free` crossings, we obtain a valid ordering of the edges that induces $x \leq x_{pc-free} + x_{pc-pc}$ crossings. Thus, the resulting edge order induces a minimum number of crossings between edges of E_v. □

We know that a minimum cost flow in $\mathcal{N}_v$ induces an edge order that minimizes `pc-free` crossings and that this order can be transformed into an optimal edge order without `free-free` crossings. Hence, after calculating the minimum cost flow f in $\mathcal{N}_v$, we can use Algorithm 4 to directly construct an optimal edge order in time $O(|E_v|)$. The function *pop*(L) (line 6) removes the first element of a list L and returns it. The function *append*(L, e) (lines 7 and 10) adds an edge e to the end of a list L.

For each vertex v the network $\mathcal{N}_v = (U, A)$ can be constructed in time $O(\delta_G(v)^3)$. For a network $\mathcal{N}_v = (U, A)$, the wave implementation of the cost scaling minimum cost flow algorithm calculates a minimum cost flow in time $O(|U|^3 \log(|U| \cdot C))$ [1, 88], where C denotes the maximum absolute value for costs. This results in a runtime of $O(\delta_G(v)^3 \log \delta_G(v)^2)$ for each vertex $v \in V$.

Port Constraints on Vertices of L_1 In the following, we show how to handle port constraints of edges on vertices of L_1. Recall that the vertex order of L_1 is fixed. For a vertex $v \in L_1$ let $A_v \subseteq L_2$ denote the set of neighbors of v in L_2. To reduce the number of crossings between edges of E_v, we refine the measure calculation of the vertices in L_2 as follows: We calculate the measure value for each vertex of L_2, using a linear measure m. For all pairs of neighbors $n_1, n_2 \in A_v$, $v \in V$ with $m(n_1) = m(n_2)$ we use a second sorting criteria which guarantees an order of the edges (v, n_1), (v, n_2) on v that complies with the given port constraints. Therefore, we simply calculate a valid ordering of the edges E_v and use the edges' index

Algorithm 4: `calcEdgeOrder`

Input: A vertex v, the edge set $E_v = E_v^{pc} \cup E_v^{free}$ as well as the network $\mathcal{N}_v$ with flow function f.
Output: A valid edge order L^{E_v} (ordered list) that induces a minimum number of crossings between the edges of E_v.

```
1  L^{E_v} ← ∅;
2  L^{free} ← edges (w,v) ∈ E_v^{free} sorted according to pos(w) in increasing
   order;
3  for i ← 0 to |E_v^{pc}| do
4      a ← edge (P_i, t) of A_P;
5      for j ← 0 to f(a) do
6          e ← pop(L^{free});
7          append(L^{E_v}, e);
8      if i < |E_v^{pc}| then
9          e' ← edge (w,v) of E_v^{pc} ∧ p_i represents pin(pc_v^{(w,v)});
10         append(L^{E_v}, e');
```

in this ordering as the sorting criteria. Note that such an ordering can be found in time $O(\delta_G(v) \log \delta_G(v))$ by first sorting the edges of E_v^{pc} and then distributing the remaining edges.

Our modified measure calculation does not completely prevent crossings between edges of E_v. However, due to our layering strategy (only one non-dummy vertex per layer), there are always a lot of neighbors $n_1, n_2 \in A_v$ with equal measure values, i.e., if both neighbors are dummy vertices inserted during the normalization. When we include constraints PARTITION and CLUSTER, there might be additional crossings between edges of E_v because the layer ordering of vertices is influenced by these constraints. Hence, after fixing the order of the vertices of L_2 we apply the same strategy as described above for counting the number of crossings between edges of E_v for each $v \in L_1$.

Runtime Our crossing minimization performs a layer-by-layer sweep and applies the one-sided two-layered crossing minimization to each pair of adjacent layers. For each non-dummy vertex v, we have to construct the network $\mathcal{N}_v$ to calculate a valid ordering of the edges E_v on v and to count the corresponding number of crossings. The number of vertices with port constraints as well as the maximum degree of a single vertex is bounded by $|V|$. Thus, the overall runtime is $\sum_{v \in V} O(\delta_G(v)^3 \log \delta_G(v)^2) = \sum_{v \in V} c \cdot \delta_G(v)^3 \log \delta_G(v)^2 \leq \sum_{v \in V} \delta_G(v) \cdot c \cdot |V|^2 \log |V|^2 = O(|V|^2 |E| \log |V|)$ with

c being a suitable constant. The horizontal coordinate assignment uses the calculated edge orders for assigning edges to pins.

4.5.2 Remaining Steps

After performing the horizonal coordinate assignment we use the resulting drawing to obtain a port constraint preserving embedding. Note that all edges are either attached to the top or bottom of vertices. Hence, if the above approach is used solely for producing layered drawings, we have to adjust the route of edges attached to the left/right sides accordingly. This can be done as described in [131] and shown in Fig. 4.12(c) by using orthogonal routes with one bend to connect those edges with the corresponding pin on the left/right side.

Unlike our crossing minimization, the rerouting can also cope with side constraints. During the rerouting of edges we have to ensure that the resulting edge order around the vertices is still port constraint preserving. Hence, before we reroute an edge $e = (v, w)$, we first traverse the cyclic edge order around v and w to determine the valid positions of e in both orders. Let D_G denote the dual graph used for routing e and v' and w' the vertices of D_G that represent v and w. We temporarily remove those edges of D_G which are incident to v' (w') and which might lead to invalid edge orders around v (w). Since, for a single edge (v, w), this can be done in time $O(\delta_G(v) + \delta_G(w))$, the overall runtime of the rerouting step remains $O((|V|+x)|E|)$. Thus, the overall runtime of the above planarization approach with respect to the `mc` scenario is $O(|V|^2|E| \log |V| + x|E|)$.

To combine the approach described in this section with that for constraints UPWARD and BIMODAL, we have to assign temporary side constraints to the edges of $E_\uparrow$ and E^b. More precisely, for each edge $e = (v, w) \in E_\uparrow \cup E^b$, we insert two side constraints sc_e^v and sc_e^w with $side(sc_e^v) = t$ and $side(sc_e^w) = b$ into SC_G. This guarantees a correct ordering of edges around vertices and thus a valid embedding for those constraints.

The runtime of our planarization framework for the different constraints depends heavily on the runtime of Sugiyama's algorithm. In the next section, we present a significant improvement of the runtime and space complexity of Sugiyama's approach which guarantees a satisfying runtime for our planarization. Note that the described improvement does not depend on our planarization approach and, thus, can also be used just to produce layered drawings of graphs.

4.6 Fast Implementation of Sugiyama's Approach

As already stated in Section 2.4, the time complexity of algorithms implementing Sugiyama's framework depends heavily on the number of dummy

vertices inserted during the normalization. Let $G = (V, E)$ denote the directed input graph. Using the fastest available algorithms for each phase, the runtime is $O(|V||E| \log |E|)$ with $O(|V||E|)$ space requirement.

In the following, we present a fast implementation of Sugiyama's approach which produces drawings in the linear segments model (introduced in Section 2.4.4) and reduces the above time and space complexity. The approach avoids introducing dummy vertices for each layer spanned by an edge. Instead, it splits edges only in a limited number of segments (at most three). As a result, there may be edges which traverse layers without having a dummy vertex in them. We will extend the existing crossing minimization and coordinate assignment algorithms to handle this case. Our algorithm is able to keep the number of dummy vertices and edges linear in the size of the graph without increasing the number of crossings. Thus, it reduces the worst-case time complexity to $O((|V|+|E|) \log |E|)$ and requires $O(|V|+|E|)$ space.

A similar idea is used in the Tulip system described in [5]. Unfortunately, no details about the theoretical or practical performance, or the implementation are given, and a comparison with the quality of the approaches commonly used has not been described. However, in their approach, only the proper edges are considered in the crossing reduction phase and the long edges are ignored. This leads to drawings which have many more crossings than those using the traditional approach. In contrast, we will show that our technique yields the same quality as the methods traditionally used in practice.

4.6.1 Basic Idea

Since in the linear segments model each edge consists of at most two bends, all corresponding dummy vertices in the middle layers have the same x-coordinate. We combine them into one *middle segment* which reduces the size of the normalized graph G_N noticeably. More precisely, if edge $e = (v, w)$ spans between layers L_i and L_j with $|j - i| > 2$, we introduce only two dummy vertices: p_e at layer L_{i+1} (called p-vertex) and q_e at layer L_{j-1} (called q-vertex), as well as three edges: (v, p_e), $s_e = (p_e, q_e)$, and (q_e, w). The first and the last edge are proper while the vertical edge s_e, called the *middle segment* of e, is not necessarily proper (Fig. 4.14(a)). If $|j - i| = 2$, we insert a single dummy vertex r_e at layer L_{i+1} as well as two edges, (v, r_e) and (r_e, w) (Fig. 4.14(b)). Single dummy vertices are treated like common vertices later on. We call this transformation *sparse normalization* and call the result the *sparse normalized graph* $G_S = (V_S, E_S)$ (Fig. 4.15(a)). The size of the sparse normalized graph is linear with respect to the size of the input graph. A similar transformation is used in the horizontal coordinate assignment approach of Brandes and Köpf [27], where vertically aligned vertices are combined into blocks.

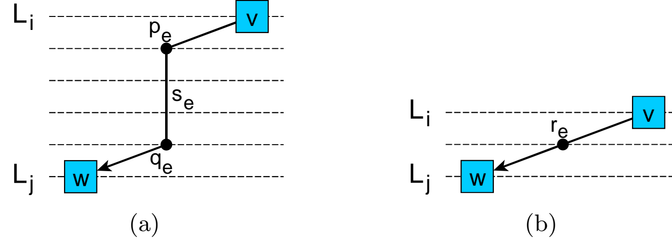


Figure 4.14: Sparse normalization: In (a), edge $e = (v, w)$ ($span(e) > 2$) is split into three segments: (v, p_e), $s_e = (p_e, q_e)$ and (q_e, w). The first and the last edge are proper (each of *span* 1) and s_e is drawn vertical. In (b), edge e ($span(e) = 2$) is split into two segments: (v, r_e) and (r_e, w).

A layer L of a sparse normalized graph contains vertices and middle segments. A layer ordering of a sparse normalized graph is a linear ordering of the vertices and middle segments in a layer and is called a *sparse layer ordering*. When we draw a graph G in the linear segments model, there is a correlation between layer orderings of the normalized graph G_N and sparse layer orderings of the sparse normalized graph G_S.

Let us look at the layer orderings of normalized graphs: instead of storing the layer ordering in lists, we can store it in a directed graph D. This graph has an edge between vertices v and w if and only if both vertices are in the same layer L_i and are consecutive. The ordering $<^o$ defined as $v <^o w$ if and only if there is a directed path from v to w in D, is a complete ordering for the vertices of a layer, i.e., either $v <^o w$ or $w <^o v$ for $v, w \in L_i$. In fact, D is the compaction graph G_a mentioned in Section 2.4.4. The graph D has $|V_N|$ vertices and $O(|V_N|)$ edges, which results in a worst-case size of $O(|V||E|)$.

We want to reduce the size of D to $O(|V|+|E|)$ without losing the property that $<^o$ defines a total layer ordering. The key observation therefore is that edges between two consecutive middle segments of L_i can be omitted if there are no crossing middle segments.

Given a layer L_i of a sparse normalized graph, we partition the layer in the following way:

$$S_{i_0}, v_{i_0}, S_{i_1}, v_{i_1}, S_{i_2}, v_{i_2}, \ldots, S_{i_{n_{i-1}}}, v_{i_{n_{i-1}}}, S_{i_{n_i}}.$$

The list S_{i_k} contains the middle segments lying between vertices $v_{i_{k-1}}$ and v_{i_k} for $1 \leq k \leq n_i - 1$, S_{i_0} contains the middle segments preceding v_{i_0} and $S_{i_{n_i}}$ the middle segments succeeding $v_{i_{n_{i-1}}}$. We denote the first element of a non-empty list S_{i_k} as $head(S_{i_k})$ and the last element as $tail(S_{i_k})$. Furthermore, we denote by $s(v)$ the middle segment to which $v \in V_S$ is incident if v is a p- or q-vertex, otherwise $s(v) = v$.

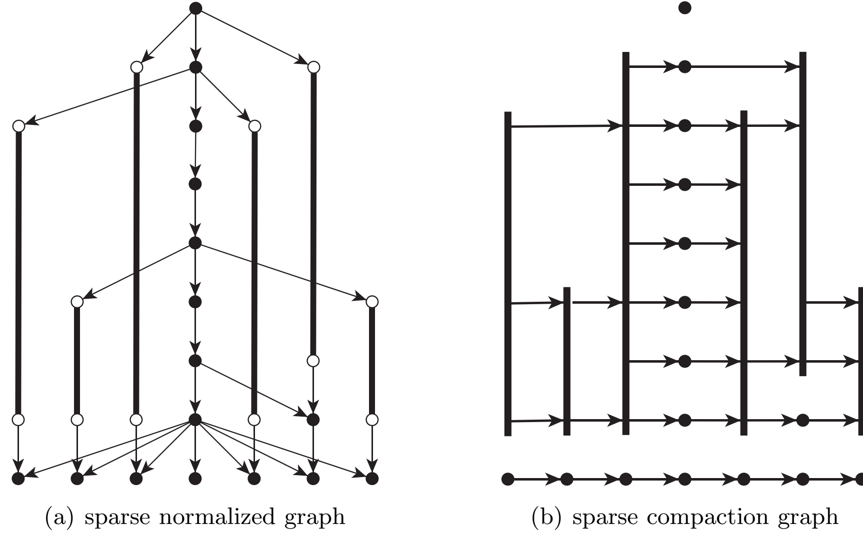


Figure 4.15: (a) shows a sparse normalized graph where thick lines denote the middle segments. (b) shows the corresponding sparse compaction graph.

Definition 4.8 *Given a directed acyclic graph $G = (V, E)$ and a sparse layer ordering in which no two middle segments cross, the sparse compaction graph (N, A) of the sparse normalized graph $G_S = (V_S, E_S)$ of G is defined as:*

$$
\begin{aligned}
N &= V \cup \{r_e : r_e \text{ single dummy vertex of } e \in E\} \cup \\
&\quad \{s_e : s_e \text{ middle segment of } e \in E\} \\
A &= \{(s(v_{i_{j-1}}), s(v_{i_j})) : 1 \le i \le h,\ 1 \le j \le n_i - 1,\ S_{i_j} = \emptyset\} \cup \\
&\quad \{(tail(S_{i_j}), s(v_{i_j})) : 1 \le i \le h,\ 0 \le j \le n_i - 1,\ S_{i_j} \neq \emptyset\} \cup \\
&\quad \{(s(v_{i_{j-1}}), head(S_{i_j})) : 1 \le i \le h,\ 1 \le j \le n_i,\ S_{i_j} \neq \emptyset\}
\end{aligned}
$$

An example of a sparse compaction graph is given in Fig. 4.15(b). If we look at two consecutive layers, L_i and L_{i+1}, of a sparse normalized graph we have the following properties:

Property 4.9 *A middle segment s_e in L_i is either also in L_{i+1} or the corresponding q-vertex q_e is in L_{i+1}.*

Property 4.10 *A middle segment s_e in L_{i+1} is either also in L_i or the corresponding p-vertex p_e is in L_i.*

Lemma 4.11 *The ordering $<^o$ induced by the sparse compaction graph (N, A) of a sparse normalized graph $G_S = (V_S, E_S)$ defines a sparse layer ordering. The compaction graph (N, A) has linear size with respect to G.*

Proof: In the sparse compaction graph, each edge is induced by a vertex in V_S. Each vertex in V_S induces at most 2 edges. Therefore, the number of edges is at most $2|V_S|$. Since there are at most $2|V_S|$ lists S, the number of vertices in the compaction graphs is linear with respect to G. We prove that the compaction graph yields a total layer ordering by induction on the layers. In the first layer, there are no middle segments and the compaction graph of the sparse normalized graph is identical to the compaction graph of the normalized graph for this layer which defines a total layering. Assume that the compaction graph defines a total layer ordering for all layers above L_i. We show that for two consecutive middle segments, s_1 and s_2, in a list S_{i_j}, there is a path from s_1 to s_2 using vertices and middle segments defined in the layers above L_i. From this fact it follows that the compaction graph defines a total ordering for layer L_i. Let L_j, $j < i$ be the layer with the largest number j such that s_1 and s_2 are no longer consecutive in a list S_{j_k}. We show that there are only vertices of V_S between s_1 and s_2 in L_j. If there was a middle segment s_e between them, then this middle segment would end in a layer L_k with $j < k < i$, otherwise s_e would cross either s_1 or s_2, which contradicts the fact that no pair of middle segments crosses. But then, using Property 4.9 in layer L_{k+1}, there is a vertex q_e between s_1 and s_2, otherwise there would again be a pair of crossing middle segments. But this contradicts the definition of L_j, which is the layer with the largest layer number in which s_1 and s_2 are non-consecutive. Because there are only vertices of V between s_1 and s_2 in layer L_j, there is a path between s_1 and s_2 according to the definition of the compaction graph. □

Our new approach works as follows: In the first phase, we create a sparse normalization of the input graph. In the second phase, we perform crossing minimization on the sparse normalization. In the third phase, we take the resulting sparse compaction graph and perform a coordinate assignment in linear time using the approach of Brandes and Köpf [27]. We must still show how we can perform crossing minimization on a sparse normalization efficiently, which is the topic of the next section.

4.6.2 Efficient Crossing Reduction

In the following, we present an algorithm which performs crossing minimization on a sparse normalization. For our algorithm it is not important which one-sided two-layer crossing minimization heuristic we choose as long as it uses a linear measure m (see Definition 2.11). The output of our crossing minimization is a sparse compaction graph which induces a sparse layer ordering with the same number of crossings as the measure would produce for a common normalization.

4.6.2.1 Two-Layer Crossing Minimization

The input of our two-layer crossing minimization algorithm is an *alternating layer* L_i and the sparse compaction graph for the layers $L_1, \dots, L_i$. An alternating layer consists of an alternating sequence of vertices and *containers*, where each container represents a maximal sequence of middle segments. The output is an alternating layer L_{i+1} and the sparse compaction graph for $L_1, \dots, L_{i+1}$, in which the vertices and middle segments are ordered by a given linear measure. Note that the representation of layer L_i will be lost, since the containers are reused for layer L_{i+1}.

The containers correspond to the lists S introduced in Section 4.6.1. Note that the contained middle segments are ordered. The data structure implementing the container must support the following operations:

- $\mathbf{S} = \mathbf{create}()$: Creates an empty container S.
- $\mathbf{append}(\mathbf{S}, \mathbf{s})$: Appends middle segment s to the end of container S.
- $\mathbf{join}(\mathbf{S_1}, \mathbf{S_2})$: Appends all elements of container S_2 to container S_1.
- $(\mathbf{S_1}, \mathbf{S_2}) = \mathbf{split}(\mathbf{S}, \mathbf{s})$: Split container S at middle segment s into two containers S_1 and S_2. All elements less than s are stored in container S_1 and those which are greater than s in S_2. Element s is neither in S_1 nor S_2.
- $(\mathbf{S_1}, \mathbf{S_2}) = \mathbf{split}(\mathbf{S}, \mathbf{k})$: Split container S at position k. The first k elements of S are stored in S_1 and the remainder in S_2.
- $\mathbf{size}(\mathbf{S})$: Returns the number of elements in S.

Our algorithm `crossingMinimization(`L_i, L_{i+1}`)` is divided into six steps (Fig. 4.16):

1. We append the middle segment $s(v)$ for each p-vertex v in layer L_i to the container preceding v. Then we join this container with the succeeding container. The result is again an alternating layer (p-vertices are omitted).

2. We compute the measure values for the elements in L_{i+1}. First we assign a position value $pos(v_{i_j})$ to all vertices v_{i_j} in L_i. $pos(v_{i_0}) = size(S_{i_0})$ and $pos(v_{i_j}) = pos(v_{i_{j-1}}) + size(S_{i_j}) + 1$. Note that the pos values are the same as they would be for the median or barycenter heuristic if each middle segment was represented as a dummy vertex. Each non-empty container S_{i_j} has pos value $pos(v_{i_j-1})+1$. If container S_{i_0} is non-empty, it has pos value 0. Now we assign the measure to all non-q-vertices and containers in L_{i+1}. The initial containers in L_{i+1} are the resulting containers of the first step. Recall that the measure of a container in L_{i+1} is its position in L_i.

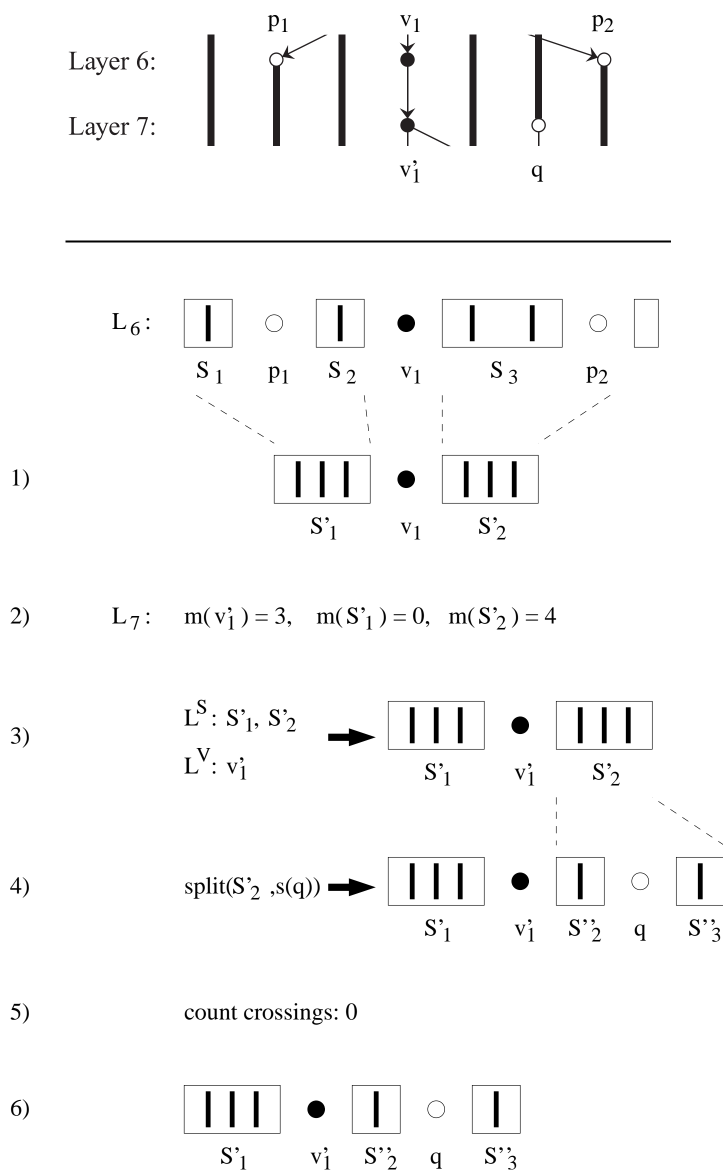


Figure 4.16: The six steps applied to layers L_6 and L_7 of Fig. 4.15(a).

3. We calculate an initial ordering of L_{i+1}. We sort all non-q-vertices in L_{i+1} according to their measure and store them in a list L^V. We do the same for the containers and store them in a list L^S. We use the following operations on these sorted lists:

   - $\mathbf{l = pop(L)}$: Removes the first element l from list L and returns it.
   - $\mathbf{push(L, l)}$: Inserts element l at the head of list L.

   We obtain L_{i+1} by merging list L^V with L^S as described by Algorithm 5.

Algorithm 5: `mergeLists`

Input: The list of vertices L^V and the list of containers L^S.
Output: The layer list L_{i+1}.

$L_{i+1} = \emptyset$;
while $L^V \neq \emptyset \ \wedge \ L^S \neq \emptyset$ **do**
  if $m(head(L^V)) \leq pos(head(L^S))$ **then**
    $v = pop(L^V),\ append(L_{i+1}, v)$;
  else if $m(head(L^V)) \geq pos(head(L^S)) + size(head(L^S)) - 1$ **then**
    $S = pop(L^S),\ append(L_{i+1}, S)$;
  else
    $S = pop(L^S),\ v = pop(L^V),\ k = \lceil m(v) - pos(S) \rceil$,
    $(S_1, S_2) = split(S, k),\ append(L_{i+1}, S_1),\ append(L_{i+1}, v)$,
    $pos(S_2) = pos(S) + k,\ push(L^S, S_2)$;
while $L^V \neq \emptyset$ **do**
  $v = pop(L^V),\ append(L_{i+1}, v)$;
while $L^S \neq \emptyset$ **do**
  $S = pop(L^S),\ append(L_{i+1}, S)$;
return L_{i+1};

4. We place each q-vertex v of L_{i+1} according to the position of its corresponding middle segment $s(v)$. We do this by calling $split(S, s(v))$ for each q-vertex v in layer L_{i+1} and placing v between the resulting containers (S denotes the container that includes $s(v)$).

5. We perform cross counting according to the scheme proposed by Barth et al. (see Section 2.4.3). During the cross counting step between layers L_i and L_{i+1}, therefore, we consider all layer elements as vertices. Besides the common edges between both layers, we also have to handle 'virtual edges', which are imaginary edges between a container element in L_i and the resulting container elements or q-vertices

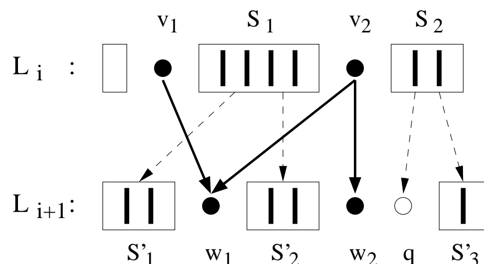


Figure 4.17: Example for our modified cross counting. Container S_1 is split into containers S'_1 and S'_2, container S_2 into a q-vertex and container S'_3. Dashed edges represent 'virtual edges'. Besides the common edges (v_1, w_1), (v_2, w_1) and (v_2, w_2) with *weight* $= 1$, we have the 'virtual edges' (S_1, S'_1) and (S_1, S'_2) both with *weight* $= 2$ as well as (S_2, q) and (S_2, S'_3), both with *weight* $= 1$. So the crossing between (v_1, w_1) and (S_1, S'_1) as well as the crossing between (v_2, w_1) and (S_1, S'_2) counts as two crossings.

in L_{i+1} (Fig. 4.17). In terms of the common approach, each 'virtual edge' represents at least one edge between two dummy vertices. The number of represented edges is equal to the size of the container element in L_{i+1}. We have to consider this fact to get the right number of edge crossings. We therefore introduce edge weights. The *weight* of a 'virtual edge' ending at a container element S is equal to $size(S)$. The *weight* of the other edges is one. Thus a crossing between two edges, e_1 and e_2, counts as $weight(e_1) \cdot weight(e_2)$ crossings.

6. We perform a scan on L_{i+1} and insert empty containers between two consecutive vertices, and call $join(S_1, S_2)$ on two consecutive containers in the list. This ensures that L_{i+1} is an alternating layer.

Finally, we create the edges in the sparse compaction graph for layer L_{i+1}.

4.6.2.2 The Overall Algorithm

In the crossing reduction phase we perform a layer-by-layer sweep on the sparse normalization and apply the two-layer crossing minimization as described above. During a reverse sweep we simply have to take the former p-vertices as q-vertices and vice versa. The first and the last layers never contain middle segments because of Properties 4.9 and 4.10. Therefore, when we perform a sweep or reverse sweep it is easy to create the initial alternating layer.

There are no other changes to the original Sugiyama approach except for the modified calculation of the linear measure m for all vertices in a layer, the normalization of the layer lists such that the lists are alternating and

the modified counting scheme for crossings. Furthermore, the horizontal coordinate assignment needs a minor modification. Recall that we are using the approach of Brandes and Köpf (see Section 2.4.4). In our case, there are no type 2 conflicts because we have used a linear measure and thus do not have crossing inner segments. We mark type 1 conflicts during the cross counting step. Since the conflicts are resolved in favor of the inner segments, we simply have to mark edges that cross a 'virtual edge'. Type 0 conflicts can be resolved as in the original approach. There are no further changes. We summarize this subsection with the following lemma:

Lemma 4.12 *Using a linear measure m, the approach described above gives the same result as the traditional crossing reduction.*

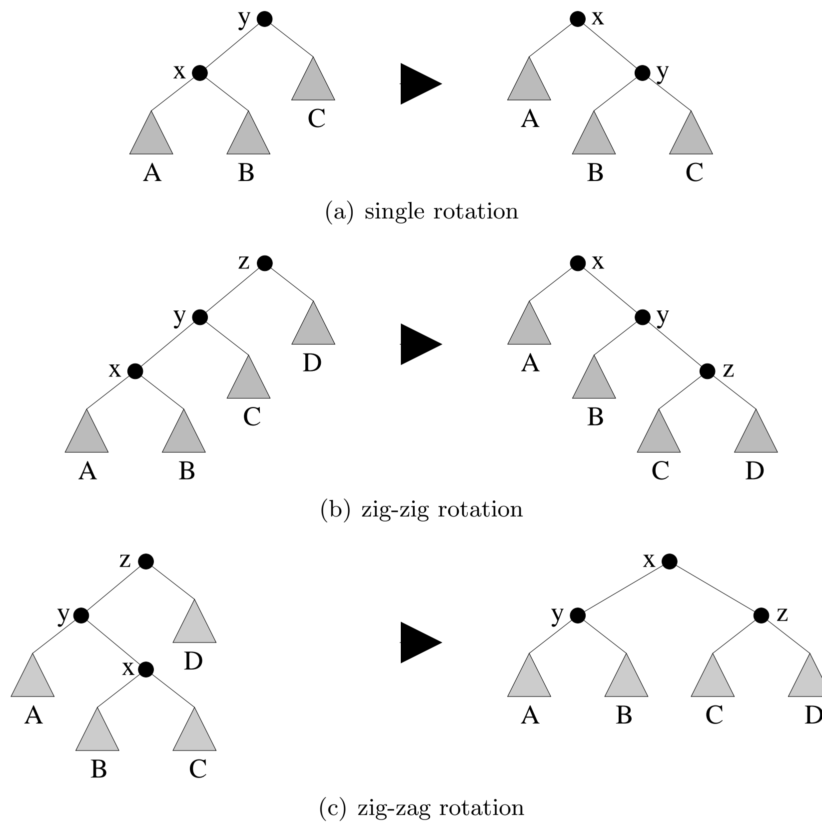
4.6.3 An Efficient Data Structure

When we use doubly linked lists to represent the containers, we are able to perform the append, size and join operations in time $O(1)$. Although we store a pointer to the split element, we cannot perform the split operation in time $O(1)$, since we have to update the size of the resulting containers. Hence we need $O(n)$ for splitting, where n denotes the maximal number of elements in a container. To be competitive, we need a data structure that supports append, split, join and size operations in $O(\log n)$. A standard binary search tree (not balanced) also requires $O(n)$. Thus we use splay trees, a data structure developed by Sleator and Tarjan [141]. Splay trees are self-adjusting binary search trees, which are easy to implement because the tree is allowed to become unbalanced and we need not keep information about its balance. Nevertheless, we can perform all required operations in $O(\log n)$ amortized time. A single operation might cost $O(n)$ but k consecutive operations starting from an empty tree take $O(k \log n)$ time. The basic operation on a splay tree is called a *splay*. Splaying vertex x makes x the root of the tree by a series of special rotations. There are three different kinds of rotations (Fig. 4.18). In the basic position, we denote the parent of x with y and the parent of y with z (if it exists).

- If y is the root of the tree we perform a **single rotation** between x and y. Fig. 4.18(a) shows a right rotation; a left rotation is symmetric. Note that this kind of rotation corresponds to a single rotation in an AVL tree [37]. It is performed at most once during a splay, because afterwards x is the root of the tree.

- If x and y are both left (right) children, we perform a **zig-zig rotation** (Fig. 4.18(b)). The effects are the same as performing a single rotation between y and z followed by a single rotation between x and y.

- If x is a right child and y is a left child, then we perform a **zig-zag rotation** (Fig. 4.18(c)). The effects are the same as performing two



(a) single rotation

(b) zig-zig rotation

(c) zig-zag rotation

Figure 4.18: Possible rotations during a splay (symmetric cases are omitted).

single rotations with x and its respective parent. The zig-zag rotation complies with a double rotation in an AVL tree. If x is a left and y is a right child, we perform the rotation in a symmetric manner.

The rotations are iteratively applied until x becomes the root of the tree.

Since we use splay trees to represent containers, we have to implement the corresponding operations.

- **size(S)** : When we perform a split operation we want to update the size of the resulting containers in $O(1)$. Therefore, each vertex knows the size of the subtree rooted by it. While performing the rotations we can update the size information at no extra cost. Let $size(x)$ denote the size of the subtree rooted at x before we perform the rotation and $size'(x)$ the size after the rotation. Furthermore, $size(A)$ denotes the size of subtree A. Recall that in order to obtain the size of a subtree it is sufficient to access the root of it. If we perform a single rotation, then $size'(x) = size(y)$ and $size'(y) = size(B) + size(C)$. If we perform a zig-zig rotation, then $size'(x) = size(z)$, $size'(z) =$

$size(C) + size(D)$ and $size'(y) = size'(z) + size(B)$. During a zig-zag rotation we have $size'(x) = size(z)$, $size'(y) = size(A) + size(B)$ and $size'(z) = size(C) + size(D)$.

- $\mathbf{append(S, s)}$: The append operation is performed once for each p-vertex. First, we search the rightmost element in the tree (last element in the container) by going from the root down always taking the right child and increasing the size of each vertex on the path between the root and the last element by 1. Now, we insert s as the right child of the rightmost element and then splay s.

- $\mathbf{join(S_1, S_2)}$: To join two containers, we search the rightmost element x of S_1, splay it and then make S_2 the right child of x. Furthermore, we set $size'(x) = size(S_1) + size(S_2)$. The join operation can only be invoked by an append operation or during the normalization of a layer list. Thus, it is invoked $O(|V| + |E|)$ times.

- $\mathbf{split(S, s)}$: The split operation is performed once for each q-vertex. First, we have to locate s in the container. We cannot perform a conventional tree search because the elements have only an implicit ordering (their container position), which they do no store. We can avoid a search by storing a pointer to s in the corresponding p-vertex (each q-vertex knows its corresponding p-vertex). So we just have to splay s and then take its left and its right children as roots for the resulting containers.

- $\mathbf{split(S, k)}$: This split operation is performed at most once for each common vertex. First, we search for the element at position k. We use a conventional binary tree search. Let $par(x)$ denote the parent of x and $l(x)$ ($r(x)$) the left (right) child of x. The positions are computed by the following formula: $pos(x) = pos(par(x)) + size(l(x)) + 1$, if x is a right child and $pos(x) = pos(par(x)) - size(r(x)) - 1$ if x is a left child. If x is the root, then $pos(x) = size(l(x)) + 1$. After we have found the element at position k, we just splay it and then take its right child as root for the second container.

All the above operations, except the size operation, are based on splaying. In [141] the following theorem is proved:

Theorem 4.13 ([141]) *A sequence of k arbitrary update operations on a collection of initially empty splay trees takes $O(k + \sum_{j=1}^{k} \log n_j)$ time, where n_j is the number of items in the tree or trees involved in operation j.*

The update operations include insert, join and split operations. The append operation is a special case of the insert operation and the size operation does not change the data structure. Each new iteration starts with

empty containers and there are at most $O(|E|)$ elements. Thus, the overall runtime of our crossing minimization is $O((|V| + |E|) \log |E|)$.

4.6.4 Runtime and Space Complexity

Our new technique leads to a noticeable reduction of the complexity of the popular algorithm of Sugiyama. We first normalize the graph by introducing at most $O(|E|)$ dummy vertices and edges. Then we perform the layer-by-layer sweep with the modified two-layer crossing minimization procedure. Using the splay-tree data structure as well as the cross counting scheme by Barth et al. [6], we can ensure that each crossing minimization step can be executed in time $O(n \log n)$, where n denotes the number of vertices and edges involved in this step. Summed over all layers, the time complexity remains $O((|V|+|E|) \log |E|)$. The coordinate assignment is performed in time $O(|V| + |E|)$ using the algorithm of Brandes and Köpf [27]. The space complexity is linear to the size of the input graph. Together with Lemma 4.12 we obtain the following theorem:

Theorem 4.14 *Our implementation of Sugiyama's approach is able to keep the number of dummy vertices and edges linear in the size of the graph without increasing the number of crossings. It has runtime $O((|V| + |E|) \log |E|)$ and requires $O(|V| + |E|)$ space.*

As the experiments in Section 7.3 show, our implementation clearly outperforms previous ones. We will now show how we can include our set of drawing constraints into it. In the remainder of this work, we will refer to our implementation as `Fast-Sugiyama`.

4.6.5 Including Drawing Constraints

Below, we sketch how to include the different constraints into `Fast-Sugiyama`. While there are no changes required for constraints FLOW, BIMODAL and PORT/SIDE, the sparse normalization demands some modifications for constraints CLUSTER and PARTITION.

4.6.5.1 Including Partitions

Recall that the layer elements are now vertices and containers. Instead of inserting a dummy vertex on each layer, we model the vertical lines which separate consecutive partition columns by means of middle segments, i.e., for each vertical line p^x_j we insert a dummy vertex into the first layer and one in the last layer and connect both with an edge. Let $s(p^x_j)$ denote the middle segment representing the vertical line p^x_j. When we build the subgraph G'_i for the i-th partition column (as described in Section 4.4.1.2), we obtain the relevant container elements by splitting the containers including $s(p^x_{i-1})$ and

$s(p_i^x)$. All containers between middle segments $s(p_{i-1}^x)$ and $s(p_i^x)$ are required for the crossing reduction of G'_i. After performing the crossing reduction, we place the resulting order between those middle segments. Note that we only need to handle non-empty partition columns of a layer, i.e., those with at least one vertex inside. If there are only middle segments inside a partition column, their order does not change. The number of additional join and split operations needed for handling partitions is $O(|E|)$. Furthermore, the number of additional vertices inserted during normalization is $O(|V|)$ since there are at most $|V|$ partition columns. Thus, the overall runtime of the crossing reduction for `Fast-Sugiyama` including partitions is $O(|E| \log |E|)$.

4.6.5.2 Including Clusters

To combine the clustered crossing reduction described in Section 4.4.1.2 with the `Fast-Sugiyama` implementation, we have to modify it to comply with the linear segments model. Analogous to the vertical lines separating partition columns, the left and right borders of cluster regions are now represented by middle segments. For a compound vertex c, let $s(c^l)$ and $s(c^r)$ denote the middle segment representing the left and right border. respectively. During a one-sided two-layer crossing minimization of a two-layered graph $G' = (L_1 \cup L_2, E')$, we first split $s(c^l)$ and $s(c^r)$ for each compound vertex $c \in C_2$ (i.e., the compound vertices of the layer cluster tree T_{L_2}). Since the size of C_2 is $O(|V|)$ and we have at most $O(|V|)$ layers, the overall number of container operations and thus that of layer elements is bounded by $O(|V|^2)$. Recall that for G' the clustered crossing minimization has runtime $O(|L'_2| \log |L'_2| + |C_2||E'| + |R|^2)$, where R denotes the set of constraints between elements of L_2. Considering containers as vertices with weighted edges, we can directly apply the clustered crossing minimization. However, to obtain drawings in the linear segments model, we have to insert additional constraints into R. More precisely, we insert a constraint between each pair of adjacent containers in L_2. This never introduces cycles into the constraint graph (L_2, R) since those constraints are derived from the (valid) layer ordering of L_1. Note that we only have to handle compound vertices of C_2 that contain at least one non-container element.

Since the number of elements inside a layer is $O(|V| + |E|)$, the number of additional constraints inserted into a layer is also $O(|V| + |E|)$. However, since the overall number of layer elements is bounded by $O(|V|^2)$, there are at most $O(\frac{|V|^2}{|E|})$ layers having $O(|V| + |E|)$ constraints. The number of constraints of the remaining layers is bounded by $O(|V|)$. Hence, the overall runtime complexity for applying the clustered crossing reduction approach to G_C is $O(|V|^2|E|)$. Note that while the runtime complexity of `Fast-Sugiyama` is still the same as for the common implementation, the space complexity is reduced to $O(|V| + x_E + x_C)$.

4.6.5.3 Runtime and Space Complexity

We end this chapter with a short overview of the time and space complexity of our planarization approach for the different constraints. Let $G = (V, E)$ denote a connected input graph and x_E the crossing number between edges of E in the initial drawing. Furthermore, x_C denotes the crossing number of edges of E with edges modeling cluster regions and x_P the crossing number of edges of E with edges of the partition grid graph. In Table 4.1 we summarize the runtime complexity for constructing a layered drawing using `Fast-Sugiyama`. The overall runtime of our planarization approach is given in Table 4.2 and the space complexity in Table 4.3. The presented results are derived from the runtime/space complexity stated in the different sections as well as from the above considerations.

Constraint	Time Complexity of Fast-Sugiyama
UPWARD	$O((\|V\|^2 + \|E\| \log \|E\|)$
BIMODAL	$O(\|V\|\|E\|)$
CLUSTER	$O(\|V\|^2\|E\|)$
PARTITION	$O(\|V\|^2 + \|E\| \log \|E\|)$
PORT/SIDE	$O(\|V\|^2\|E\| \log \|V\|)$

Table 4.1: Time complexity for constructing a layered drawing for the different constraints using `Fast-Sugiyama`.

Constraint	Overall Time Complexity
UPWARD	$O((\|V\| + x_E)\|E\|)$
BIMODAL	$O((\|V\| + x_E)\|E\|)$
CLUSTER	$O(\|V\|^2\|E\| + (x_E + x_C)\|E\|)$
PARTITION	$O((\|V\| + x_E + x_P)\|E\|)$
PORT/SIDE	$O(\|V\|^2\|E\| \log \|V\| + x_E\|E\|)$

Table 4.2: Overall time complexity of our planarization approach for the different constraints.

When we combine multiple constraints we have to sum the time and space complexity of the single constraints. If we combine all constraints, we obtain an overall runtime complexity of $O(|V|^2|E| \log |V| + (|V|+x_E+x_P+x_C)|E|)$ and require $O(|V|^2 + x_E + x_P + x_C)$ space. Note that the crossing number x_E depends highly on the given constraints. In our experiments in Section 7.3, we investigate the impact of the number of constraints on that of crossings.

Constraint	Space Complexity
UPWARD	$O(\|V\| + x_E)$
BIMODAL	$O(\|V\| + x_E)$
CLUSTER	$O(\|V\| + x_E + x_C)$
PARTITION	$O(\|V\| + x_E + x_P)$
PORT/SIDE	$O(\|V\|^2 + x_E)$

Table 4.3: Overall space complexity of our planarization approach for the different constraints. Recall that $|E| = O(|V| + x_E)$.

Chapter 5
Orthogonalization with Constraints

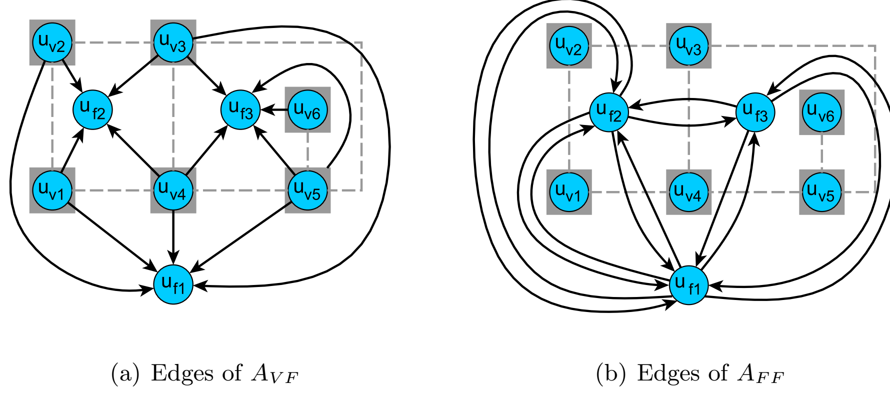
The result of the planarization phase is a planar representation $\mathcal{P}$ of the planarized input graph. Now we have to perform the orthogonalization phase which extends $\mathcal{P}$ to a quasi-orthogonal representation $\mathcal{H}$. Besides aesthetic criterion BEND, we also have to include the different drawing constraints here. Our orthogonalization is based on the Kandinsky model described in Section 2.3.2.1.

First, we review the network flow-based orthogonalization algorithm of Tamassia. Then we describe the Kandinsky network formulation which extends Tamassia's approach and produces orthogonal drawings in the Kandinsky model. Subsequently, we sketch two extensions for modeling prescribed angles and bends [26, 64]. Based on these extensions as well as an approach for realizing valid shapes of upward edges [64, 67], we describe an orthogonalization approach that includes our set of drawing constraints. Special issues which have to be considered during the orthogonalization with port/side constraints are discussed in a separate section. At the end of this chapter, we briefly sketch how to handle self-loops as well labels of graph elements.

5.1 Tamassia's Approach

In this section, we review the network flow-based algorithm of Tamassia [147] which computes bend-minimum orthogonal point drawings for plane 4-graphs with fixed embedding. Note that the number of bends heavily depends on the chosen embedding [49]. Minimizing the number of bends over all planar embeddings is NP-complete [87].

Let $G = (V, E)$ denote a planar 4-graph with given planar representation $\mathcal{P}$ and face set F. We use $\mathcal{P}$ to construct a network $\mathcal{N}(\mathcal{P}) = (U, A)$ whose minimum cost flow induces a bend-minimum orthogonal representation $\mathcal{H}$. Let $c\colon A \to \mathbb{N}$ denote the cost function, $u\colon A \to \mathbb{N}$ the capacity function and $b\colon U \to \mathbb{Z}$ the supply/demand function (see Section 2.1.1). The vertex set U consists of the following subsets:



(a) Edges of A_{VF} (b) Edges of A_{FF}

Figure 5.1: Construction of the network $\mathcal{N}(\mathcal{P})$. Gray elements denote vertices/edges of the underlying graph.

- A set U_V containing a vertex u_v for each vertex $v \in V$. The supply $b(u_v)$ of a vertex $u_v \in U_V$ is set to $4 - \delta_G(v)$.
- A set U_F containing a vertex u_f for each face $f \in F$. The supply/demand of a vertex $u_f \in U_F$ is set to:

$$b(u_f) = \begin{cases} -4 - \delta(f) & \text{if } f \text{ is the outer face,} \\ 4 - \delta(f) & \text{otherwise.} \end{cases}$$

The edge set A consists of the following subsets:

- A set A_{VF} containing edges $e^V_{\langle v,w \rangle} = (u_v, u_f)$ for each dart $\langle v, w \rangle$ of $P(f)$, $f \in F$ starting at a vertex $v \in V$ (Fig. 5.1(a)). Edges of A_{VF} have cost 0 and capacity ∞.
- A set A_{FF} containing edges $e^F_{\langle v,w \rangle} = (u_f, u_g)$ for each dart $\langle v, w \rangle$ of $P(f)$, $f \in F$ that separates face f and a face $g \in F$ with $f \neq g$ (Fig. 5.1(b)). Edges of A_{FF} have cost 1 and capacity ∞.

The above construction of $\mathcal{N}(\mathcal{P})$ is based on the following intuitive interpretation: The flow on an edge $e^V_{\langle v,w \rangle} = (u_v, u_f) \in A_{VF}$ represents the angle between the dart $\langle v, w \rangle$ and its cyclic predecessor in list $\mathcal{P}(f)$. A flow $x = f(e^V_{\langle v,w \rangle})$ defines an angle of $(x+1) \cdot 90°$ (Fig. 5.2(a)). Each flow unit on an edge $e^F_{\langle v,w \rangle} = (u_f, u_g) \in A_{FF}$ represents a bend on edge (v, w). The bend forms a $90°$ angle inside face f and a $270°$ angle inside face g (Fig. 5.2(b)). Note that due to the chosen costs of edges of A, the overall number of bends in the induced orthogonal shape $\mathcal{H}$ is

$$\#bends(\mathcal{H}) = \sum_{e \in A} c(e) f(e).$$

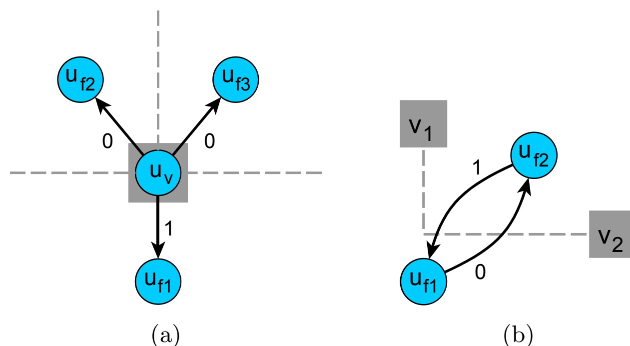


Figure 5.2: Interpretation of flow in the network $\mathcal{N}(\mathcal{P})$. Labels denote the flow value of edges of A according to the shape of the underlying graph.

Tamassia states the following theorem:

Theorem 5.1 ([147]) *Let $\mathcal{P}$ be a planar representation. The minimum number of bends for any orthogonal graph with planar representation $\mathcal{P}$ is equal to the minimum cost of a feasible flow in $\mathcal{N}(\mathcal{P})$. Furthermore, the orthogonal representation $\mathcal{H}$ of any optimal orthogonal graph can be computed from some optimal flow in $\mathcal{N}(\mathcal{P})$.*

With the optimized minimum cost flow algorithm proposed in [86] the runtime for calculating a bend-minimum orthogonal representation of a plane 4-graph $G = (V, E)$ is $O(|V|^{\frac{7}{4}}\sqrt{\log|V|})$.

5.2 The Kandinsky Network

In Section 2.3.2.1 we introduced the Kandinsky model. In this section we show the corresponding Kandinsky network flow formulation which is based on Tamassia's approach and calculates a quasi-orthogonal representation $\mathcal{H}$ for general planar graphs $G = (V, E)$ with given embedding $\mathcal{P}$. Our description is based on the work described in [64, 78].

5.2.1 Basic Network Formulation

We obtain the Kandinsky network $\mathcal{N}^K(\mathcal{P})$ by extending $\mathcal{N}(\mathcal{P})$ as follows: Since a flow of x over an edge $e \in A_{VF}$ models an angle of $(1 + x) \cdot 90°$, a $0°$ angle corresponds to negative flow which is not feasible in $\mathcal{N}(\mathcal{P})$. Hence, we model $0°$ angles by inserting additional vertices and edges into $\mathcal{N}^K(\mathcal{P})$ that allow flow coming from a vertex $u_f \in U_F$ to enter a vertex $u_v \in U_V$. To be compliant with the Kandinsky model properties, such a flow has to induce a vertex-bend on an edge incident to the $0°$ angle.

We realize this in the following way: For a vertex $v \in V$ let $f_0, \dots, f_{k-1}$ denote a (clockwise) ordered list of the faces around v and $e_0, \dots, e_{k-1}$ the edges that separate these faces, i.e., edge e_i separates face $f_{(i-1) \bmod k}$ and face f_i (Fig. 5.3(a)). For each face f_i, $0 \le i \le k-1$ around v we insert a vertex u_{h_i} into $\mathcal{N}^K(\mathcal{P})$ and set its supply to $b(u_{h_i}) = 0$. Furthermore, we add the following edges (Fig. 5.3(b)):

- Edges (u_{h_i}, u_v), $0 \le i \le k-1$ with cost 0 and capacity 1. Note that the chosen capacity avoids flow which would correspond to negative angles.
- Edges $(u_{f_{(i-1) \bmod k}}, u_{h_i})$ and $(u_{f_{(i+1) \bmod k}}, u_{h_i})$, $0 \le i \le k-1$ with cost 1 and capacity 1.

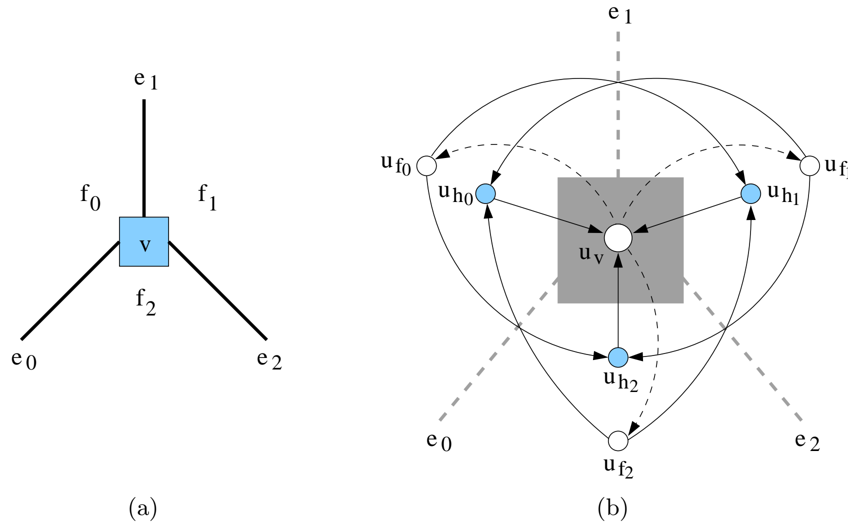


Figure 5.3: The Kandinsky network $\mathcal{N}^K(\mathcal{P})$. (a) shows a piece of the underlying graph that contains a vertex v incident to 3 edges. The corresponding vertices and edges of $\mathcal{N}^K(\mathcal{P})$ are illustrated in (b). New vertices are drawn as blue circles and new edges as solid lines.

For vertices representing crossings we do not add the above vertices and edges to $\mathcal{N}^K(\mathcal{P})$ since all angles between consecutive edges around crossing vertices are 90°. Up to now a feasible flow in $\mathcal{N}^K(\mathcal{P})$ has not induced a valid Kandinsky shape $\mathcal{H}$. Consider the example in Fig. 5.3(b). If there is flow on edge (u_{f_0}, u_{h_1}) as well as on edge (u_{f_1}, u_{h_0}), this would induce two vertex-bends at one endpoint of edge e_1, which is not allowed. We can fix this using an extended network flow definition. Let $D = \{d_0, \dots, d_k\}$ with $d_i \subseteq A$, $0 \le i \le k$ denote a partition of the edge set A. The elements d_i of D are called *devices*. We define a capacity function $u' \colon D \to \mathbb{N}$ on these devices. A minimum cost flow problem on $\mathcal{N}^K(\mathcal{P})$ is called an *edge partition*

minimum cost flow problem [64] if it is a minimum cost flow problem with the additional restriction that

$$\sum_{e \in d} f(e) \leq u'(d) \quad \forall d \in D.$$

Using an edge partition minimum cost flow problem, a valid Kandinsky shape $\mathcal{H}$ can be calculated by putting the edges $(u_{f_{(i-1) \bmod k}}, u_{h_i})$ and $(u_{f_{(i+1) \bmod k}}, u_{h_i})$ for each vertex u_{h_i} into one device d with capacity $u'(d) = 1$.

In [64] it was shown that solving the above edge partition minimum cost flow problem is NP-hard. Note that this does not imply that finding a bend-minimum Kandinsky drawing is also NP-hard. The complexity of this problem is still unknown. There is a 2-approximation for the Kandinsky bend minimization problem which is based on a relaxation of the edge partition minimum cost flow problem and runs in time $O(|V|^{\frac{7}{4}}\sqrt{\log|V|})$. Furthermore, there is an improved heuristic that runs in time $O(|V|^{\frac{11}{4}}\sqrt{\log|V|})$ and produces satisfying results in practice [64].

5.2.2 Incorporating Prescribed Angles and Bends

Below, we show how to incorporate prescribed angles and bends into the Kandinsky network $\mathcal{N}^K(\mathcal{P})$. More precisely, we review the approach given in [26, 64] that allows defining target values for angles formed by intervals of consecutive edges around a vertex as well as the number and types of bends on edges. Both can be realized in a natural way since in network flow-based orthogonalization approaches, there is a correspondence between angle size/bends and flow on edges. We use these extensions to obtain a quasi-orthogonal shape $\mathcal{H}$ that includes our set of drawing constraints.

Let AC denote the set of given *angle-constraints*. An angle-constraint $ac \in AC$ is a tuple (I, v, a, c) where $I \subseteq E$ denotes an interval of consecutive edges around vertex $v \in V$, $a \in \{0, 90, 180, 270, 360\}$ denotes the target value of the sum of angles defined by the edges of I around v and $c \in \mathbb{N}$ the cost for deviations of this target value. $AC_v \subseteq AC$ denotes the set of angle-constraints associated with a vertex $v \in V$. Note that for any pair of angle-constraints $ac = (I, v, a, c)$ and $ac' = (I', v, a', c')$ of AC_v the intervals I and I' are not allowed to intersect except at their endpoints. For each angle-constraint $ac = (I, v, a, c) \in AC_v$, $v \in V$ we insert a vertex u_{ac} (called angle-vertex) into $\mathcal{N}^K(\mathcal{P})$. Furthermore, we insert two directed edges (u_{ac}, u_v) and (u_v, u_{ac}). Both edges have cost c and capacity ∞. The supply of a vertex u_{ac} is set to

$$b(u_{ac}) = \frac{a}{90} - |I| + 1$$

and the supply of u_v to

$$b(u_v) = \delta_G(v) - 4 - \sum_{ac \in AC_v} b(u_{ac}).$$

All edges originally connected to vertices of U_V are now connected to the corresponding angle-vertices. Figure 5.4 illustrates the above modifications.

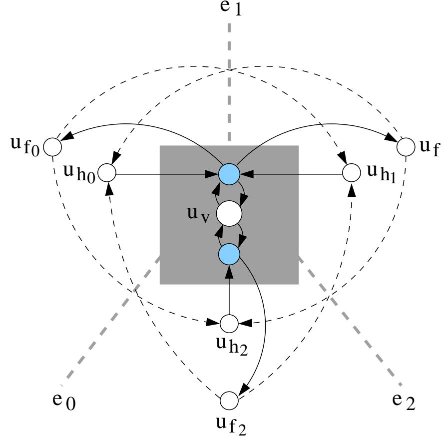


Figure 5.4: Incorporating prescribed angles into the Kandinsky network. Blue circles denote the inserted angle-vertices and solid edges inserted or redirected edges. The example shows an angle-constraint for the interval $[e_0, e_1, e_2]$ as well as for the interval $[e_2, e_0]$.

Let BC denote the set of given *bend-constraints*. A bend-constraint $bc \in BC$ is a tuple (e, s, c_i, c_d) where $e = (v, w) \in E$ denotes an edge and s a bit string. The k-th bit of s represents the k-th prescribed bend b_k that should appear when going along edge e from v to w. A "1" represents a left bend with respect to the edge's direction and a "0" a right bend. If e should have no bend, s is set to the empty string ϵ. The third entry $c_i \in \mathbb{N}$ states the cost of inserting an additional bend into e and the last entry $c_d \in \mathbb{N}$ the cost for a prescribed bend that is not considered. Note that each bend b_i, $0 \leq i < |s|$ in a bend-constraint $bc = (e = (v, w), s, c_i, c_d) \in BC$ is treated as a vertex of degree two. For each b_i we insert two vertices $u_{(e,i)}$ and $u'_{(e,i)}$ into $\mathcal{N}^K(\mathcal{P})$ (Fig. 5.5(a)). The supply of $u_{(e,i)}$ is set to $b(u_{(e,i)}) = 2$ and the supply of $u'_{(e,i)}$ to $b(u'_{(e,i)}) = 0$. Both vertices are connected by a directed edge $(u_{(e,i)}, u'_{(e,i)})$ with cost 0 and capacity 1. Let f_1 denote the face in which the bend should form a 270° angle and f_0 the face in which the bend should form a 90° angle. We add a demand of one to the vertices u_{f_1} and u_{f_0} (i.e., we set $b(u_f) = b(u_f) - 1$). Additionally, we add an edge $(u_{(e,i)}, u_{f_1})$ with cost 0 and capacity 2 as well as an edge $(u'_{(e,i)}, u_{f_0})$ with cost c_d and capacity 1. The effect of this modification is that $\mathcal{H}$ either contains the prescribed bend b_i at zero cost or removes it at cost c_d. If b_i is the first $(i = 0)$ or

last ($i = |s| - 1$) prescribed bend of bc, we add an additional directed edge $(u'_{(e,i)}, u_{h_1})$ to $\mathcal{N}^K(\mathcal{P})$ as shown in Fig. 5.5(b). The cost of this edge is set to 0 and its capacity to 1. This ensures that vertex-bends can be confirmed at zero costs. Note that edge $(u'_{(e,i)}, u_{h_1})$ is assigned to the same device as the edges (u_{f_0}, u_{h_1}) and (u_{f_1}, u_{h_0}) since only one of these edges is allowed to carry flow. Additionally, we assign cost c_i to edge $e^F_{\langle v,w\rangle}$ and $e^F_{\langle w,v\rangle}$. Hence, each additional bend on edge e has cost c_i. A more detailed description for both modifications stated above can be found in [64].

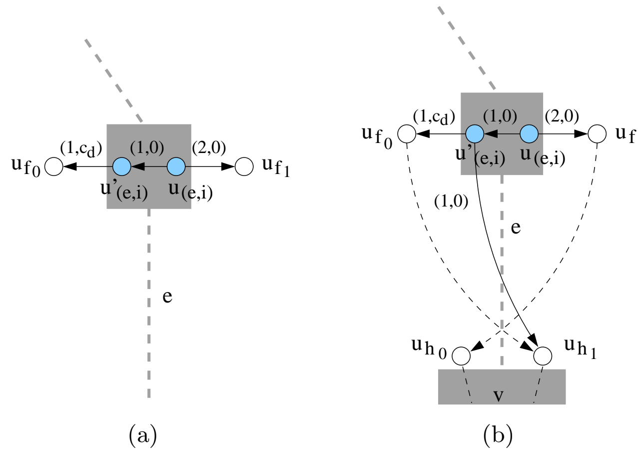


Figure 5.5: Incorporating prescribed bends into the Kandinsky network. Blue circles denote inserted vertices. The first entry of an edge label denotes the capacity and the second the cost of the corresponding edge.

5.3 Incorporating Constraints

In this section we describe an orthogonalization approach that includes constraints FLOW, CLUSTER, BIMODAL and PARTITION. Furthermore, we show how to deal with vertices of types `two_sided`, `hyper_dummy` and `note_dummy`. Constraint PORT/SIDE is handled in Section 5.4.

The input of the orthogonalization phase is a planarized graph $G = (V, E)$ and its port constraint preserving bimodally mixed-upward p,c-planar embedding $\mathcal{P}$ calculated during the planarization phase. We assume that all dummy vertices representing crossings are marked accordingly. Our orthogonalization approach consists of the following three steps:

1. Let $G_\uparrow$ denote the induced subgraph of G on the upward edges $E_\uparrow$. Note that $G_\uparrow$ contains all skeleton edges inserted to guarantee a uniform orientation of the vertices (see Section 4.1). In the first step, we assign a fixed upward shape to each edge of $G_\uparrow$. Therefore, we calculate a tail- and a head-shape for each edge. The tail-shape (head-shape)

depends on the type of the corresponding source (target) vertex. The algorithm iteratively chooses a vertex of $G_{\uparrow}$ and assigns the head-shape to the incoming edges and the tail-shape to the outgoing edges. The head-/tail-shapes assigned to edges incident to vertices v of type `hyper_dummy` are shown in Fig. 5.6(a),(b). We always choose the middle of the incoming/outgoing edges to be straight on v. For crossing vertices we use the shapes shown in Fig. 5.6(c). Note that if one of the edges incident to a crossing represents a vertical segment of the partition grid graph, it is chosen to be the straight edge. For edges incident to other vertices we assign the shapes illustrated in Fig. 5.6(d). The final shape of an edge is constructed by concatenating its tail- and head-shape. In [67] we successfully applied a similar strategy and showed that there is always a valid Kandinsky drawing in which the shapes of edges correspond to the shapes assigned above.

We use the bend-stretching transformations described in [67] to remove superfluous bends. Bend-stretching successively scans the shape of edges and if a shape contains a specific pattern it is replaced by a new shape with less bends. To maintain a valid upward drawing, the first and last direction of a shape is never changed.

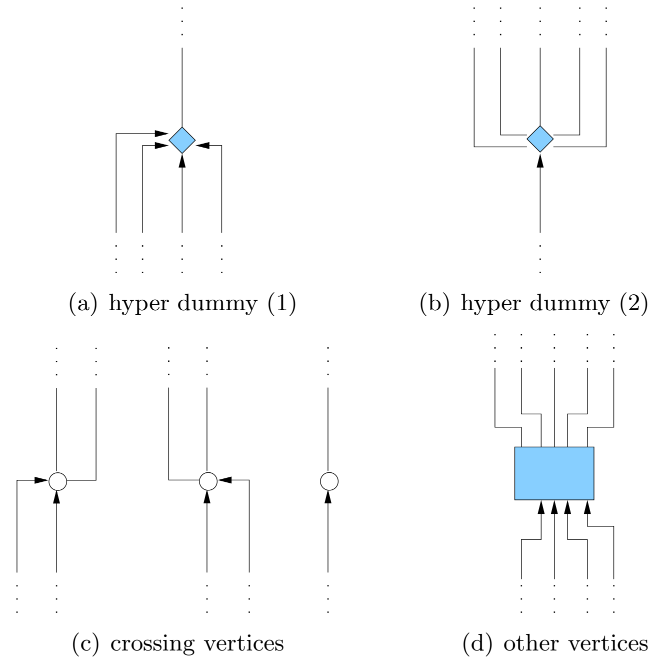


Figure 5.6: Different shapes assigned to upward edges.

2. In the second step, we consider all edges of G. To satisfy constraint TWO_SIDED_VERTICES we have to put restrictions on the angles between specific edges. More precisely, for vertices of type `two_sided`,

angles between incoming and outgoing edges are fixed to 180°, as shown in Fig. 5.7(a). If there are only incoming (outgoing) edges on such a vertex we define a 360° angle between the first and last incoming (outgoing) edges as shown in Fig. 5.7(b). The angles around a vertex of type `note_dummy` are fixed to the values shown in Fig. 5.7(c). Furthermore, Fig. 5.7(d) shows the angles around vertices of the partition grid graph. Note that the angle assignment is consistent with the fixed shapes calculated in the first step. This is necessary because there could be both edge types on a vertex.

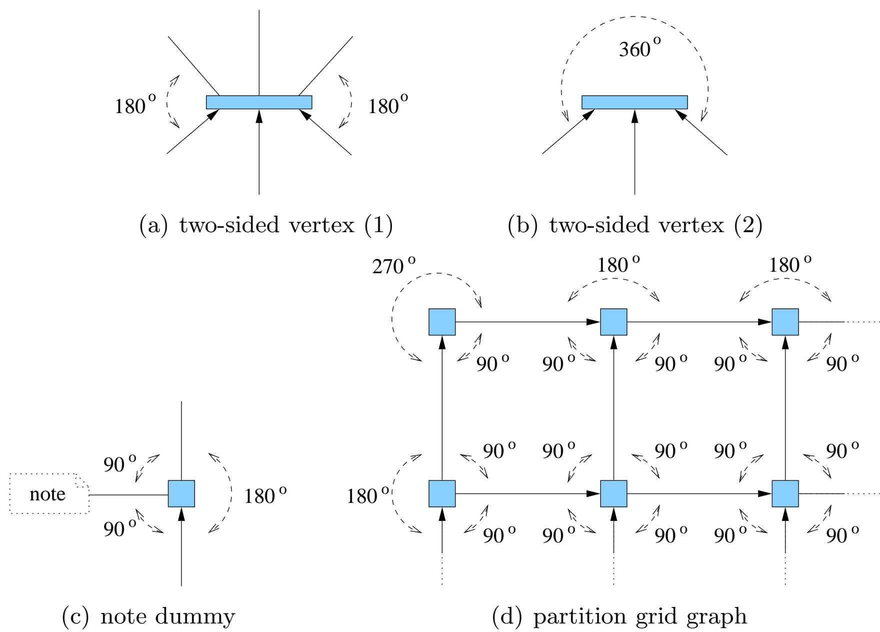


Figure 5.7: Different angles assigned to edges around vertices.

3. In the third step, we calculate the shapes for edges of E^*. Shapes assigned to upward edges as well as prescribed angles specified in the previous steps are not allowed to change. Furthermore, all edge segments of the partition grid graph are not allowed to bend. We model these requirements by constructing a minimum cost flow network $\mathcal{N}^K(\mathcal{P})$ for G and adding appropriate angle- and bend-constraints. The transformation of the shapes assigned to upward edges into a set of angle- and bend-constraints is described in [67]. Note that there is always a valid Kandinsky drawing that satisfies all these constraints. For each segment e of the partition grid graph we add a bend-constraint $bc = (e, \epsilon, c_i, c_d)$. Now, as described in Section 5.2.2, we map the angle- and bend-constraints to $\mathcal{N}^K(\mathcal{P})$. Note that the number of bends assigned to an edge $e \in E$ is at most 4. Furthermore, the number of

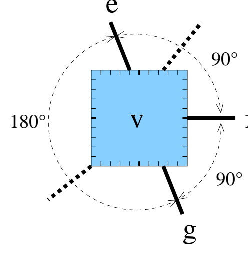


Figure 5.8: Angles induced by edges e, f and g with side constraints sc_e^v, sc_f^v and sc_g^v ($side(sc_e^v) = t$, $side(sc_f^v) = r$, $side(sc_g^v) = b$). Dashed edges do not have port/side constraints on v.

inserted angle-constraints is bounded by $O(|E|)$. Hence the size of $\mathcal{N}^K(\mathcal{P})$ is still linear to the size of the planarized input graph.

We realize the rectangular shape of cluster regions as described in [112] by putting additional constraints on the flow traversing edges that represent the region border. Note that each edge segment e which represents the border of a cluster region $c \in C$ separates two distinct faces, f and g. Let f denote the face lying outside of c. Then, for each such segment e, we remove the corresponding edge (u_f, u_g) from the Kandinsky network $\mathcal{N}^K(\mathcal{P})$.

To obtain the shapes of edges of E^*, we apply the network flow algorithm described in [64] to $\mathcal{N}^K(\mathcal{P})$. This algorithm runs in time $O(|V|^{\frac{7}{4}}\sqrt{\log|V|})$ and has the property that the number of bends on edges of E^* is not more than 3 times the minimum number of bends in a Kandinsky shape that satisfies the given constraints. Let $\mathcal{C}$ denote the overall cost of the minimum cost flow. If the costs assigned to the angle- and bend-constraints (c, c_i and c_d) are larger than $\mathcal{C}$, we always obtain a valid Kandinsky drawing that satisfies the given constraints. For our experiments, setting $c = c_i = c_d = 5|E|$ was always sufficient.

Since the first two steps can be implemented in linear time, the overall runtime of the above approach is $O(|V|^{\frac{7}{4}}\sqrt{\log|V|})$.

5.4 Adding Port and Side Constraints

To incorporate port/side constraints into the orthogonalization phase, we have to consider different issues. Recall that for each constraint $sc_e^v \in SC_G \cup PC_G$, edge e should enter/leave v at side $side(sc_e^v)$. Hence, as shown in Fig. 5.8, port/side constraints define angles between edges around vertices. These angles can be realized with the approach described in the previous section.

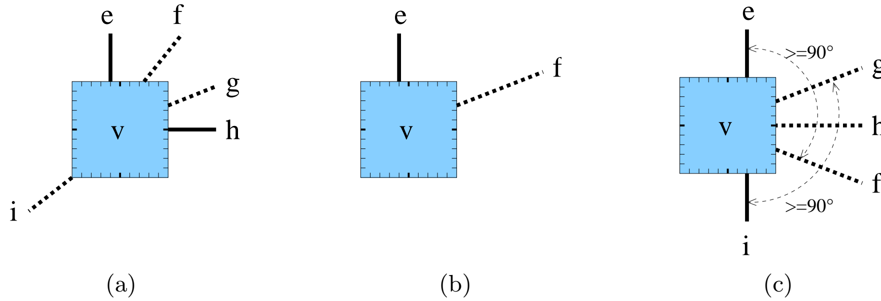


Figure 5.9: Different examples that illustrate the `straight-line edge assignment issue` as well as the `no pin left issue`. Edges without constraints are drawn dashed.

In the `sc` scenario (only side constraints) we are always able to place straight-line edges at the center of the corresponding vertex sides. However, when we include port constraints, there are different cases which prevent such a placement (Fig 5.9). We call this issue the `straight-line edge assignment issue`. Recall that in our drawing model an edge can only be drawn straight-line if it can be assigned to the κ-th pin of the respective vertex side on both endpoints without changing the given embedding. In Fig. 5.9(a), edges e and h have port constraints pc_e^v and pc_h^v with $side(pc_e^v) = t$, $location(pc_e^v) = 4$ as well as $side(pc_h^v) = r$, $location(pc_h^v) = 5$ (recall that on each vertex, side pins are numbered in clockwise order). Since edge h is assigned to pin $5 = \kappa$ of the right side, it can be centered there. Edge e cannot be drawn straight-line because it is fixed to a pin $\neq \kappa$. Since edge h blocks the center pin on the right side, edges f, g and i cannot be centered on that side. While edge f can be centered at the top, edge g cannot be centered here because there would be no free pin left for edge f. Edge i can be centered at the left side or bottom of v. It cannot be centered at the top, because edge e prevents an assignment to a pin larger than 3. Fig. 5.9(b) shows a special case where f can only be centered at the top of v if it is placed to the right of e.

Lemma 5.2 *Incorporating the* `straight-line edge assignment issue` *may introduce at most* $4(|V| - \sqrt{|V|})$ *additional bends.*

Proof: Assume that we have a Kandinsky shape $\mathcal{H}$ which does not consider the `straight-line edge assignment issue`. Then we can construct a shape that incorporates this issue by adding two additional bends to affected straight-line edges as shown in Fig. 5.10(a). Obviously, the resulting shape still maintains the Kandinsky properties. Since we introduce at most two additional bends for each straight-line edge and the number of those edges is bounded by $2(|V| - \sqrt{|V|})$ (see Lemma 2.8), the overall number of additional

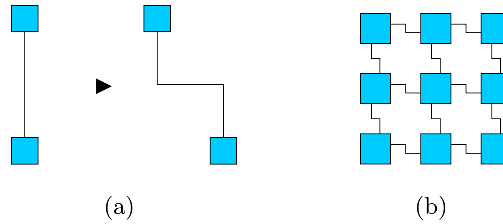


Figure 5.10: (a) illustrates a valid transformation of a straight-line edge that introduces two additional bends. In (b) we demonstrate that $4(|V| - \sqrt{|V|})$ is a tight bound on the number of additional bends required when we incorporate the `straight-line edge assignment issue`. Assume that each edge of the 3×3 grid has a port constraint which is associated with a pin $\neq \kappa$. Then, instead of a drawing without bends, we obtain the drawing with 24 bends (b).

bends is less than or equal to $4(|V| - \sqrt{|V|})$. As shown in Fig. 5.10(b) this bound is tight. □

In the `mc` scenario there may appear edges which could not be placed on a given side, because there are not enough pins left. We call this issue the `no pin left issue`. Fig. 5.9(c) shows two edges, e and i, with port constraints pc_e^v ($side(pc_e^v) = t$, $location(pc_e^v) = 7$) and pc_i^v ($side(pc_i^v) = b$, $location(pc_i^v) = 3$) as well as three (dotted) edges, g, h and f, without constraints on v. Edge f cannot be placed at the top and edge g cannot be placed on the bottom of v, because on both sides there are only two free pins when we have to preserve the given embedding. As shown in Fig. 5.9(c), this issue can be modeled with additional angle-constraints. However, this kind of angle-constraint (angles between overlapping edge intervals) cannot be handled by common network flow-based approaches.

Lemma 5.3 *Incorporating the* `no pin left issue` *may introduce at most* $2|E|$ *additional bends.*

Proof: Assume that we have a Kandinsky shape $\mathcal{H}$ which does not consider the `no pin left issue`. To incorporate it, we successively identify affected edges for each vertex $v \in V$ and assign them to the cyclic previous/next side of v by adding one additional bend as shown in Fig. 5.11. Note that a port constraint preserving embedding always ensures that there is a free pin on the previous/next side. The resulting shape still maintains the Kandinsky properties. Since each edge requires at most one additional bend on each of its endpoints, the overall number of additional bends introduced by this issue is bounded by $2|E|$. □

Now, after looking at the different issues that need to be considered during the orthogonalization, we can realize port/side constraints as follows:

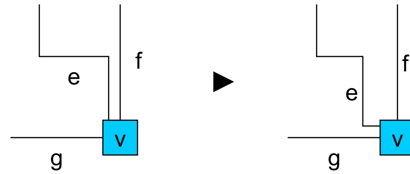


Figure 5.11: A valid transformation that resolves the `no pin left issue`. Assume that we have to assign edge f to the leftmost pin on the top of v and edge g to a pin on the left side. In the given embedding, edge e cannot be assigned to the top of v. Hence, we assign it to the left side by adding one additional bend. Note that for a port constraint preserving embedding there is always a free pin between edges g and f.

For each non-dummy vertex $v \in V$ we determine the angles between incident edges with port/side constraints on v. Such angles can be realized with the approach described in the previous section. Note that a port constraint preserving embedding ensures that there is a valid Kandinsky drawing realizing these angles. Since the number of inserted angle-constraints is linear to the number of edges, the runtime complexity of the orthogonalization is still $O(|V|^{\frac{7}{4}}\sqrt{\log|V|})$. A uniform orientation of the vertices is guaranteed by inserting appropriate skeleton edges as described in Section 4.1. The `straight-line edge assignment issue` and the `no pin left issue` are handled after the orthogonalization by inserting additional bends as depicted in the proofs of Lemma 5.2 and Lemma 5.3. This can be done in linear time.

5.5 Handling of Self-Loops

We propose the following strategy to handle self-loops: Before we apply the layout algorithm, we first remove all self-loops from the input graph. After the orthogonalization phase, we reinsert them by choosing a suitable corner on the corresponding vertex. As shown in Fig. 5.12(a), a self-loop can be placed at each of the four corners without introducing new crossings and with a minimum number of bends (3) in the Kandinsky model. There are two exceptions to this rule: First, port and side constraints on a self-loop can prevent such a placement, e.g., if a self-loop has two side constraints lying on opposite vertex sides (Fig. 5.12(b)). Second, port constraints on other edges incident to the same vertex can prevent a self-loop from being placed on a corner since the required pins might be occupied and, thus, the insertion of the self-loop would introduce additional crossings. In both cases we suggest inserting those self-loops in a postprocessing step without worrying about the Kandinsky model properties.

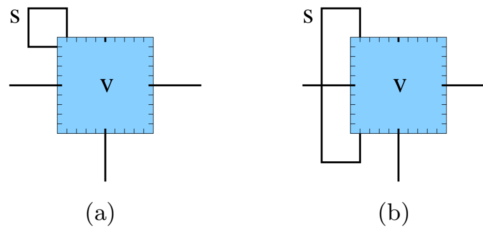


Figure 5.12: Placement of a self-loop s.

5.6 Placement of Labels

Labels can be placed at vertices, edges, clusters and partitions (LABEL). Edge labels $l \in L_E$ with $pos(l)$ = `source` or `target` are placed during a postprocessing step by a map labeling algorithm [153]. All edge labels with $pos(l)$ = `center` are placed after the orthogonalization phase. At that time, edges are paths of horizontal and vertical segments. For each label $l \in L_E$ with $refer(l) = e$ we choose one of e's middle segments and split it into two parts by inserting a dummy vertex (Fig. 5.13). The dummy vertex represents label l and thus gets size $size(l)$. After the compaction phase such dummy vertices are removed and replaced by the corresponding label. Note that we always align labels horizontally (HORIZONTAL_LABELS).

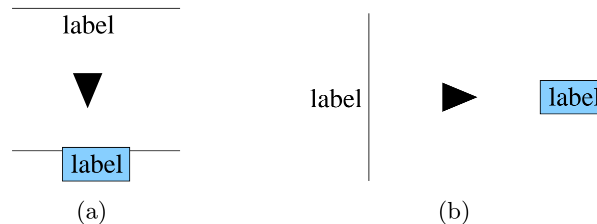


Figure 5.13: Placement of labels on a horizontal/vertical edge segment.

Analogously, for labels of clusters and partition cells we split the edge representing the top of the enclosing rectangle (Fig. 5.13(a)). For regular partitions we provide an alternative strategy which is typically used to label swimlanes (e.g., in UML activity diagrams). Instead of labeling each partition cell separately, we only label each row/column of the partition. Thus, semantically, the label of a row/column applies to each partition cell of that row/column. We realize this in the following way: For each column of the partition we split the edge representing the top border of the corresponding rectangle and for each row the edge representing the left border. Vertex labels are placed inside the corresponding vertex and thus do not require special handling.

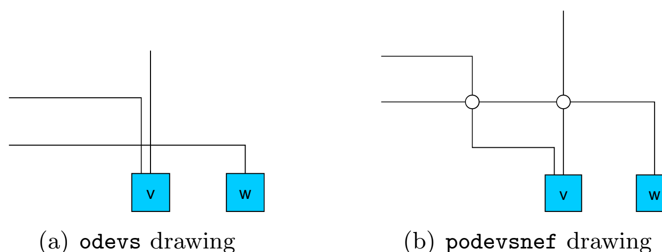
Chapter 6

Alternatives for Realizing Port/Side Constraints

In this chapter we sketch and analyze alternative approaches for realizing port/side constraints. First, we describe a fast orthogonal drawing approach which implements the so-called three-phase method [13] instead of the TSM approach. It produces port constraint preserving drawings in the `odevs` model, which is introduced below. In the second section, we come back to the TSM approach and present alternative planarization approaches which are mainly derived from state-of-the-art planarization techniques. Two alternative orthogonalization approaches which produce port constraint preserving drawings in the Kandinsky model are given in the third section. In the last section, we list additional requirements needed to apply port/side constraints to a wider field of applications.

6.1 An Alternative Drawing Method

While for orthogonal drawings without constraints TSM-based approaches have been shown to be clearly superior to other known approaches (see [48]), this is not necessarily true when we include constraints. Thus, in this section, we take a look at an alternative drawing approach which is based on the three-phase method [13]. This method belongs to the so-called draw-and-adjust approaches which work directly on the geometry of a drawing. The three-phase method consists of the following phases: First, the vertices are placed onto a rectilinear grid. In the second phase, edges are routed according to their endpoints. Overlap among edges as well as intersections between edges and vertices may occur during this phase. They are removed in the third phase, where the ports of edges are determined (each edge is assigned to a pin of its incident vertices). Compared to layout algorithms based on the TSM approach, algorithms implementing the three-phase method are usually faster, much simpler to implement and are better suited to include constraints on the relative and absolute positions of vertices.



(a) `odevs` drawing (b) `podevsnef` drawing

Figure 6.1: An example drawing in the `odevs` model (a), and in the `podevsnef` model (b). In (b) crossings (white circles) are handled like vertices.

6.1.1 The `Odevs` Model

The Kandinsky model assumes that the underlying graph is planar and is thus dedicated to approaches where non-planar graphs are planarized first. Since the three-phase method does not demand a planarization, we introduce a more suitable drawing model called the *`odevs` model* (**o**rthogonal **d**rawing with **e**qual **v**ertex **s**ize). Note that the number of crossings produced by algorithms implementing the three-phase method is usually much higher than that for TSM-based algorithms. Recall that the `podevsnef` model utilizes a coarse uniform grid for placing vertices and a finer grid for routing edges (see Section 2.3.2.1). The `odevs` model uses the same grid structure as well as vertices of equal size. Furthermore, neither model allows intersections between vertices and edges. The `odevs` model differs from the `podevsnef` model in the following points: Since a drawing in the `podevsnef` model is subject to a planar or planarized graph, crossings are handled like common vertices and thus are always placed on intersection points of lines of the coarse grid. The `odevs` model has a more appropriate handling of crossings. It does not require a planarized graph and places crossings on the finer grid lines, too. Furthermore, in the `podevsnef` model, the bend-or-end property is required for the edges of the planarized graph. In the `odevs` model we only demand the bend-or-end property for the original edges, which often saves a lot of bends as shown in Fig. 6.1. In the `podevsnef` drawing (b) the left edge on the top of v has to bend before it reaches the crossing vertex to satisfy the bend-or-end property regarding the planarized graph. In the `odevs` drawing (a) the edge bends after the crossing, which is sufficient to satisfy the bend-or-end property in the `odevs` model. The `odevs` model does not exclude empty faces if they can be drawn without intersections between vertices and edges. Thus, the L-triangle (see Section 2.3.2.1) is still excluded. Table 6.1 summarizes the main differences between the two drawing models.

Note that any drawing in the `podevsnef` model is also a valid drawing in the `odevs` model. Thus, we have the following property:

	`podevsnef`	`odevs`
bend-or-end property	apply to the planarized graph	apply to the original graph
empty faces	not allowed	allowed if drawable
crossings	handled like vertices	special handling

Table 6.1: Differences between the `podevsnef` and `odevs` drawing models.

Property 6.1 *For a given graph G, the minimum number of bends of a drawing in the* `odevs` *model is less than or equal to that of a drawing in the* `podevsnef` *model.*

Below we present a theorem about the complexity of minimizing bends in `odevs` drawings with port/side constraints. We also give a tight upper bound on the bend number of such drawings.

Theorem 6.2 *The bend minimization in the* `odevs` *model with side as well as port constraints is NP-hard.*

Proof: The proof is based on a reduction of the minimum feedback arc set problem (introduced in Section 2.4.1) to the `odevs` bend minimization problem. Let $G_D = (V, A)$ denote a directed graph with feedback arc set A'. The directed input graph $G^* = (V^*, A^*)$ of the corresponding bend minimization problem consists of the vertex set $V^* = V \cup N$ where N contains two vertices n_1^v, n_2^v for each vertex $v \in V$ and the edge set $A^* = A \cup E'$ where E' contains two directed edges (n_1^v, v) and (v, n_2^v) for each vertex $v \in V$. We put the following side constraints on the edges: for each edge $e = (v, w) \in A^*$ we restrict e to leave v on the right side ($side(sc_e^v) = r$) and enter w on the left side ($side(sc_e^w) = l$).

First, we show that in a port constraint preserving, bend-minimum `odevs` drawing, all edges of E' are drawn straight-line (without bends). Assume that there is a bend-minimum drawing with an edge $e = (u, n_2^u) \in E'$ that is not drawn straight-line. To satisfy the side constraints, e must have at least 2 bends (with the edge route shown in Fig. 6.3(b)). Then there is a straight-line edge $a = (u, v) \in A$ leaving u at the same side as edge e. Otherwise, the number of bends can be reduced since edges of E' can always be drawn straight-line (all n-vertices have degree one). Since edge a is drawn straight-line there is also an edge $e' = (n_1^v, v) \in E'$ with at least 2 bends; see Fig. 6.2(a). Thus, the number of bends can be reduced by drawing a with 2 bends and e, e' straight-line as shown in Fig. 6.2(b). Note that this can be done without changing the shape of the remaining edges. Hence, in a bend-minimum drawing each edge of E' is drawn without bends.

It follows that each edge of A has at least 2 bends (with the edge route shown in Fig. 6.3(b)). Such a route is always possible for edges $e = (u, v) \in A$

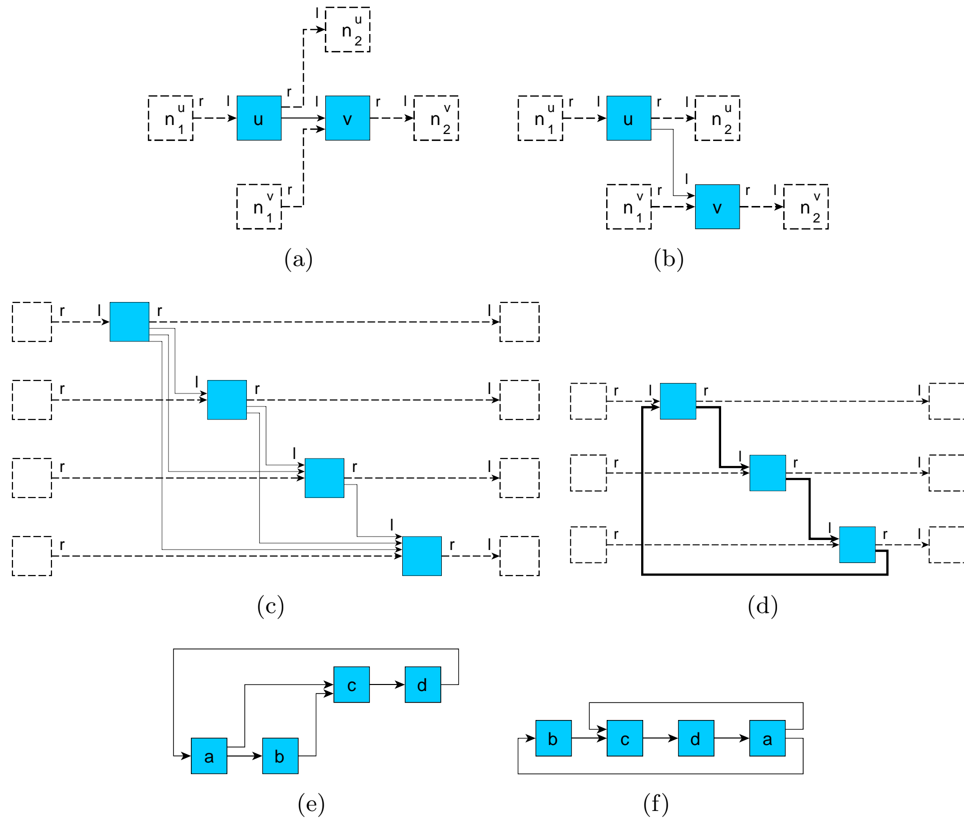


Figure 6.2: Illustration of Theorem 6.2. (a) and (b) demonstrate why edges of E' (drawn as dashed lines) are always drawn straight-line in a bend-minimum `odevs` drawing. (c) shows a drawing strategy for acyclic graphs that yields bend minimum results for the given port/side constraints. (d) depicts a bend-minimum drawing of an edge cycle. (e) and (f) illustrate the motivation for inserting dummy vertices N (drawn as dashed rectangles). While both drawings are bend-minimum (8 bends) under the given constraints (edges leave vertices at the right side and enter them at the left side), they have a different number of edges with 4 bends. Only in (e) does the number of those edges correspond with the number of feedback arcs.

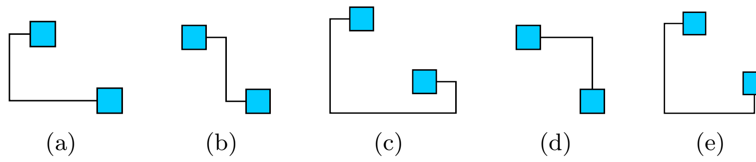


Figure 6.3: Examples of different edge routes in an `odevs` drawing with port/side constraints (we omit symmetric routes here).

if u is placed to the left of v. Hence, if G_D is acyclic, it has a topological order and thus all edges of A can be drawn with 2 bends. A drawing strategy is to place each vertex $v \in V$ on coordinate $(\pi(v), \pi(v))$ where π denotes the index of v in a topological order of G_D; see Fig. 6.2(c). For each cycle there is an edge which has 4 bends; see Fig. 6.2(d). Note that 4 bends are always sufficient since we can always take the route shown in Fig. 6.3(c) when we use the above drawing strategy. Since a minimum number of those edges corresponds to a feedback arc set A', the bend number in a bend-minimum drawing of G^* is $|A \setminus A'| \cdot 2 + |A'| \cdot 4$. Thus, in order to minimize bends we have to minimize edges having 4 bends and thus the number of feedback arcs, which is known to be NP-hard [83].

The proof also holds if we replace each side constraint by a port constraint on the same vertex side. To guarantee that all edges of E' can be drawn straight-line, the corresponding port constraints must be assigned to the κ-th pin. □

A simple but tight approximation on the bend number gives the following theorem:

Theorem 6.3 *Let $G = (V, E)$ denote the input graph. There is always a port constraint preserving* `odevs` *drawing of G with less than or equal to $3|E|$ bends (omitting self-loops).*

Proof: Let us assume that all vertices are placed arbitrarily onto a grid in general position such that there is no pair of vertices sharing the same grid line. W.l.o.g we further assume that each edge has port/side constraints on both ends. The edge routes shown in Fig. 6.3 are complete, i.e., there is always an `odevs` drawing that preserves port/side constraints and uses only those routes (or symmetric routes). More precisely, if both port/side constraints of an edge $e = (v, w)$ lie on the same side (e.g., $side(sc_e^v) = t$, $side(sc_e^w) = t$), e can always be drawn with two bends as shown in Fig. 6.3(a). If both port/side constraints lie on opposite sides (e.g., $side(sc_e^v) = t$, $side(sc_e^w) = b$), we can use the edge route of Fig. 6.3(b) or (c). For the other cases where both port/side constraints lie on adjacent sides (e.g., $side(sc_e^v) = t$, $side(sc_e^w) = l$), we can take one of the two routes shown in Fig. 6.3(d) and (e).

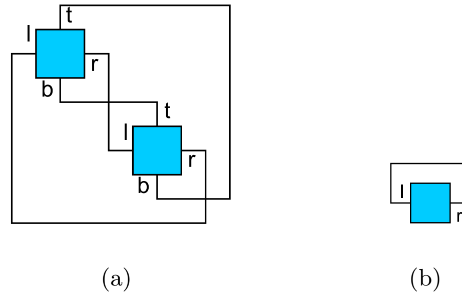


Figure 6.4: Examples of bend-minimum port constraint preserving `odevs` drawings.

Let E^* denote the edges where both port/side constraints lie on opposite sides. Since only edges of E^* can have more than three bends, it is sufficient to show that the overall number of bends of those edges is less than or equal to $3|E^*|$. Let E^*_{lr} and E^*_{tb} denote the subsets of E^* containing the edges whose ports lie on the left or right side and on the top or bottom, respectively. While the number of bends of edges of E^*_{lr} depends only on the x-coordinate of the incident vertices, the number of bends of edges of E^*_{tb} depends only on the y-coordinate. We temporarily direct edges $e \in E^*_{lr}$ from the endpoint v with $side(sc_e^v) = r$ to the endpoint w with $side(sc_e^w) = l$. We calculate a feedback arc set A' of the subgraph induced by the edges of E^*_{lr} that guarantees that $\frac{|E^*_{lr}|}{|A'|} \geq 2$ (e.g., with the heuristics described in [58]). Each topological order of the vertices of the subgraph $(V, E^*_{lr} \setminus A')$ induces an x-coordinate assignment that guarantees that all edges $E^*_{lr} \setminus A'$ can be drawn with 2 bends, as in Fig. 6.3(b). The edges of A' can be drawn with at most 4 bends (Fig. 6.3(c)). Thus, the overall number of bends for edges of E^*_{lr} is less than or equal to $3|E^*_{lr}|$. We do the same for edges of E^*_{tb} by assigning appropriate y-coordinates to the vertices. Hence, the overall number of bends for edges of E^* is less than or equal to $3|E^*|$. □

When we allow multi-edges, a simple example that shows that this bound is tight can be found in Fig. 6.4(a). Clearly, this bound increases to $4|E|$ if we also include self-loops (Fig. 6.4(b)). Edges with one or without associated port/side constraint can always be drawn in with at most two bends (using the routes shown in Fig. 6.3(d) and Fig. 6.3(a)).

If we drop the restriction that vertices are placed in general positions (different grid lines), the number of required edge routes increases, because straight-line edges may cause more complicated edge routes. Recall that due to Lemma 2.8, the maximum number of straight-line edges in an `odevs` drawing of a graph $G = (V, E)$ is bounded by $\lceil 2(|V| - \sqrt{|V|}) \rceil$ and hence we may save up to $2\lceil 2(|V| - \sqrt{|V|}) \rceil$ bends without this restriction. However,

even without it, the above bound on the bend number is still tight for the example in Fig. 6.4(a).

6.1.2 Incorporating Port/Side Constraints

The result of Theorem 6.3 provides the basis for the construction of the following approach which produces port constraint preserving `odevs` drawings with less than or equal to $3|E|$ bends (omitting self-loops) and less than or equal to 4 bends per edge. The approach is based on the three-phase method and runs in linear time. In the following, we look at the realization of its single phases. Note that a similar approach is presented in [12]. However, the author neither gives a bound on the bend number nor the runtime complexity.

6.1.2.1 Vertex Placement

First, we place the vertices on a grid in general positions, i.e., such that no two vertices share a grid row/column. Our placement strategy is based on the observation that port/side constraints imply geometric preferences on the vertex placement, e.g., if an edge $e = (v, w)$ should leave v on the bottom and enter w on the right side, it is preferable – in terms of a low bend number – to place v to the right and above of w. Then e can be routed with one bend as shown in Fig. 6.3(d). Obviously, including such preferences into the placement strategy guarantees orthogonal edge routes with few bends. During the vertex placement we consider all port constraints as side constraints; the specific port is not yet relevant. Table 6.2 shows the number of bends for different placements. More precisely, for each combination of side constraints on an edge (v, w) (first column) it states the minimum number of bends (second column) for the given condition on the coordinates of the endpoints (third column). For each bend number it also contains a reference to a figure showing the corresponding edge route.

To generate a drawing with few bends, we try to place vertices such that, for many edges, the conditions leading to the fewest bends are fulfilled. Therefore, we introduce two directed constraint graphs C_x and C_y representing the preferred relative x- and y-coordinates of the endpoints of an edge. Both constraint graphs have the same vertex set as the input graph. We assume that the lower left corner of the drawing has coordinate $(0, 0)$. Let $x(v)$ denote the x-coordinate and $y(v)$ the y-coordinate of a vertex v. An edge (a, b) in C_x (C_y) represents the constraint that a should have a smaller x-coordinate (y-coordinate) than b. Thus, we proceed as follows: for an (undirected) edge (v, w) with side constraints $side(sc_e^v) = t$ and $side(sc_e^w) = r$ we refer to Table 6.2. In the corresponding row, we see that in order to obtain a route with lowest bend number for (v, w) we have to satisfy $y(v) < y(w)$ and $x(v) > x(w)$. Hence, we insert an edge (v, w)

Side Constraints	Bends (Shape)	Condition
$side(sc_e^v) = side(sc_e^w)$	2 (Fig. 6.3(a))	-
$side(sc_e^v) = t$, $side(sc_e^w) = b$	2 (Fig. 6.3(b))	$y(v) < y(w)$
	4 (Fig. 6.3(c))	else
$side(sc_e^v) = r$, $side(sc_e^w) = l$	2 (Fig. 6.3(b))	$x(v) < x(w)$
	4 (Fig. 6.3(c))	else
$side(sc_e^v) = t$, $side(sc_e^w) = r$	1 (Fig. 6.3(d))	$y(v) < y(w) \wedge x(v) > x(w)$
	3 (Fig. 6.3(e))	else
$side(sc_e^v) = t$, $side(sc_e^w) = l$	1 (Fig. 6.3(d))	$y(v) < y(w) \wedge x(v) < x(w)$
	3 (Fig. 6.3(e))	else
$side(sc_e^v) = b$, $side(sc_e^w) = r$	1 (Fig. 6.3(d))	$y(v) > y(w) \wedge x(v) > x(w)$
	3 (Fig. 6.3(e))	else
$side(sc_e^v) = b$, $side(sc_e^w) = l$	1 (Fig. 6.3(d))	$y(v) > y(w) \wedge x(v) < x(w)$
	3 (Fig. 6.3(e))	else
$side(sc_e^v) = t$	1 (Fig. 6.3(d))	$y(v) < y(w)$
	2 (Fig. 6.3(a))	else
$side(sc_e^v) = b$	1 (Fig. 6.3(d))	$y(v) > y(w)$
	2 (Fig. 6.3(a))	else
$side(sc_e^v) = r$	1 (Fig. 6.3(d))	$x(v) < x(w)$
	2 (Fig. 6.3(a))	else
$side(sc_e^v) = l$	1 (Fig. 6.3(d))	$x(v) > x(w)$
	2 (Fig. 6.3(a))	else

Table 6.2: The table gives the minimum number of bends of edges subject to the given coordinates of their endpoints. Note that port constraints are considered as side constraints here.

into C_y as well as an edge (w, v) into C_x. Note that for edges (v, w) with side constraints $side(sc_e^v) = side(sc_e^w)$ we do not insert edges into C_x or C_y since we can always draw those edges with two bends without any placement restrictions. The same holds for edges without side constraints, which can always be drawn in with one bend.

The constraint graphs might contain cycles which can be broken using a minimum feedback arc set heuristic. The heuristic described in [58] runs in linear time and guarantees that at most half of the edges of C_x and C_y are feedback arcs. As soon as both graphs are acyclic we assign the vertices to grid coordinates. We, therefore, calculate two vertex sequences which correspond to a topological ordering π_y, π_x of the vertices of C_y and C_x, respectively. Each vertex $v \in V$ is then placed on grid coordinate $(\pi_x(v), \pi_y(v))$. For the calculation of the sequences, we have to keep in mind that not all edges have port/side constraints. We also want to produce satisfying results in cases where the number of port/side constraints is small. If we use a traditional topological sorting algorithm, vertices which are not associated with port/side constraints are placed arbitrarily since they do not induce edges in C_y and C_x.

Thus, we use a modified version of the first phase of the GT heuristic to calculate the vertex sequences. Similar to the calculation of the layering described in Section 4.2.1, this modified version operates on the input graph G and includes the constraints given by the constraint graphs. Hence, the result corresponds to a topological ordering of the vertices of C_y (C_x), which additionally incorporates the adjacencies of the vertices in G. Recall that the GT heuristic tries to place adjacent vertices of G next to each other and thus may produce better results in terms of edge length as well as the number of crossings. In the following, we describe how to calculate the horizontal (vertical) vertex sequence in linear time.

Let $\pi_x \colon V \to \mathbb{N}$ ($\pi_y \colon V \to \mathbb{N}$) denote the function that maps vertices to their positions in the horizontal (vertical) sequence. We must ensure that for each edge $(v, w) \in C_x$ (C_y), $\pi_x(v) < \pi_x(w)$ ($\pi_y(v) < \pi_y(w)$). A sequence is constructed incrementally as follows: Assume that vertex v is chosen in the k-th step. Let G_k denote the subgraph of G induced by vertices not yet chosen. In the $k+1$-th step, we choose a vertex which is adjacent to v and which has minimum degree in G_k, but is not the successor of an unchosen vertex in C_x (C_y). If this is not possible, we take a vertex of minimum degree in G_k which, additionally, is not the successor of an unchosen vertex in C_x (C_y). Since both constraint graphs are acyclic we always find such a vertex. The first vertex in the sequence is a vertex without incoming edge in C_x (C_y) and minimum degree in G. Algorithm 6 gives the corresponding pseudo-code. It is very similar to that of algorithm `calcLayering` on page 64. Since C_x contains the horizontal and C_y the vertical constraints, the vertex ordering can be interpreted as an assignment of x- and y-coordinates to each

vertex. Recall that the vertices are placed such that there is at most one vertex per x- as well as y-coordinate.

Algorithm 6: `calcCoordinates`

Input: A graph $G = (V, E)$ and the acyclic directed horizontal/vertical constraint graph $C = (V, A)$.
Output: The ordering function $\pi : V \to \mathbb{N}$.

1 $V' \leftarrow V$;
2 $G' \leftarrow G$;
3 $C' \leftarrow C$;
4 *Neighbors* $\leftarrow \emptyset$;
5 **for** $i = 1$ **to** $|V|$ **do**
6 　*Candidates* $\leftarrow \{v \in \textit{Neighbors} \mid \delta^-_{C'}(v) = 0\}$;
7 　**if** *Candidates* $= \emptyset$ **then**
8 　　*Candidates* $\leftarrow \{v \in V' \mid \delta^-_{C'}(v) = 0\}$;
9 　$\mathcal{X} \leftarrow \{v \in \textit{Candidates} \mid \delta_{G'}(v) \leq \delta_{G'}(w) \ \forall\, w \in \textit{Candidates}\}$;
10 　$v \leftarrow$ choose element of $\mathcal{X}$;
11 　$\pi(v) \leftarrow i$;
12 　*Neighbors* $\leftarrow \{w \in V' \mid w$ adjacent to v in $G\}$;
13 　$V' \leftarrow V' \setminus v$;
14 　$G' \leftarrow$ subgraph of G induced by V';
15 　$C' \leftarrow$ subgraph of C induced by V';
16 **return** π;

The overall runtime of the vertex placement phase is as follows: For each edge $e \in E$, we perform a table lookup to determine the corresponding constraint edges in C_x or C_y. Since the table has a constant number of entries, this can be done in time $O(|E|)$. We insert at most 2 constraints per edge and use a linear-time feedback arc set heuristic to remove cycles in the constraint graphs. Thus, it remains to show that algorithm `calcCoordinates` can be implemented in linear time. Assume that vertex list *Neighbors* in line 6 is non-empty. Then, the time complexity of choosing a vertex $v \in \textit{Neighbors}$ of minimal degree and $\delta^-_{C'}(v) = 0$ over all iterations of the for-loop is $O(|V|+|E|)$. Hence, it is sufficient to show that inside the for-loop, we can determine a vertex v of minimal degree in G' and $\delta^-_{C'}(v) = 0$ in constant time (line 8). Therefore, we use an array `vertex_degree` of vertex lists. A vertex v is contained in the vertex list at index i of `vertex_degree` if it has degree i in G' and zero degree in C'. Note that the corresponding list cell is stored by v. We choose the size of the array to be constant, say $c+1$. All vertices $v \in V$ with $\delta_{G'}(v) \geq c$ are contained in the vertex list at index c. Thus, finding a vertex of minimum degree in G' and zero degree in C' can be done in constant time by searching for the first non-empty vertex list

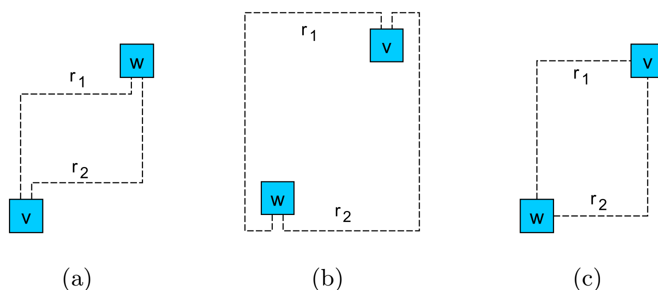


Figure 6.5: Different edge routes appearing in the `odevs` approach. In (a) and (b) the edge should leave v at the top and enter w at the bottom. In (c) there are no restrictions on the edge route.

in `vertex_degree`. If the degree of the chosen vertex is larger than or equal to c, it is not guaranteed that it is a vertex of minimum degree. However, as experiments show, this has only insignificant effects on the quality of the result if $c \geq 10$. Determining the subgraphs G' and C' in lines 14 and 15 simply consists of removing the current vertex v and its incident edges from G' and C'. For all previous neighbors w of v in G' with $\delta_{C'}(w) = 0$ we have to move w to another list in `vertex_degree` since we decreased w's degree. Furthermore, we have to remove v from `vertex_degree`. For a single vertex, both can be done in constant time since each vertex stores a reference to its list cell in the current vertex list. Vertices whose degree in C' becomes 0 after removing v from C' have to be inserted into `vertex_degree`. Obviously, the overall runtime of these steps is bounded by $O(|V| + |E|)$.

6.1.2.2 Edge Routing

We first determine the route for all edges $E' \subseteq E$ with at least one port/side constraint. The route depends on the vertex placement and can be determined by referring to Table 6.2. Note that port constraints are still handled like side constraints during this phase. For edges with two port/side constraints lying on opposite sides there are two different valid routes. Let $e = (v, w)$ denote an edge with side constraints $sc_e^v = t$ and $sc_e^w = b$. Fig. 6.5(a) shows the valid routes of e when w is placed above v. We always choose the route where the middle segment is placed close to (and above) the vertex v with $sc_e^v = t$ (route r_2 for the example in Fig. 6.5(a)). When w is placed below v as shown in Fig. 6.5(b) we always place the middle segment close to the vertex v with $sc_e^v = t$ (either to the left- or right-hand side, depending on the relative horizontal positions of v and w). Thus, for the example in Fig. 6.5(b) we choose route r_2. Analogously, if e has side constraints $sc_e^v = r$ and $sc_e^w = l$, side r is handled like side t and side l like side b.

We continue iteratively determining the route for edges without port/side constraints. For each such edge $e = (v, w) \in E \setminus E'$ there are two valid routes with one bend (Fig. 6.5(c)). Both routes are incident to different vertex sides. Let $\delta^s_{r_1}(v)$, $\delta^s_{r_1}(w)$ denote the number of already handled edges which are incident to the side where edge route r_1 attaches v and w, respectively. Analogously, we define $\delta^s_{r_2}(v)$, $\delta^s_{r_2}(w)$ for the second route. If $\delta^s_{r_1}(v) + \delta^s_{r_1}(w) > \delta^s_{r_2}(v) + \delta^s_{r_2}(w)$ we choose route r_2 and otherwise route r_1. This strategy can be implemented in linear time and leads to a more balanced distribution of the edges on the vertex sides.

To guarantee that there are enough fine grid lines for routing edges, we have to insert coarse grid lines as follows: Let v and w denote a pair of vertices which are placed on consecutive coarse grid lines with respect to the vertical direction. Let ξ denote the number of required horizontal fine grid lines lying between both vertices. Due to our vertex placement and the set of valid edge routes, only edges incident to v or w may require such fine grid lines. Hence, we can determine ξ $(0 \leq \xi \leq \delta_G(v) + \delta_G(w))$ by looking at the routes of the corresponding edges. We have to insert $\lceil \frac{\xi}{2\kappa - 1} \rceil$ horizontal coarse grid lines between v and w. Analogously, we have to insert additional vertical coarse grid lines between vertices placed on consecutive coarse grid lines with respect to the horizontal direction.

In Theorem 6.3 we showed that there is always a port constraint preserving drawing with less than or equal to $3|E|$ bends. We, therefore, only considered port/side constraints lying on opposite sides. To improve practical results, we also include other constraints in our algorithm. In the following lemma we show that we still maintain the same bound for the overall number of bends.

Lemma 6.4 *For an input graph $G = (V, E)$ the generated drawing has less than or equal to $3|E|$ bends and at most 4 bends per edge.*

Proof: Let E_1, E_2, E_3 and E_4 denote the sets of edges with one, two, three and four bends. Since there are neither edges with zero bends nor with more than four bends, we have $E = E_1 \cup E_2 \cup E_3 \cup E_4$. We want to show that $|E_1| + 2|E_2| + 3|E_3| + 4|E_4| \leq 3(|E_1| + |E_2| + |E_3| + |E_4|)$, which is equivalent to $|E_4| \leq 2|E_1| + |E_2|$.

For each edge of $E_1 \cup E_3$, we inserted at most two constraints into the constraint graphs and for each edge of $E_2 \cup E_4$ at most one constraint (Table 6.2). Hence, the overall number of inserted constraints is less than or equal to $2|E_1| + |E_2| + 2|E_3| + |E_4|$. Let c^+ denote the number of satisfied constraints and c^- the number of unsatisfied constraints. The applied feedback arc set heuristic guarantees that $c^+ \geq c^-$. Furthermore, we have $c^- \geq |E_3| + |E_4|$ since each edge of $E_3 \cup E_4$ can be associated with at least one unsatisfied constraint. Hence, the number of satisfied constraints is $c^+ \leq 2|E_1| + |E_2| + |E_3|$. It follows that $2|E_1| + |E_2| \geq |E_4|$ and thus, the overall number of bends is less than or equal to $3|E|$. □

Type	Constraints	Order
a	$y(v) > y(w) \wedge x(v) < x(w) \wedge rside(e,w) = b, l$	decreasing y
b	$y(v) > y(w) \wedge x(v) > x(w) \wedge rside(e,w) = t, l$	decreasing x
c	$y(v) < y(w) \wedge x(v) > x(w) \wedge rside(e,w) = b$	increasing x
d	$y(v) < y(w) \wedge x(v) > x(w) \wedge rside(e,w) = t, r, l$	increasing y
e	$y(v) < y(w) \wedge x(v) < x(w) \wedge rside(e,w) = t, r, l$	decreasing y
f	$y(v) < y(w) \wedge x(v) < x(w) \wedge rside(e,w) = b$	increasing x
g	$y(v) > y(w) \wedge x(v) < x(w) \wedge rside(e,w) = t, r$	decreasing x
h	$y(v) > y(w) \wedge x(v) > x(w) \wedge rside(e,w) = b, r$	increasing y

Table 6.3: Ordering of edges $e = (v, w)$ on the top of a vertex v (from left to right).

6.1.2.3 Port Assignment

For each side $s \in dir$ of a vertex $v \in V$, we have to assign the corresponding edges to pins (i.e., fine grid lines) of v. Let $E_{v,s}$ denote the set of edges assigned to side s of vertex v, $E_{v,s}^{pc} \subseteq E_{v,s}$ the subset of edges with port constraints on v and $E_{v,s}^{*} = E_{v,s} \setminus E_{v,s}^{pc}$ the subset of remaining edges. For edges of $E_{v,s}^{pc}$, the pins and thus the ordering is given. For edges of $E_{v,s}^{*}$, we want to find an assignment to pins such that the number of crossings among edges of $E_{v,s}$ is small (for a given side $s \in dir$). Therefore, we first determine a suitable order for the edges of $E_{v,s}^{*}$ and then merge these edges with that of $E_{v,s}^{pc}$.

Let $rside \colon (E, V) \to dir$ denote the function that returns the side $s \in dir$, where the route of an edge $e \in E$ enters vertex $v \in V$. Tables 6.3 and 6.4 give a suitable sorting order (from left to right) for edges on the top and bottom, respectively. For an edge $e = (v, w)$ on a vertex v, we first determine the type of e which depends on the side $rside(e, w)$ where the route of e enters w as well as the relative positions of v and w. Note that the number of different types in both tables is not the same because of the different routes chosen for the middle segment of edges (Fig. 6.5(a),(b)). Edges $e \in E_{v,s}^{*}$, $s \in \{t, b\}$ of different types are placed in ascending alphanumerical order, according to their type from left to right, i.e., all edges of type b are placed to the left of those of type f. Edges of the same type are ordered as specified in the third column of the tables according to the x- or y-coordinate of w, either in decreasing or increasing order. Edges $e \in E_{v,s}^{*}$, $s \in \{r, l\}$ on the right or left sides of a vertex are sorted analogously. As can be seen in Fig. 6.6, for a given side s, our ordering strategy can prevent crossings between edges of $E_{v,s}^{*}$ in most cases.

Let $L_{v,t}^{pc}$ denote the sorted list of edges of $E_{v,t}^{pc}$ and $L_{v,t}^{*}$ the sorted list of edges of $E_{v,t}^{*}$. To assign edges of $E_{v,t}^{*}$ to pins, we merge both lists using

Type	Constraints	Order
A	$y(v) < y(w) \wedge x(v) < x(w) \wedge rside(e,w) = l$	increasing y
B	$y(v) < y(w) \wedge x(v) > x(w) \wedge rside(e,w) = t,b,l$	decreasing x
C	$y(v) > y(w) \wedge x(v) > x(w) \wedge rside(e,w) = t,b,r,l$	decreasing y
D	$y(v) > y(w) \wedge x(v) < x(w) \wedge rside(e,w) = t,b,r,l$	increasing y
E	$y(v) < y(w) \wedge x(v) < x(w) \wedge rside(e,w) = t,b,r$	decreasing x
F	$y(v) < y(w) \wedge x(v) > x(w) \wedge rside(e,w) = r$	decreasing y

Table 6.4: Ordering of edges $e = (v,w)$ on the bottom of a vertex v (from left to right).

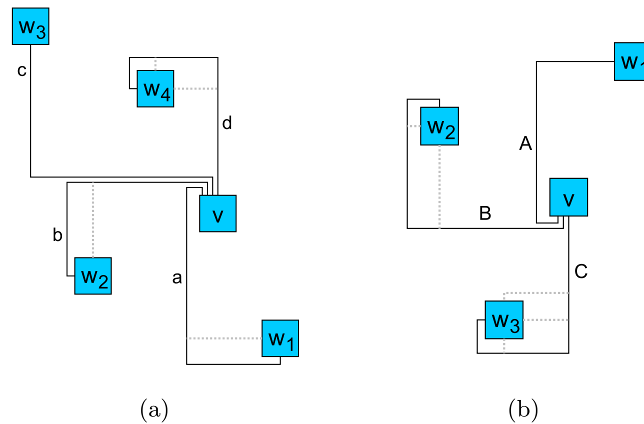


Figure 6.6: Examples of ordering edges on the top/bottom of vertices. In both examples the order of edges on v yields a crossing-free drawing. The labels of edges correspond to the different types given by Table 6.3 and 6.4. Dotted gray edges denote alternative routes from vertices w_i to v.

a variant of the "first-fit" heuristic described in [12]. Our pin assignment satisfies the following two properties:

1. For both lists, the relative order of elements remains unchanged.
2. The assignment is valid regarding our port/side constraint model, i.e., the order of edges satisfies the points given in Definition 3.7 on page 46.

Let $maxPin_v \colon E^*_{v,t} \to \{1, \ldots, 2\kappa - 1\}$ denote the function that assigns the largest possible pin number to edges of $E^*_{v,t}$ such that the above properties are still satisfied. Such a "rightmost" assignment can be calculated in $O(|E^{pc}_{v,t}| + |E^*_{v,t}|)$ time. We try to obtain a more suitable pin assignment by using algorithm `calcPins`. Note that it is only applied if both edge sets are non-empty since otherwise the pin assignment is trivial. The ordering relation given by Table 6.3 is denoted $<^t$. Furthermore, the list function *pop* removes and returns the first element of a list. The result of our algorithm is a function $location'_v \colon E^*_{v,t} \to \{1, \ldots, 2\kappa - 1\}$ giving the pin number for edges of $E^*_{v,t}$.

Algorithm 7: `calcPins`

Input: A vertex $v \in V$, the function $maxPin_v$, the ordering relation $<^t$ as well as the sorted lists $L^{pc}_{v,t}$ and $L^*_{v,t}$.
Output: The function $location'_v \colon E^*_{v,t} \to \{1, \ldots, 2\kappa - 1\}$.

```
1  e' ← pop(L^pc_{v,t});
2  pin ← 0;
3  empty ← false;
4  while L*_{v,t} ≠ ∅ do
5      e ← pop(L*_{v,t});
6      if not empty then
7          pin' ← location(pc^v_{e'});
8          while (e' <^t e ∨ pin' ≤ pin) ∧ (pin' < maxPin_v(e)) do
9              pin ← pin' + 1;
10             if L^pc_{v,t} ≠ ∅ then
11                 e' ← pop(L^pc_{v,t});
12                 pin' ← location(pc^v_{e'});
13             else
14                 empty ← true;
15     location'_v(e) ← pin;
16     pin ← pin + 1;
17 return location'_v;
```

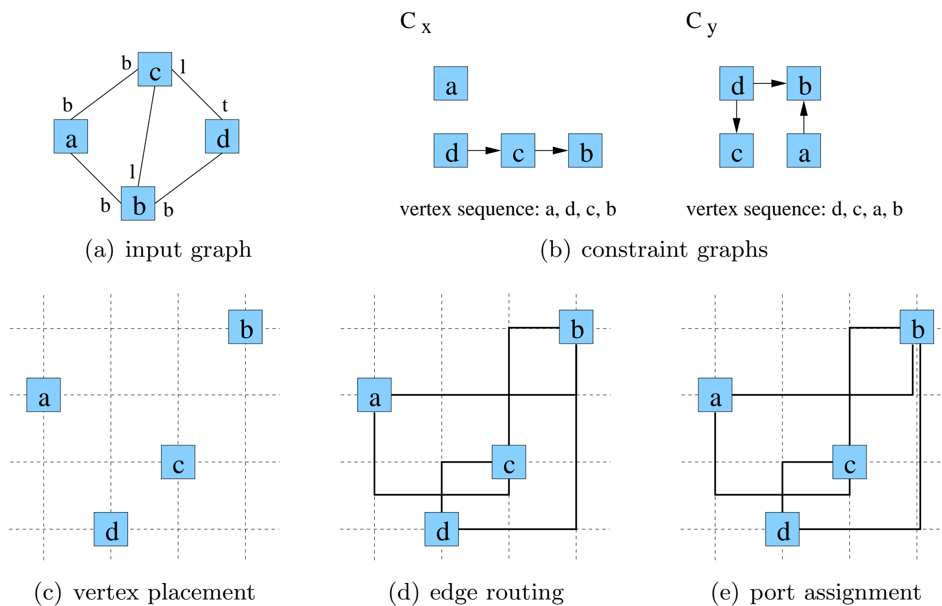
The algorithm iteratively removes the first edge e of $L^*_{v,t}$ and searches for the first edge g of $L^{pc}_{v,t}$ with $e <^t g$. Then it tries to assign e to a pin between

g and its successor in $L^{pc}_{v,t}$. Note that the expression $pin' < maxPin_v(e)$ in line 8 guarantees that $location'_v(e) \leq maxPin_v(e)$. Furthermore, the expression $pin' \leq pin$ handles the following special case: let f, g denote two consecutive edges in list $L^{pc}_{v,t}$ with $f <^t e$ and $e <^t g$. If there is no free pin between edge f and g, we cannot place e in between and, hence, simply continue. Since the pins assigned to the edges of $L^*_{v,t}$ increase strictly monotonically we also satisfy the first property stated above.

The outer while-loop is iterated once for each edge of $E^*_{v,t}$ and the inner while-loop once for each edge of $E^{pc}_{v,t}$. Thus the algorithm has linear runtime. Pins of edges attached to the other sides are determined analogously. The final coordinates of bends of edges can be derived from the pins assigned to these edges.

In order to maintain a linear overall runtime, we also have to sort the edge sets $E^{pc}_{v,s}$ and $E^*_{v,s}$ in linear time. For a given vertex $v \in V$ and side $s \in dir$, we can sort the edges of $E^{pc}_{v,s}$ using bucket sort [37] with $2\kappa - 1$ buckets (each bucket represents a different pin). We cannot maintain a linear runtime if we calculate the order of edges for each vertex v separately. Hence, we sort the edges for all vertices simultaneously. Note that each single edge $e = (v, w)$ is assigned to two edge lists, one associated with v and one associated with w. Thus, for each edge $e = (v, w)$ we obtain two sorting elements $\langle v, e\rangle$ and $\langle w, e\rangle$ representing the position of e in an edge list of v and w, respectively. When we sort the sets $E^{pc}_{v,s}$, we only consider those elements $\langle v, e\rangle$ for which there is a port constraint pc^v_e. We first sort those elements according to their sides, i.e., for each side s we determine all elements $\langle v, e\rangle$ with $rside(e, v) = s$. Then for each side $s \in dir$, we sort the corresponding elements $\langle v, e\rangle$ according to their associated pins ($location(pc^v_e)$) in ascending order (using $2\kappa - 1$ buckets). Now, for each side s, we traverse the sorted elements from left to right. For each element $\langle v, e\rangle$ we assign edge e to list $L^{pc}_{v,s}$. Since the number of sides is constant and the number of pins on a vertex side is $O(|E|)$, the overall runtime for generating these lists is linear.

Edges of $E^*_{v,s}$ can be sorted similarly. Now we only consider elements $\langle v, e\rangle$ if there is no port constraint pc^v_e. Again, we first sort the elements according to their sides. Then we sort all elements of the same side according to their type in ascending alphanumerical order. All elements $\langle v, e = (v, w)\rangle$ of the same side and type are again sorted according to the x-/y-coordinates of w in decreasing/increasing order, dependent on the type. Recall that the number of different x-/y-coordinates of vertices is $|V|$ in our approach and, hence, we require $|V|$ buckets. Since the number of sides and types is constant the overall runtime for sorting the elements is still linear. Now we use the same strategy as above to assign edges to lists $L^*_{v,s}$. Note that due to the high constant factor, for smaller graphs it might be more suitable to use a common sorting strategy.



(a) input graph (b) constraint graphs

(c) vertex placement (d) edge routing (e) port assignment

Figure 6.7: The different phases of the `odevs` approach for a graph with side constraints.

Fig. 6.7 gives an example of applying the `odevs` drawing approach to an input graph. Since each step has linear runtime and due to Lemma 6.4, we can conclude this section with the following theorem:

Theorem 6.5 *For an input graph $G = (V, E)$ without self-loops, our approach calculates an* `odevs` *drawing with less than or equal to $3|E|$ bends and at most 4 bends per edge in linear time.*

6.2 Alternative Planarization Approaches

In this section, we state alternative approaches for including port/side constraints into the planarization phase. More precisely, we describe different heuristics for calculating maximum planar subgraphs including port/side constraints. Therefore, we focus especially on approved approaches used for graphs without constraints. After the calculation of a planar subgraph, we can obtain a port constraint preserving planarization of the input graph by using a modified shortest path routing, which maintains a valid edge order around the vertices.

6.2.1 Successive Planarity Testing

A planar subgraph $G' = (V, E')$ of a graph $G = (V, E)$ is called *maximal planar subgraph* if there is no edge $e \in E \setminus E'$ which can be added to G' with-

out losing planarity. A maximal planar subgraph of G can be constructed by starting with the empty graph $(V, \emptyset)$ and then successively testing if the reinsertion of an edge $e \in E$ would still leave the graph planar. If so, e is reinserted; otherwise it is discarded. Unfortunately, up to know there has been no general planarity test which includes port/side constraints. For the `pc` and `sc` scenarios, planarity can be tested in linear time with the approach of [92] described in Section 3.1.5. This results in a quadratic overall runtime.

6.2.2 Spanning Tree-Based Planarization

Obviously, each tree can be drawn planar because it does not contain a cycle and hence no subgraph that is a subdivision of K_5 or $K_{3,3}$. Thus, each spanning tree of a graph $G = (V, E)$ yields a planar subgraph with $|V| - 1$ edges. An ordered tree (with fixed order of the children) can also be drawn planar, since the embeddings of the subtrees rooted at any vertex $v \in V$ are independent of each other and thus the cyclic order around v can be chosen arbitrarily. It follows that each spanning tree of G is a planar subgraph which preserves the given port/side constraints. A spanning tree can be calculated in linear time using, e.g., a breadth-first search [37]. To obtain a planar embedding we have to determine the cyclic order of the edges around each vertex $v \in V$. The ordering has to be consistent with the given port/side constraints. While it is fixed for edges with port constraints on v, the remaining edges can be arranged arbitrarily taking into account the restrictions given by side constraints and occupied pins. As a basic rule for the arrangement, edges should be equally distributed around the vertices such that we still maintain a high degree of freedom for routing the remaining edges. If, for example, there are only a few free pins between two edges with port constraints on v, there should be no edge placed in between. A weakness of this approach is, that the resulting planar subgraph always contains only $|V| - 1$ edges. Thus, a lot of edges have to be routed through the extended dual graph, which often results in a higher number of edge crossings. For the `mc` scenario it is preferable to have many edges with port/side constraints in the planar subgraph. While the number of its edges is then still $|V| - 1$, we obtain a higher degree of freedom during the routing of the remaining edges. We realize this by calculating a minimum spanning tree (e.g., with Prim's algorithm [37]), where the weight of an edge e depends on the number of port/side constraints associated with e. Hence, the overall runtime of this approach is $O(|V| \log |V| + |E|)$.

6.2.3 GT-Based Planarization

The following calculation of the planar subgraph is based on the GT heuristic described in Section 2.3.1. Since the heuristic does not incorporate port/side constraints, we have to extend it accordingly.

We first look at the `sc` scenario, which allows a realization that is close to the original heuristic. More precisely, we reduce the problem of finding a planar subgraph which observes side constraints to finding one without constraints. After having calculated the vertex sequence Π_V in the same way as for the original heuristic, we split each vertex v into three vertices v_b, v_t and $v_{l,r}$. Vertex v_b represents the bottom and v_t the top of v. Furthermore, when we calculate $\mathcal{L}$ (the set of edges placed to the left of the line), the vertex $v_{l,r}$ denotes the left side and when we calculate $\mathcal{R}$ (the set of edges placed to the right of the line) it denotes the right side of v. Let V^* denote the resulting set of vertices. For the new vertex sequence Π_{V^*} we have $\pi^*(v_{l,r}) = 3 \cdot \pi(v)$, $\pi^*(v_b) = 3 \cdot \pi(v) - 1$ and $\pi^*(v_t) = 3 \cdot \pi(v) + 1$. Obviously, π^* has the property that $\pi^*(v_x) < \pi^*(w_y)$, $x, y \in \{b, t, \{l, r\}\}$ if $\pi(v) < \pi(w)$. Furthermore, we have $\pi^*(v_b) < \pi^*(v_{l,r}) < \pi^*(v_t)$. Each edge is reconnected according to its side constraints, e.g., an edge $e = (v, w)$ with side constraints sc_e^v, $side(sc_e^v) = l$ and sc_e^w, $side(sc_e^w) = t$ changes to $(v_{l,r}, w_t)$. Recall that in the `sc` scenario each edge has side constraints on both ends. During the calculation of the edge sets $\mathcal{L}$ and $\mathcal{R}$ in the second phase, two edges cross each other if they cross with respect to π^*. When we calculate $\mathcal{L}$ we only consider edges $e = (v, w)$ with $side(sc_e^v) \neq r \ \wedge \ side(sc_e^w) \neq r$ and when we calculate $\mathcal{R}$ only edges $e = (v, w)$ with $side(sc_e^v) \neq l \ \wedge \ side(sc_e^w) \neq l$. After calculating both sets, the cyclic ordering of the edges around a vertex $v \in V$ can be obtained by contracting the vertices v_b, $v_{l,r}$ and v_t as shown in Fig. 6.8(c).

For the `sc` scenario, the above approach is able to calculate a planar subgraph which observes side constraints. It has the same runtime as the GT heuristic, $O(|V||E|^2)$, since the number of vertices in the extended graph is $3|V|$ and the number of edges is $|E|$. A weakness of that approach is that the planar subgraph never contains edges with one side constraint on the left and the other on the right. Note that when the number of those edges is large, we can alternatively apply the heuristic to the same graph where the sides assigned to edges with side constraints are temporarily rotated by 90° (i.e., top becomes right, right becomes bottom etc.). It is not possible to efficiently adopt this approach for port constraints.

6.3 Alternative Orthogonalization Approaches

The following alternative orthogonalization approaches produce port constraint preserving drawings in the `podevsnef` model for given port constraint preserving embeddings of graphs.

6.3.1 An Integer Linear Program Formulation

The ILP formulation given in [66] is already able to handle port/side constraints in the `podevsnef` model and to determine a bend-minimum solution.

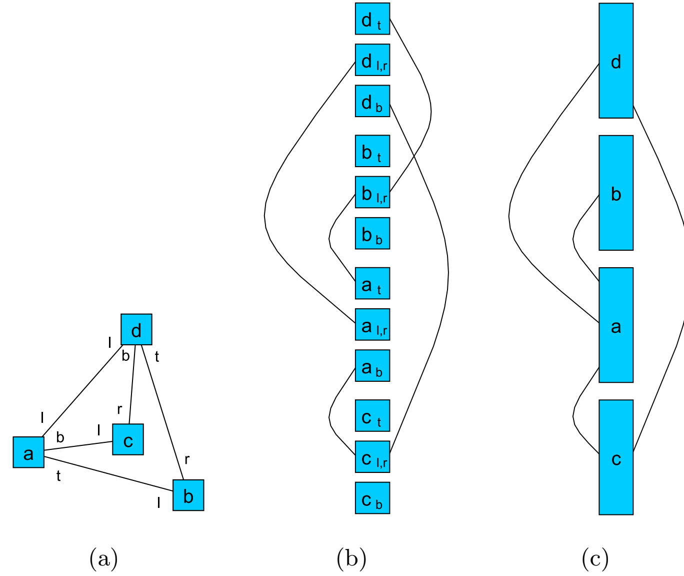


Figure 6.8: The modified GT heuristic. (a) shows the input graph and (b) the resulting graph after splitting the vertices and reconnecting the edges. After applying the heuristic and contracting the vertices, we obtain the planar subgraph shown in (c).

It is restricted to planar graphs, but the extension to handle vertices which represent crossings inserted during the planarization phase is straightforward. The ILP contains variables that denote angles between consecutive edges. Thus, for each vertex c representing a crossing, we add further constraints to ensure that all angles around c are fixed to 90°. Recall that the degree of such vertices is always equal to 4.

The existing ILP does not guarantee that straight-line edges can always be assigned to the κ-th (center) pin of the corresponding vertex sides. We can maintain this issue by putting additional constraints on the ILP: The ILP contains variables $s^{dir}(v,w) \in \{0,1\}$ with $s^{dir}(v,w) = 1$ if and only if edge (v,w) leaves vertex v at side dir ($dir \in \{t,b,l,r\}$). Furthermore, it contains variables $lb^v(v,w)$, $rb^v(v,w) \in \{0,1\}$ which represent the vertex-bends of edge (v,w) at vertex v. Vertex v has a left vertex-bend if and only if $lb^v(v,w) = 1$ and a right vertex-bend if and only if $rb^v(v,w) = 1$. Note that $lb^v(v,w) + rb^v(v,w) \leq 1$. Thus, if an edge $e = (v,w)$ cannot be assigned to the κ-th pin at side dir of vertex v, we add the constraint $s^{dir}(e) \leq lb^v(e) + rb^v(e)$. Fig. 6.9 shows a special case which has to be considered. Assume that edge e has a port constraint pc_e^v with $side(pc_e^v) = t$, $location(pc_e^v) = 4$ and edge f has no constraints. Edge f can only be centered at the top if it is placed to the right-hand side of e. Hence, for edge f we cannot determine in advance if it can be drawn straight-line.

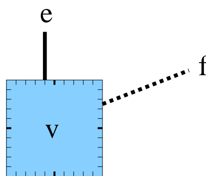


Figure 6.9: A special case occurring during the orthogonalization with port constraints. Edge e has a port constraint pc_e^v with $side(pc_e^v) = t$, $location(pc_e^v) = 4$. Edge f can only be centered at the top if it is placed to the right-hand side of e.

Thus, for each edge $e = (v, w)$ with port constraint pc_e^v we set $lb^v(e) = 1$ if $location(pc_e^v) < \kappa$ and $rb^v(e) = 1$ if $location(pc_e^v) > \kappa$. This is sufficient, because the ILP formulation guarantees a consistent bend assignment of the remaining edges.

6.3.2 Orthogonalization Without Fixing a Skeleton

The following approach is a variant of the network flow-based orthogonalization with port/side constraints described in Section 5.4. Recall that, in order to get a uniform orientation of the vertices, we fixed the shape of so-called skeleton edges there. In general, finding such skeleton edges together with a valid shape is a difficult task since the shape has to be consistent with the given planar embedding and the given port/side constraints. It is advisable to already consider this issue during the planarization phase. The planarization approaches described in Section 6.2 do not incorporate the calculation of an appropriate skeleton and thus cannot be combined with our primary orthogonalization. Hence, we present below an alternative orthogonalization approach. Its only difference to the approach of Section 5.4 is that it does not fix the shape of skeleton edges. Thus, after calculating the shape of edges, we additionally have to unify the orientation of vertices. Therefore, we rotate vertices which do not conform to a prescribed direction by 90° to the left/right or by 180°. As shown in Theorem 6.6, this produces at most $2|E|$ additional bends. After rotating vertices, we apply bend-stretching transformations [46] to reduce the number of unnecessary bends.

Theorem 6.6 *Let $G = (V, E)$ denote a graph and Γ a* `podevsnef` *drawing of G without a uniform orientation of the vertices. The orientation can then be unified by introducing at most $2|E|$ bends.*

Proof: Γ can be transformed into a drawing with a uniform orientation of the vertices by rotating vertices which do not conform to a prescribed direction by 90° to the left/right or by 180°. As illustrated in Fig. 6.10, a

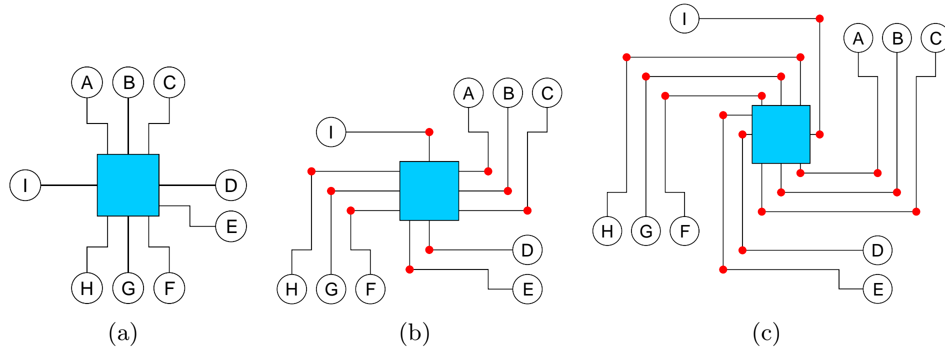


Figure 6.10: A vertex and its incident edges (a). A rotation of the vertex by 90° to the right adds one additional bend per edge (drawn as red point) (b) and a rotation by 180° two additional bends (c).

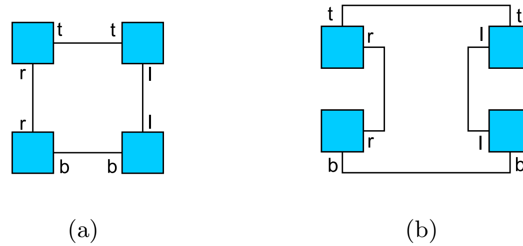


Figure 6.11: A bend-minimum `podevsnef` drawing which preserves the angles induced by the given side constraints (a) as well as a bend minimum drawing with a uniform orientation of the vertices (b).

rotation of a vertex by 90° introduces one additional bend at its incident edges and a rotation by 180° introduces two additional bends. Clearly, the resulting drawing is still a `podevsnef` drawing because the rotations neither create crossings nor empty faces. Note that some of the additional bends are superfluous and can be removed by bend-stretching transformations. For a given orientation of the drawing let V_0, V_{90} and V_{180} denote the set of vertices which need not be rotated, rotated by 90° and rotated by 180°, respectively. Note that $V = V_0 \cup V_{90} \cup V_{180}$. We chose the orientation of the drawing such that $\sum_{v \in V_0} \delta_G(v) > \sum_{v \in V_{180}} \delta_G(v)$. Thus, the overall number of bends introduced by rotating vertices is $\sum_{v \in V_0} \delta_G(v) \cdot 0 + \sum_{v \in V_{90}} \delta_G(v) + \sum_{v \in V_{180}} \delta_G(v) \cdot 2 \leq \sum_{v \in V} \delta_G(v) = 2|E|$. □

Fig. 6.11 shows that this bound is tight: While there is a drawing without bends when we omit the uniform orientation of the vertices, the bend-minimum drawing observing the orientation has $2|E|$ bends. Note that the above strategy does not produce a bend-minimum solution in most cases. The rotation step can be implemented in linear time.

6.4 Additional Requirements

Below, we shortly sketch different additional requirements that are necessary in order to apply port/side constraints to a wider field of applications. These requirements can serve as motivation for future work in this area.

- **Multi-Candidates Port/Side Constraints:** Recall that, in general, pins determine the possible attachment points of edges on their incident vertices. In practice, there often appear side constraints like "edge e should leave its source vertex either at the right- or left-hand side" (e.g., in UML class diagrams). Moreover, there are often edges which should end at an arbitrary pin out of a given set of pins. We call such port/side constraints *multi-candidates port/side constraints*. The increasing degree of freedom of choosing a valid pin/side where an edge enters/leaves a vertex may further improve the quality of a drawing. It is also possible to assign different weights to the different candidates.

- **Limited Number of Pins:** The second requirement concerns the arrangement of pins as well as their number. Up to now we have assumed that each side of a vertex has $2\kappa - 1$ pins with $\kappa \geq \max_{v \in V} \delta(v)$. For vertices of different size this is often inappropriate. In practice it might be more suitable to allow an irregular arrangement of pins, e.g., for wiring diagrams where electric components like multiplexer and integrated circuits have prescribed connection points. In such a model the only restriction regarding the number of pins on a vertex v is that it has to be greater than or equal to $\delta(v)$. The distribution and arrangement of pins on the vertex sides are arbitrary. Note that such an extension makes things more complicated even for edges without constraints since we are no longer able to place these edges at an arbitrary vertex side.

- **Pin Sharing:** Another useful requirement is to allow multiple edges to be attached to the same pin. This is essential when we want to visualize wiring diagrams. Furthermore, for some applications it might be useful to allow the specification of an upper bound on the number of edges that can be attached to a given pin.

Chapter 7
Usability Study

In this chapter we investigate the usability and performance of the methods and algorithms presented in the previous chapters. First, we demonstrate how to use algorithm `Constraint-Kandinsky` to draw UML activity diagrams. We identify requirements for such diagrams and give an overview of related work followed by a brief evaluation of layout capabilities for different popular UML tools. We also provide some example drawings to demonstrate the quality of our layout approach. Our work on drawing activity diagrams was published in [136, 137].

Section 7.2 is based on joint work with Wolfgang Blochinger, which is described in [14, 15]. It demonstrates how to visualize execution graphs of parallel computations using our `Fast-Sugiyama` implementation. Again we identify the corresponding requirements and review related work done in this area. We introduce specific modifications needed for visualizing execution graphs and describe the integration of the visualization into an integrated development environment. Furthermore, we demonstrate our visualization methodology using a representative example application.

In Section 7.3, we give an experimental evaluation of the runtime and quality of different algorithms presented in this work. We perform experiments for our `Fast-Sugiyama` implementation, the `Constraint-Kandinsky` algorithm as well as for the different approaches for handling port/side constraints discussed in the previous chapter.

For the implementation of our algorithms, we used the programming language Java and the yFiles library [156]. The Java-based yFiles library features a flexible and extensible architecture and provides basic graph-based data structures, graph algorithms and view components. The library also supports several graph formats.

7.1 Visualization of Activity Diagrams

Activities and the corresponding activity diagrams (Fig. 7.1) belong to the basic concepts of the Unified Modeling Language (UML) which has become the standard modeling language for specifying, visualizing and documenting software systems. Activity diagrams are used for modeling behavioral logic

like business processes or workflow. The current version 2.2 of the UML was released in February 2009 by the Object Management Group (OMG). While activity diagrams were considered a special case of state diagrams in UML version 1.x, they are now enriched with additional constructs that widen the range of their applicability. This makes them more suitable for areas like economics or bio-informatics [134].

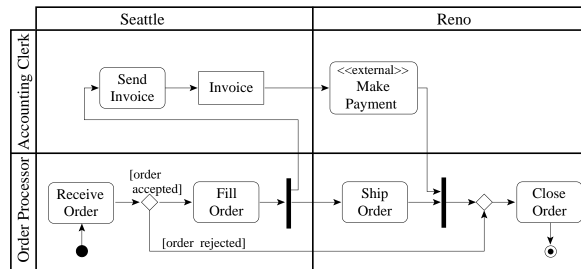


Figure 7.1: An activity diagram taken from the OMG UML 2.2 Superstructure specification [118].

While automatic layout for class diagrams received considerable attention by developers of corresponding software tools and designers of fundamental concepts of diagramming, activity diagrams were considered to be either just a special case for tools supporting state diagrams or too complex to handle them by a direct application of basic layout algorithms.

Since activity diagrams require properties that are different from those for class diagrams, existing concepts cannot be simply transferred. For activity diagrams the emphasis lies on partitions and flow, which are properties that are influenced by the semantics of activity diagrams. These properties have to be supported by an automatic layout algorithm because it is difficult, or sometimes even impossible, to place all elements appropriately by hand. Furthermore, an automatic layout algorithm with customizable layout options allows us to include different user preferences.

7.1.1 Layout Requirements

In this section we examine requirements for the automatic layout of activity diagrams. We, therefore, briefly introduce the notation elements. This is necessary to derive a graph theoretic concept to layout those diagrams. We also analyze aesthetics and standards for creating activity diagrams.

7.1.1.1 Notation of UML Activity Diagrams

In the following we give an overview of the visual notation of activity diagrams and identify the resulting requirements for a layout algorithm. We

do not focus on semantics unless necessary. More details can be found in the UML Superstructure specification [118].

An *activity* specifies the coordination of executions of actions using a control and data flow model. Each *activity diagram* shows exactly one activity. An activity is modeled as a labeled graph of *activity nodes* which are connected by edges denoting data or control flow. It can optionally be represented by a border rectangle containing all its elements and a name shown in the upper left corner. There are three different node types appearing in an activity - action, object and control nodes.

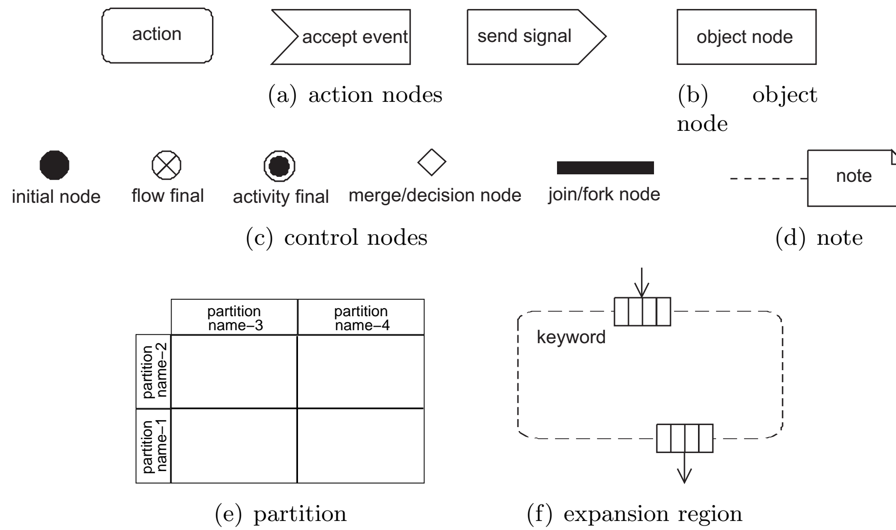
Action nodes (Fig. 7.2(a)) are the basic elements of an activity. An action node may have outgoing and incoming edges denoting control or data flow to or from other nodes. There are two special kinds of action nodes to handle signal events, the *accept event* and the *send signal* action. *Object nodes* indicate an instance of a particular classifier (Fig. 7.2(b)).

A *control node* (Fig. 7.2(c)) coordinates the flow between other nodes. If an activity is invoked, a flow starts at each *initial node.* If a flow reaches an *activity final node*, the activity terminates. In contrast, a *flow final node* terminates only its incoming flow. A *decision node* has one incoming edge and multiple outgoing edges. The incoming flow is forwarded to only one outgoing edge. This edge is commonly determined by *guards*, which are edge labels representing conditions. A *merge node* has one outgoing edge and multiple incoming edges. Each flow arriving at an incoming edge is forwarded to the outgoing edge. A *fork node* is similar to a decision node, but it splits the incoming flow into multiple concurrent flows. A *join node* is similar to a merge node, but synchronizes multiple flows. Edges are always connected to the long side of the fork/join node. The nodes can be placed vertically or horizontally.

Node elements can be connected by two different edge types, one denoting control flow and one denoting data flow. Both edge types are represented by an arrowed line.

There are also notation elements for grouping subsets of nodes and edges in an activity. *Partitions (swimlanes)* use vertical or horizontal boundaries to partition a diagram into logical areas, e.g., organizational units in a business model. Nodes and edges are placed inside the related partition. UML2 activity diagrams also allow the use of two-dimensional (grid-like) partitions as depicted in Fig. 7.2(e). Such partitions are combinations of horizontal and vertical swimlanes. Another grouping element is an *expansion region.* This is a strictly nested region of an activity with explicit input and output nodes called *expansion nodes* (Fig. 7.2(f)). Each input is a collection of values. Expansion regions can have keywords in the upper left corner. An *interruptible activity region* has the same notation as an expansion region but without expansion nodes.

A notation element available to all UML diagrams is a *note.* Notes comment on some diagram elements. They are attached to the corresponding



(a) action nodes (b) object node (c) control nodes (d) note (e) partition (f) expansion region

Figure 7.2: Activity diagram notation elements.

elements by a dashed line (Fig. 7.2(d)). In activity diagrams, notes are often used to show post- and preconditions of actions.

Below we list requirements resulting from the visual notation. If a requirement corresponds to one that was already identified in Chapters 2 or 3, we also state the corresponding abbreviation given there.

- Nodes (synonym for vertices) have different sizes (VERTEX_SIZE).
- We have to consider swimlanes/partitions (PARTITION).
- Since edges denote control and data flow it seems to be natural to draw them with respect to a given flow direction (FLOW, BIMODAL).
- Regions can be nested and edges can start/end at a region (expansion input/output nodes). This corresponds to the concept of compound graphs (see Section 2.1) (CLUSTER).
- Notes can be attached to nodes and edges (NOTES).
- Join/fork nodes are two-sided nodes (TWO_SIDED_VERTICES).
- Activities, regions, nodes and edges can have labels (LABEL).

7.1.1.2 Aesthetics

Unfortunately, up to now there have been no empirical studies on aesthetics of activity diagrams. However, since the underlying structure of such diagrams are graphs, we will use common aesthetics (see Section 2.2) that apply to abstract graphs (graphs without special semantics) and thus to

activity diagrams, too. Obviously, aesthetics BEND, CROSSING, AREA and OVERLAP are also important here. SYMMETRY seems to be useless since activity diagrams normally do not have a symmetric structure.

7.1.1.3 Standards for Creating Activity Diagrams

Although the UML specification does not explicitly describe how to layout diagrams, there are a lot of standards, conventions and guidelines on how to do this. A comprehensive collection of those principles that have been proven in practice is given in [3]. Their use increases the usability and clarity of diagrams.

The principles for general UML diagrams are:

- Minimize the number of crossings (CROSSING).
- Draw edges orthogonally (ORTHOGONAL).
- Organize diagrams with respect to the reading direction from left to right or top to bottom (FLOW).
- Use only horizontal labels (HORIZONTAL_LABELS).
- Reorganize larger diagrams into several smaller ones. This principle is based on the fact that it is often easier to have several diagrams at various levels of detail than a single complex one. Hence, complex actions are often decomposed into subactivities. In practice there are almost no activity diagrams containing more than 50 nodes and 80 edges, which allows the use of more complex algorithms than for larger graphs.

The following principles are specific to activity diagrams:

- Incoming and outgoing edges enter a join (fork) node on different sides (TWO_SIDED_VERTICES).
- Swimlanes (partitions) should be ordered in a logical manner, with the primary swimlane placed leftmost. Thus, we assume that the ordering of the partitions is given as input.

7.1.2 Related Work

Up to know there has been no work about layout algorithms for activity diagrams. However, there are several publications about related diagram types.

Most of the recent work on layout approaches for UML diagrams was dedicated to class diagrams. Representatives of two different approaches are UML-Kandinsky/GoVisual [67, 70, 90] and SugiBib [62, 63, 132] (both were

already mentioned in Chapter 3). The first one is based on refinements of the TSM approach while the second one is based on Sugiyama's approach. They support aesthetic criteria like ORTHOGONAL, OVERLAP, CROSSING in various ways and are able to handle constraint FLOW. SugiBib is also able to consider advanced properties like CLUSTER, but on the other hand, it is rather weak for basic properties like VERTEX_SIZE or ORTHOGONAL. None of the above approaches is able to handle constraint PARTITION.

A layout approach for statecharts is given in [33, 34]. Statecharts are extended finite state machines and support the repeated decomposition of states into substates. The approach is based on a combination of Sugiyama's approach with a recursive floorplanning algorithm and an integrated labeling method. It supports aesthetics AREA, CROSSING, OVERLAP, BEND and ASPECT_RATIO as well as constraint CLUSTER. However, the clustering is specific to state decompositions and thus not applicable to activity diagrams.

There is also some work about the visualization of process diagrams. Process diagrams are related to flowcharts and visualize the flow through a process or system. The layout approaches described in [155] and [139] support constraints FLOW and PARTITION. The second one produces orthogonal drawings and is also able to support aesthetics CROSSING, BEND and OVERLAP. However, both approaches are restricted to one-dimensional partitions (horizontal swimlanes) and do not include constraint CLUSTER.

As we have seen, there is no conceptual approach covering all requirements for activity diagrams. On the other hand, there exist quite a few commercial products that claim to support activity diagrams. In the following subsection we review some of them.

7.1.3 Layout Capabilities of UML Tools

An evaluation of automatic layout capabilities of different UML tools with respect to class diagrams was given in [61]. For most tools supporting automatic layout, the results were not satisfying. Meanwhile some of the tools provide improved layout capabilities. In the following we review some popular UML tools with respect to their layout capabilities for activity diagrams. We are particularly interested in their ability to handle constraints FLOW, CLUSTER and PARTITION. All of the tools below except Microsoft Visio support UML2.

- EclipseUML 2008 Studio Edition (Omondo)
  (`http://www.omondo.com`)
  EclipseUML does not support automatic layout for activity diagrams.

- Enterprise Architect 7.1 Professional Edition (Sparx Systems)
  (`http://www.sparxsystems.com`)

Enterprise Architect does not support automatic layout for activity diagrams.

- MagicDraw UML 16.5 Enterprise Edition (NoMagic)
(`http://www.magicdraw.com`)
MagicDraw provides several automatic layout approaches including a hierarchic layout that is able to consider FLOW, PARTITION and CLUSTER. However, using two-dimensional partitions often leads to poor layout results as well as to a violation of constraint FLOW (Fig. 7.3(b)).

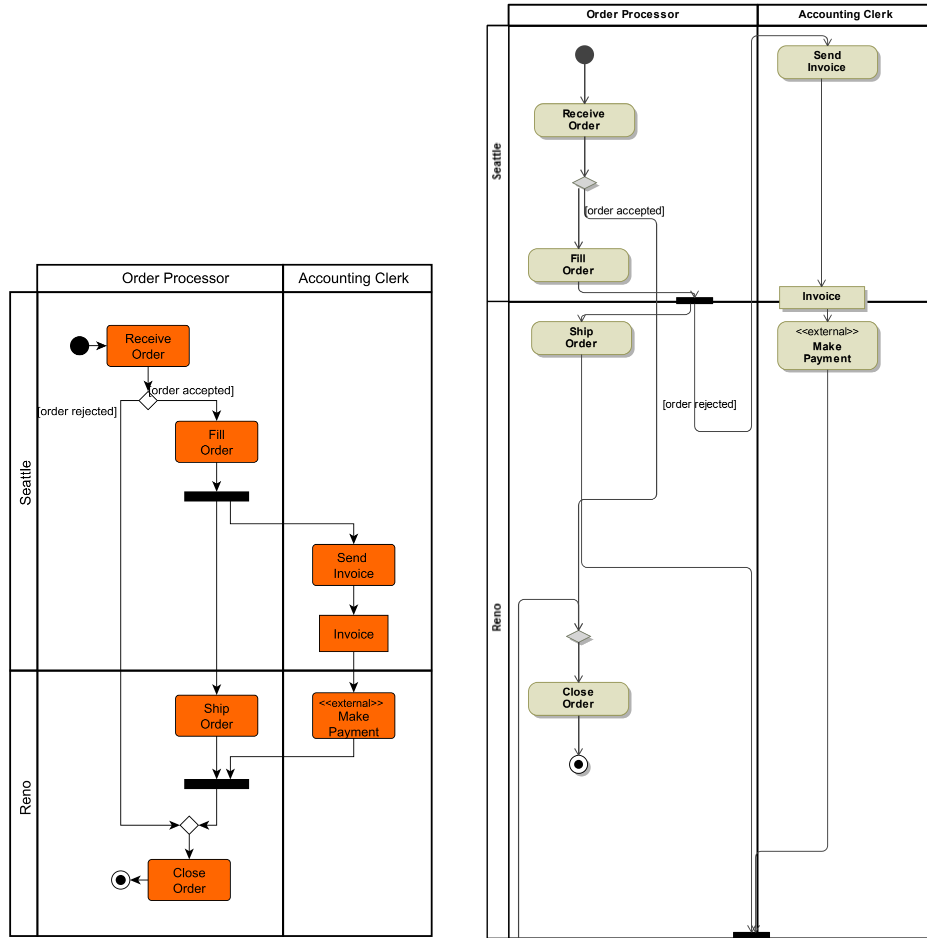
- Office Visio Professional 2007 (Microsoft)
(`http://office.microsoft.com/visio`)
Visio does not allow modeling regions for activity diagrams. It includes different layout approaches that are able to consider constraint FLOW. However, constraints PARTITION and CLUSTER are not supported.

- Poseidon for UML 5.0 Professional Edition (Gentleware)
(`http://www.gentleware.com`)
Poseidon does not support automatic layout for activity diagrams.

- Rational Software Architect V 7.5 Standard Edition (IBM)
(`http://www.ibm.com/software/rational`)
Rational Software Architect does not allow modeling two-dimensional partitions. It only allows one-dimensional partitions, i.e., either vertical or horizontal swimlanes. Furthermore, it does not support the modeling of interruptible activity regions. The provided automatic layout considers FLOW, CLUSTER and one-dimensional partitions. However, constraint FLOW is not observed for edges running between two different partitions. These edges are often routed unfavorably and may intersect with node elements.

- Together 2008 (Borland)
(`http://www.borland.com/us/products/together`)
Together does not allow modeling regions for activity diagrams. Furthermore, it only supports one-dimensional partitions. The provided layout approaches consider constraint FLOW and one-dimensional partitions. Similar to Rational Software Architect, edges running between two different partitions are often routed unfavorably and may intersect with node elements.

- Visual Paradigm for UML 6.4 Enterprise Edition (Visual Paradigm)
(`http://www.visual-paradigm.com`)
Visual Paradigm supports automatic layout for activity diagrams. It generates a hierarchic layout which considers FLOW and CLUSTER. The use of partitions often produces broken layouts.



(a) Our Approach

(b) MagicDraw UML 16.5

Figure 7.3: Two automatic layouts for the example of Fig. 7.1.

The layout capability and quality of the above UML tools differs significantly. None of them was able to produce satisfying results. Only MagicDraw supports automatic layout for two-dimensional partitions. However, as shown in Fig. 7.3, it produces poor layout results when using partitions even for small diagrams.

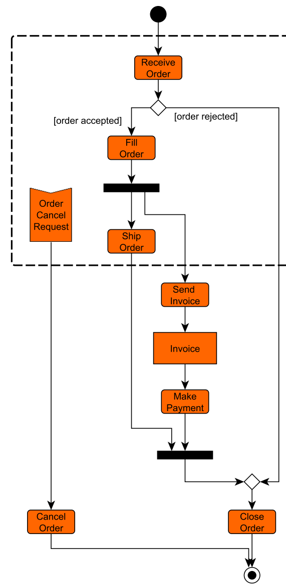
7.1.4 Applying Constraint-Kandinsky

Below, we describe how to apply the `Constraint-Kandinsky` algorithm to activity diagrams. First, an activity has to be transformed into a suitable input graph. The transformation step is straightforward since the underlying structures of activities are mixed graphs containing clustering and partitioning information. All essential requirements identified in Section 7.1.1 are supported by `Constraint-Kandinsky`. This comprises the notation-dependent requirements CLUSTER, PARTITION, FLOW, BIMODAL, VERTEX_SIZE, NOTES, TWO_SIDED_VERTICES and LABEL, the structure-dependent aesthetics BEND, CROSSING, AREA and OVERLAP as well as the requirements ORTHOGONAL and HORIZONTAL_LABELS, which are derived from standards and conventions for creating activity diagrams. Moreover, `Constraint-Kandinsky` allows specifying layout parameters, e.g., if partitions should be drawn in and which kind of edges should be drawn upward.

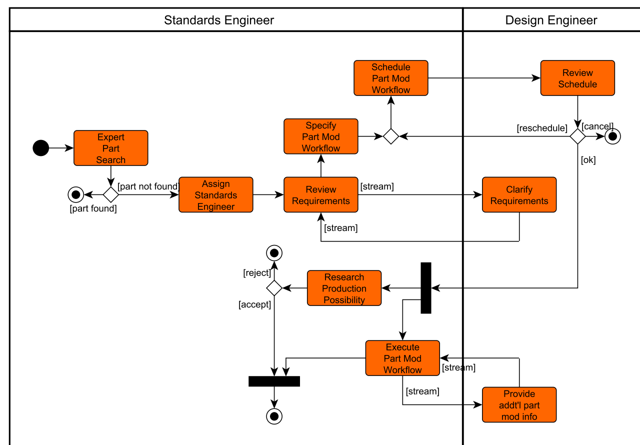
Note that in activity diagrams, the edges of $E_{\uparrow}$ are drawn downward (from top to bottom) instead of upward. We realize this by temporarily reversing the edges' direction. Recall that in activity diagrams, all edges except edges incident to notes are directed. We propose using the following layout configuration for activity diagrams: All edges which are not incident to an initial node, final node or note are added to $E_{\uparrow}$. We set the type of fork and join nodes to `two_sided` and the type of all remaining nodes to `bimodal`. When the user decides to consider FLOW, we assign type `hyper_dummy` to decision and merge nodes, which results in a pleasing fork style representation of these nodes. After applying `Constraint-Kandinsky`, we optionally draw the enclosing rectangle denoting the border of the activity and place the activity's name in the upper left corner. As the experiments in Section 7.3.3 show, the runtime of `Constraint-Kandinsky` is low considering the typical size of activity diagrams.

7.1.5 Examples

Below, we present some activity diagrams drawn with `Constraint-Kandinsky`. Our approach was able to produce satisfying results for all of them.



(a)



(b)

Figure 7.4: `Constraint-Kandinsky` applied to two diagrams of the UML Superstructure specification [118]. While the drawing in (a) considers constraint FLOW, the drawing in (b) does not.

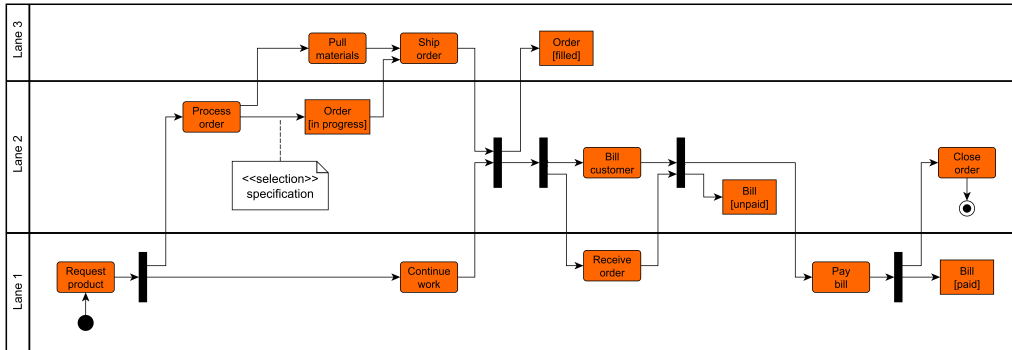


Figure 7.5: `Constraint-Kandinsky` applied to an activity diagram taken from the HyperGraphics website on http://www.hypergraphics.co.uk. The flow direction is from left to right.

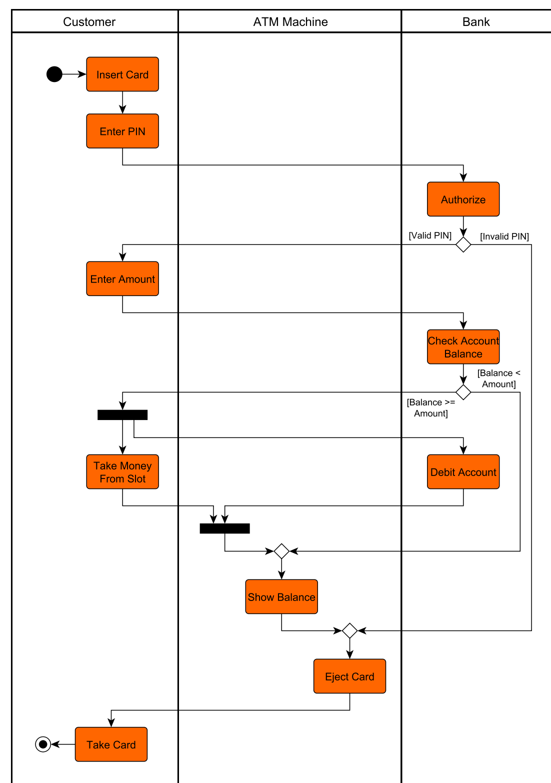


Figure 7.6: Another activity diagram taken from the Borland Together Modeling Guide (http://techpubs.borland.com/together/2008/Together.pdf).

7.2 Visualization of Parallel Computations

Parallel programs are considerably more difficult to design and implement than sequential programs. One has to cope with several additional aspects, in particular, task decomposition, task mapping, communication and synchronization. Moreover, parallel applications usually exhibit a significantly more complex runtime behavior. Despite these aspects, maximizing performance is most crucial in parallel computing. Hence, sophisticated tools for performance analysis and tuning are of paramount importance [113, 126].

Visualization is increasingly being recognized as an effective tool for gaining detailed insights into critical aspects of parallel computing, especially correctness and performance. In this section we present a visualization approach which employs the `Fast-Sugiyama` implementation described in Section 4.6 and which is capable of depicting several performance-relevant properties of task-parallel computations.

Our visualization is based on the *multithreading* parallel programming model (not to be confused with the shared-memory model), which specifically supports irregular task-parallel applications. This programming model was originally introduced in the Cilk programming language [125]. It is located on a medium level of abstraction, thus low-level synchronization and communication are carried out completely transparently. Our visualization component is integrated with the parallel system platform DOTS [17] (*Distributed Object-Oriented Threads*), which provides extensive support for the multithreading model on a wide range of shared and distributed memory architectures [16]. Since we use the properties and features of DOTS on a rather abstract level, our method remains applicable without major changes for other task-parallel models [140].

7.2.1 A Graph-Theoretic Model for Parallel Computations

Parallel programs based on the multithreading programming model are called *multithreaded programs*. A *multithreaded computation* results from the execution of a multithreaded program on a given input.

In [19], Blumofe and Leiserson introduce a graph-theoretic model for multithreaded computations. Such a computation can be represented as a directed acyclic graph called an *execution graph*. The vertices of the execution graph represent basic tasks which can be either regular computations or communication operations. Vertices are connected by *continue*, *spawn*, or *data-dependency edges*. *Continue edges* indicate the sequential ordering of basic tasks and also define the extent of a *thread*. Threads are the main structural entities of a multithreaded computation. *Spawn edges* depict a parent/child relationship among two threads. The parent thread creates the child thread and passes arguments to it. The tree composed of all threads of a computation and corresponding *spawn edges* is called a *spawn*

tree. A thread can produce one or more results which are consumed by other threads. Such producer/consumer relationships between two threads are indicated by *data-dependency edges*. Threads can be executed concurrently, provided that data-dependencies between threads are considered. An important subclass of general multithreaded computations are *strict multithreaded computations* where all data-dependency edges of a thread go to an ancestor of the thread in the spawn tree. Strict computations are considered to be the most comprehensive class of multithreaded computations that are well-structured. This property was utilized for the design of the DOTS API.

The DOTS API is a compact and completely orthogonal API for writing multithreaded programs. A thread is created using the dots_fork or dots_hyperfork primitive. The primitives differ in the assignment of threads to so-called thread groups [16]. We omit the concept of thread groups here, since they are not important for our visualization. In both cases, a procedure to be executed by the child thread and an argument-object has to be supplied. Threads return result objects employing dots_return. The last result of a thread is delivered by the final return statement of the procedure. The dots_join primitive is used to retrieve results of threads applying *join-any* semantics: The first result which becomes available from any thread in the group is delivered. If no results are available, the calling thread is blocked until a thread of the group delivers a result. Fig. 7.7 shows the execution graph of a strict computation.

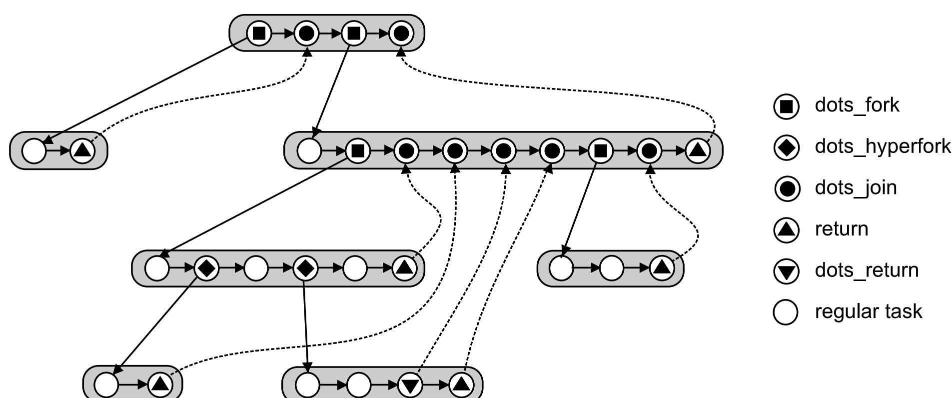


Figure 7.7: Execution graph of a strict multithreaded computation [16]. Communication operations are marked with the corresponding DOTS primitives.

The graph-theoretic concept described in this section provides a starting point for our visualization and enables us to take advantage of our `Fast-Sugiyama` implementation.

7.2.2 Requirements for Visualizing Parallel Computations

To determine the basic requirements for our visualization, we first look at task decomposition and load balancing issues. This is necessary to identify performance-critical issues appearing in this context. Our visualization should enable the application designer to reliably detect manifestations of parallel overhead and to investigate their individual root causes.

7.2.2.1 Dynamic Task Decomposition and Load Balancing

Since, in typical multithreaded computations, threads are continuously generated, task decomposition and load balancing are tightly coupled and can interfere with each other in various ways. The DOTS runtime system employs the so-called *distributed task pool model* for load balancing. Upon creation, a thread is *reified*. It only exists in the form of a passive object which encapsulates the thread's complete state. Initially, reified threads are placed in the local task pool. They can be instantiated and executed locally (when a processor becomes available) or might first be transferred to a remote task pool for load balancing purposes. In the execution graph, the load balancing process can be represented by additional vertices adjacent to fork vertices which denote enqueue and dequeue events.

To ensure high parallel scalability, dynamic task decomposition and load balancing have to be performed in a fully distributed manner. However, in a fully distributed system each processor has only limited knowledge of the global state of the parallel computation. As a consequence, dynamic task decomposition and the load balancing process must be steered by local strategies and several corresponding parameter values, e.g., thresholds on the size of task pools, which interact in a complex manner. The parameters of the decomposition and load balancing strategies are sensitive to individual characteristics of applications, like the degree of irregularity. Furthermore, they depend on the specific properties of the parallel target platform, like heterogeneity or volatility of resources. Finding appropriate strategies along with optimal parameter values which minimize both processor idling and task interaction overhead is a delicate task. In practise, this can only be accomplished by extensive experimentation on representative problem instances. In the following, we identify several classes of performance-critical issues typically encountered in this context.

7.2.2.2 Manifestations and Causes of Parallel Overhead

Basically, achieving high parallel efficiency requires minimizing the overall parallel overhead of a computation. Parallel overhead is mainly determined by three factors: idle times of processors, communication overhead and excess computation. The main goal of our visualization approach is to reveal

the following specific conditions that directly or indirectly contribute to one of these factors:

- **Processor Idling:** One or more processors remain idle for a prohibitively long time interval. A typical reason for this situation is that (temporarily) too few reified threads are present in the system due to an unsuitable decomposition approach. Another possible reason is that the load balancing strategy is too slow-acting because of inappropriate threshold and timeout values or due to limited scalability.
- **False Sharing:** Threads with a too fine granularity are generated and transferred to other processors. This effect can be caused by an unsuitable decomposition approach or by badly chosen parameter values. False sharing can considerably increase communication overhead.
- **False Parallelism:** Threads are dynamically created and subsequently executed on the same processor (after the parent thread's execution is finished). This condition is mainly affected by inappropriate threshold values. False parallelism contributes to overhead due to excess computation.
- **Roaming:** Threads are passed to several processors before they are actually executed. The so called *ping-pong effect* is a special case of roaming. Here, a thread is passed back and forth between two processors before it finally gets executed. Extensive roaming can be caused by inapt threshold values. Roaming can seriously increase communication overhead of the computation.

7.2.2.3 Basic Requirements for the Visualization

To reliably identify all of the above manifestations of parallel overhead and to investigate their individual root causes, the visualization must present the following information in a distinct manner:

- Dynamic mapping of threads to processors.
- Activity and idle times of processors.
- Parent/child relationships of threads.
- Data dependencies between threads.
- Time flow of the execution process.
- Transfer activities of reified threads between processors.
- Documentation of the internal state of the runtime system for each relevant event.

To effectively transfer insights gained by visualization into actual performance improvements, it is essential to tightly embed the visualization into a comprehensive development workflow, establishing a seamless execute-analyze-tune cycle. A crucial prerequisite for tool-integration is that the employed visualization method is fast enough to enable interactive use. Note that for applications with high irregularity, execution graphs typically become considerably large.

Basically, the requirements listed above can be met by appropriately visualizing execution graphs which are enriched with supplemental information. In Section 7.2.4.1 we introduce an advanced layout algorithm for execution graphs based on our `Fast-Sugiyama` implementation. Section 7.2.4.2 deals with means for further strengthening the expressiveness of execution graphs.

7.2.3 Related Work

Heath et al. [97] present a general model for the visualization of parallel performance data along with a discussion of many visualization concepts and previously developed principles. In the following, we give a short overview of representative visualization environments which contributed to the development and understanding of these basic concepts and principles.

ParaGraph [96] represents an early research effort in performance evaluation and visualization of message-passing parallel programs. It provides graphical animations of different runtime properties and also graphical summaries of performance metrics.

Vampir [116], the commercial version of the PARvis [115] system, is especially designed for MPI (Message Passing Interface) programs. Vampir features diverse visualization capabilities for MPI-related resources at different levels of abstraction, e.g., timeline views and parallelism displays.

Paradyn [114] is an integrated suite of performance measurement tools for parallel and distributed applications. It is based on dynamic instrumentation of unmodified executables for generating performance profiles. This approach is capable of adapting the data collection process during execution to be able to dynamically change the focus of the analysis. Paradyn features several visualization capabilities, e.g., time histograms or bar charts for individual performance metrics.

Parade [107] is based on the Polka [142] animation toolkit and represents a general purpose visualization environment designed to create algorithm animations. A well-known sub-project is PVaniM [149], which provides support for the visualization of message-passing parallel programs that are based on PVM (Parallel Virtual Machine). It is designed as a two-phase approach where online visualization is employed for large-grained events that are influenced by the computing environment, and post-mortem visualization is used for a more detailed program analysis and subsequent tuning.

A commonality of all systems discussed above is that they primarily focus on the visualization of diverse system-level performance metrics, like processor utilization, communication rates or memory access times. In contrast, our approach is located on a higher level of abstraction, focusing on specific properties of the multithreading task-parallel programming model. It is based on visualizing execution graphs and thus substantially relies on human intuition and pattern recognition capabilities.

A comparable approach for visualizing execution graphs of multithreaded parallel applications is described in [143]. That approach relies on the distributed threads system (DTS) [31], which is a programming environment for portable parallel applications. However, the described visualization is much simpler and does not include an adequate representation of the different processors and transfer activities. Furthermore, our approach is based on a tight integration of the visualization into an integrated development environment to assist the application designer in carrying out an interactive performance-analyzing and tuning process.

7.2.4 A Layout Algorithm for Execution Graphs

For the layout of execution graphs we use our `Fast-Sugiyama` implementation. The specific modifications needed to produce adequate results are described below. Afterwards, we present additional visual decorations which we use to emphasize pertinent details of these graphs.

7.2.4.1 Specific Modifications of the Layout Algorithm

The main objective of our new visualization approach for parallel program execution is an adequate representation of the execution's time flow as well as of the threads and the different processors. To meet the last objective, we use vertical partitions (swimlanes) which represent the processors. The time flow is visualized by using horizontal partitions.

The input of our algorithm is a directed acyclic execution graph $G = (V, E_D)$, constructed as described in Section 7.2.1. For its construction we analyze the event trace generated during the execution of a parallel program. To ensure a correct event ordering, we also have to compensate for the time skew. Additionally, the input consists of a function $p\text{-}id\colon V \to \mathbb{N}$ assigning each vertex the ID of the corresponding processor, a function $\tau\colon V \to \mathbb{N}$ denoting the time when $v \in V$ was processed as well as a function $t\text{-}id\colon V \to \mathbb{N}$ assigning thread IDs to vertices (vertices that denote queuing events for reified threads are not assigned to threads). We assume w.l.o.g. that $1 \leq p\text{-}id(v) \leq \Phi$, $\forall\, v \in V$, where Φ denotes the number of processors. An edge $e = (v, w)$ between two vertices of the same thread ($t\text{-}id$(v)=$t\text{-}id$(w)) is called an *execution edge.* Note that the execution edges of a thread build up a simple monotone path. Furthermore, if the ordering

of the different processors is not given as input, we can use the heuristic described in Section 4.4.4 to find a suitable order for them.

For the visualization, we use our `Fast-Sugiyama` implementation along with the modifications to handle partitions (see Section 4.4). In the following, we introduce the specific modifications for visualizing parallel execution graphs. We, therefore, reconsider the different phases of Sugiyama's algorithm. Since the input graph G is acyclic by construction, we can skip the cycle removal phase. Hence, we start with the layer assignment.

Layer Assignment:

In our approach we use the layering (y-coordinate) to reflect the time flow of the program execution. We, therefore, introduce "synchronization points", which represent certain points in time. A synchronization point is drawn as a horizontal line. For each pair of consecutive synchronization points representing times t_j and t_{j+1} our layering has the following property: a vertex $v \in V$ is placed between the two synchronization points if and only if $t_j \leq \tau(v) \leq t_{j+1}$.

We can affect the layering results by changing the length of the time interval between two synchronization points. If we choose large intervals, the height of the graph is smaller but the time flow is poorly reflected. If we choose short intervals, the height of the graph increases but the time flow becomes clearer.

The customized layering works as follows: First, we sort V according to τ in ascending order. We call the resulting list L_τ. Let T_j denote the time interval between times t_j and t_{j+1}. We partition V into time intervals $T_1,\ldots,T_k$, where k depends on a uniform, user-defined interval length δ. A vertex $v \in V$ is assigned to interval $1 + \lfloor \frac{\tau(v) - min_\tau}{\delta} \rfloor$, where min_τ is the time value of the first vertex in L_τ. If we choose a small δ, the number of time intervals k can be very large. However, this has no impact on the running time, because it is sufficient to consider only non-empty time intervals (intervals with at least one vertex). The number of such intervals l is bounded by the number of vertices. Let $T_{i_1},\ldots,T_{i_l}$ denote the subsequence of non-empty time intervals. Each time interval is mapped to a partition row using function py (introduced in Section 3.3.1). More precisely, for a vertex $v \in V$ is $py(v) = j$, $1 \leq j \leq l$ if and only if $v \in T_{i_j}$. For the layer assignment, we propose to use a longest path layering because it produces a minimum number of layers and can be computed in linear time (see Section 2.4.2).

Sorting the vertices of V according to τ requires $O(|V| \log |V|)$ time and the assignment to time intervals $O(|V|)$. The number of additional dummy vertices and edges needed to model time intervals is $O(|V|)$. Thus, the layer assignment phase runs in time $O(|E_D| + |V| \log |V|)$.

After the layer assignment, we normalize the input graph. We have to extend the functions *p-id* and *t-id* to dummy vertices as follows: each dummy vertex d_e which splits an edge $e = (u, v)$ gets the value $p\text{-}id(d_e) = p\text{-}id(u)$. Furthermore, if e represents an execution edge we set $t\text{-}id(d_e) = t\text{-}id(u)$ $(= t\text{-}id(v))$.

Crossing Minimization:

The crossing minimization has to consider the representation of the processors and threads. Since we represent each processor by a vertical partition we simply set $px(v) = p\text{-}id(v)$ for each vertex $v \in V$. To obtain the vertical line representation of threads (all vertices of a thread have the same x-coordinate), we have to change the calculation of the measure values. The measure of a vertex $v \in L_i$ is set to the position of its neighbor $w \in L_{i-1}$ for which $t\text{-}id(w) = t\text{-}id(v)$ holds. Note that for such neighbors we always have $p\text{-}id(w) = p\text{-}id(v)$ and thus $px(w) = px(v)$. For a vertex v there is at most one such neighbor, because the execution within a thread is strictly serial. If there is no such neighbor, the measure is determined traditionally with the median or barycenter heuristics. Our measure calculation has the property that it prevents crossings between execution edges as well as between middle segments of long edges and execution edges. Thus, execution edges can always be drawn vertically.

Horizontal Coordinate Assignment:

Let $G_C = (V_C, E_C)$ denote the compaction graph computed during crossing minimization. The vertical alignment of vertices belonging to the same thread can be realized by simply mapping those vertices to the same vertex in the compaction graph. The above modification of the measure calculation guarantees that this never introduces directed cycles in the compaction graph. Obviously, this modification does not increase the size of the compaction graph.

We use the compaction graph to perform the horizontal coordinate assignment, using the algorithm proposed by Brandes and Köpf (see Section 2.4.4). We always resolve alignment conflicts in favor of middle segments and execution edges, between which no alignment conflicts occur.

The number of processors Φ as well as the number of non-empty time intervals l is less than or equal to $|V|$. Since the introduced modifications do not increase the runtime of our original implementation of Sugiyama's approach and we only use partition constraints, our visualization algorithm for execution graphs has runtime $O((|V| + |E_D|) \log |E_D|)$ and requires $O(|V| + |E_D|)$ space. In Section 7.3 we will give an experimental evaluation of this algorithm.

7.2.4.2 Visual Decoration of Execution Graphs

As our approach essentially relies on displaying execution graphs, the main perspective of the visualization is closely related to the programming model. This characteristic considerably reduces the semantic gap between the visual representation of computations and the application code. In addition to applying the above layout algorithm for generating the display, we visually emphasize pertinent details of execution graphs. Moreover, we link the visual representation with information about the system level performance as well as the internal state of the runtime system. This approach provides the application designer with detailed and cross-linked information at several levels of abstraction enabling a thorough investigation of reasons for performance problems. In particular, we provide the following visual decorations and enhancements for execution graphs:

Representation of Processors
We represent swimlanes (denoting processors) as gray background rectangles. Idle times of processors are indicated by a red-colored background. This enables us to quickly grasp the processor utilization of a computation.

Representation of Threads
To emphasize vertices belonging to the same thread, we vertically align them and give them the same color. Vertices representing DOTS primitives, e.g., fork and join vertices, are given individual shapes. We additionally introduce vertices indicating start- and end-execution events of a thread. Each vertex is associated with information about the internal state of the runtime system at the time the corresponding event took place.

Edge Types
Execution graphs contain two different kinds of edges: Execution edges indicate the execution process of a thread, while data-flow edges depict communication between threads. We distinguish both types by drawing them with different thickness. According to which of these aspects is of interest, the representation of the two edge types can be switched.

Load Balancing Activities
As discussed in Section 7.2.2.1, load balancing is performed by transferring reified threads between task pools. Segments of the execution graph representing times when a thread is reified (including the corresponding queueing events) can be selectively shown for individual threads. This allows detailed investigations of roaming effects.

Additionally, we provide extensive customization capabilities for layout parameters, e.g., time interval length, distance between layers or appearance of swimlanes.

7.2.5 Tool Integration

We employ Eclipse [133] for building a comprehensive development environment around our visualization method. Eclipse is a universal tool platform which considerably facilitates integration of new functionality via a sophisticated plug-in mechanism. For the development of parallel C++ programs we use the C++ Development Tool (CDT), which is a fully functional C and C++ IDE for the Eclipse platform. Our Eclipse plug-in (Fig. 7.8) provides the following wizards and views:

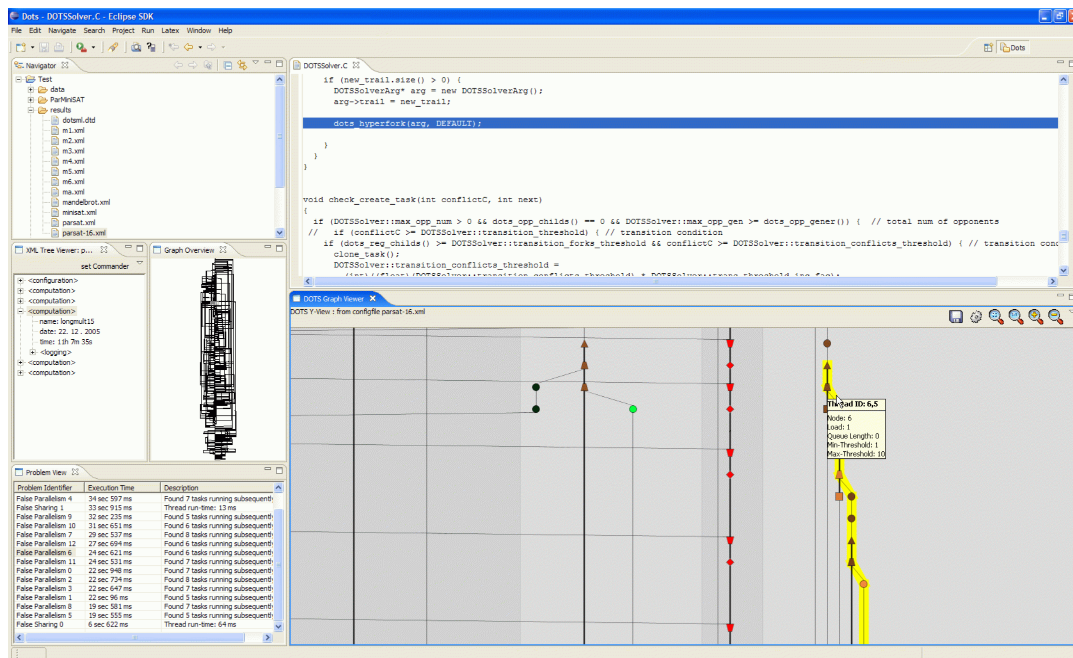


Figure 7.8: Integration with Eclipse.

Configuration Wizard

The Configuration Wizard guides users through the configuration of the parallel computation environment (e.g., the definition of the employed nodes/processors, locations of executables and startup procedures). An XML document stores the configuration as well as information about individual program runs performed in this configuration (e.g., parameters of the DOTS runtime system, program input/output data and event traces).

Launch View
: The launch view displays information about the progress of a parallel computation, e.g., program output and event logs.

Graph View
: The graph view is the main view used by our tool, displaying the decorated execution graph. Double-clicking on a vertex denoting a DOTS primitive marks the corresponding line in the source code editor. Leaving the mouse-pointer over a vertex pops up a tooltip displaying the internal state of the runtime system at the time of the corresponding event. Cross-execution views (enabling visual comparison of different programs runs) can simply be realized by opening several graph views, one for each computation of interest. Furthermore, the graph view provides scrolling and zooming functionality.

Graph Overview
: The graph overview displays the entire execution graph and highlights the region currently displayed within the graph view. By moving the highlighted region on the graph overview, the graph view is set to the corresponding region.

Performance Problems Catalog
: This view presents the list of possible performance problems, which can be automatically detected from the event trace. Double-clicking on a list item points the graph view to the affected region of the execution graph and highlights the relevant graph elements. The different problems are identified as follows:

- **Roaming**: Execution graphs contain vertices denoting queue events. We identify possible roaming problems by searching maximum sequences of consecutive queue vertices whose length exceeds a given threshold value.
- **False Sharing**: Execution graphs also contain vertices that denote start-execution, end-execution and fork events. To identify possible false sharing problems, we check, for each fork vertex, whether the runtime of the created thread is smaller than a given threshold value. We, therefore, use the function τ and the start- and end-execution vertices of a thread to determine the thread's runtime and the function *p-id* and the fork and start-execution vertex to determine if the thread was transferred to another processor.
- **False Parallelism**: To identify possible false parallelism problems, we use the function *p-id* to check if the start-execution vertex of a created thread has the same processor-id as the corresponding fork vertex.

Besides these main components, our tool provides several auxiliary features, e.g., exporting a graph representation for postprocessing and documentation purposes or customizing threshold values for adapting the severity level of the problems to be listed in the performance problems catalog.

7.2.6 Analysis Methodology and Usability Example

In this section, we demonstrate our visualization methodology in the light of a representative example application. We consider parallel Boolean satisfiability (SAT) solving [18]. State-of-the-art sequential SAT solving methods employ sophisticated heuristics to prune the search space. The parallelization of these heuristics results in a high degree of irregularity. Thus, all subsequently discussed issues of parallel SAT solving can be considered archetypal for a large class of highly irregular task-parallel applications.

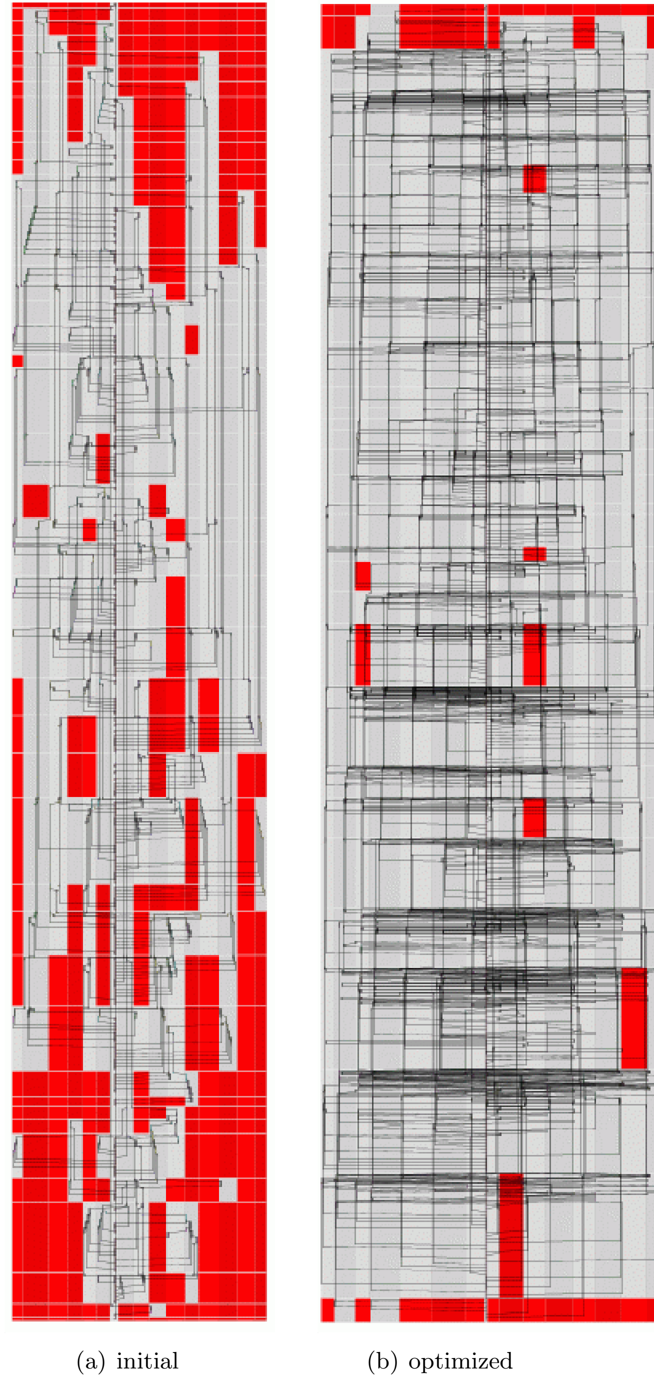
Typically, the first step in analyzing a parallel program run is to check the overall load balancing. Fig. 7.9(a) shows a computation with a high degree of processor idling (red-colored swimlanes in the background of the execution graph indicate idle times). Using the navigation capabilities of our tool one can zoom to the affected regions and investigate possible causes of processor idling by examining the local structure of the graph and checking the state of the runtime system on other processors, e.g., sizes of task pools. To see the effect of modifications, cross execution views can be employed for visual comparison. Fig. 7.9(b) shows the execution graph of a computation with the same input resulting from tuning parameter values for problem decomposition and load sensing intervals. As the figure indicates, processor idling is now significantly reduced.

When processor idling has been minimized, attention can be turned to optimizing communication overhead and reducing excess computation. Specifically, false sharing, false parallelism and roaming phenomena must be eliminated to achieve this objective (see Section 7.2.2.2). For optimizations carried out at this stage in the tuning process, it must be ensured that the overall load balancing is not affected. Thus, all tuning actions should be first evaluated by using the overview graph, as described previously.

By employing the performance problem catalogue, one can detect, for each class of performance problems, the affected region in the execution graph. The display is automatically pointed to the region and the corresponding part of the execution graph is highlighted in yellow.

Fig. 7.10 shows a part of the execution graph exhibiting typical staircase patterns indicating a high degree of false parallelism. This is a result of inadequate timing parameters that control the minimum delay for consecutive thread forks. By determining the largest of the thread cascades, one can obtain a better estimate of the corresponding timing parameters.

In Fig. 7.11 one can see the execution graph of a computation which suffers from noticeable false sharing effects. Here, employing a different



(a) initial (b) optimized

Figure 7.9: Overview of the load balancing of two parallel SAT computations. Idle times are indicated by a red-colored background.

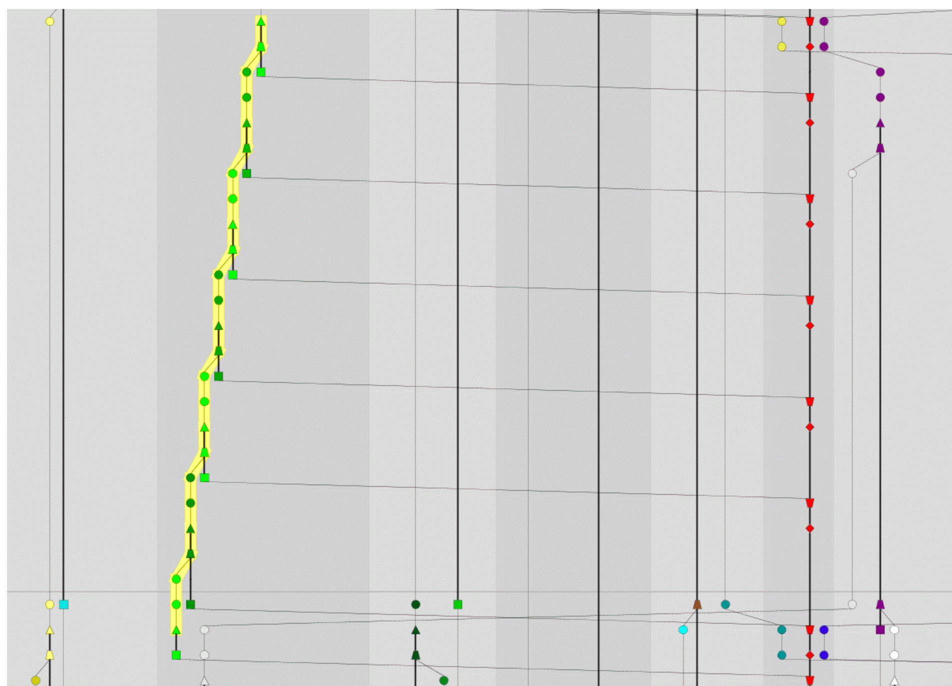


Figure 7.10: Execution graph of a parallel SAT computation exhibiting false parallelism.

task transfer strategy and refining load-sampling intervals can result in an improvement.

A typical example of roaming is shown in Fig. 7.12. Roaming effects can be reduced by adapting thresholds on the size of task pools or considering different transfer strategies.

In principle, these performance-related issues could also be detected by more generic visualization methods for distributed-memory architectures, e.g., message passing displays. Typically, in such displays, processor activity is depicted in a horizontal time-line and message transfers between processors are indicated by corresponding arrows. Thus, generic message passing displays are located more closely to the system level. But at this level of abstraction one can not distinguish between different message types occurring in our programming model, like task transfers and task requests (employed by receiver-initiated load balancing strategies). Also, events which do not immediately trigger a communication operation (e.g., creation of a new thread or calling the join primitive) cannot be easily identified in generic message passing displays. This makes it considerably more difficult to extract appropriate information for optimizing program execution. In particular, finding root causes of performance problems located on the pro-

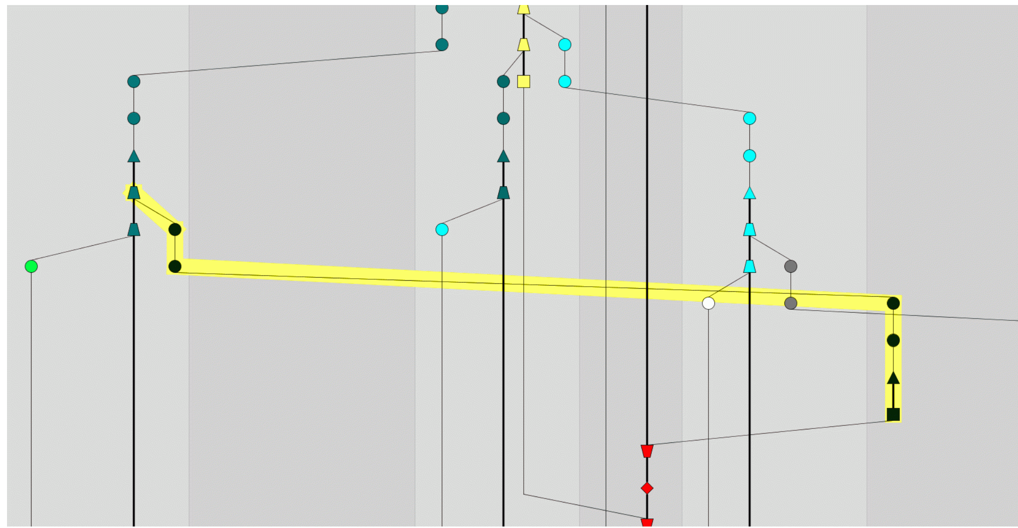


Figure 7.11: Execution graph of a parallel SAT computation with false sharing effects.

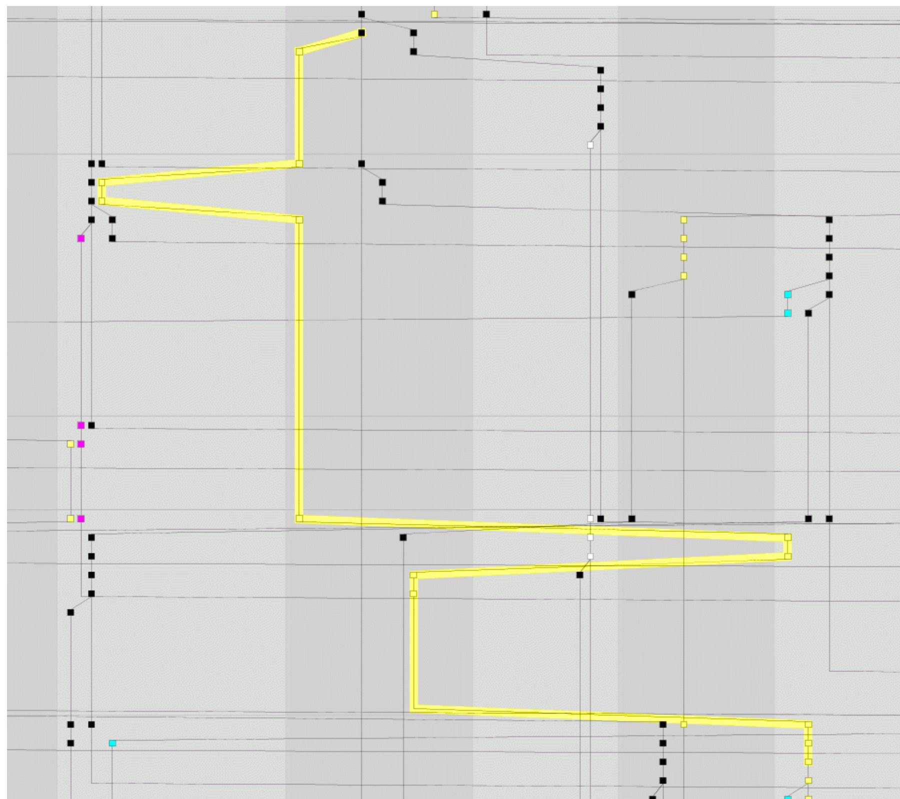


Figure 7.12: Execution graph of a parallel SAT computation suffering from roaming.

gramming model level is virtually impossible with displays at lower levels of abstraction.

7.3 Experiments

In this section we present the results of our empirical evaluation. First, we describe the experimental setting as well as the different graph sets used for testing our algorithms. Then we state the results related to our `Fast-Sugiyama` implementation as well as the results of the `Constraint-Kandinsky` algorithm.

7.3.1 Data and Experimental Setting

All experiments were performed on an Intel Core 2 Duo System with 2.13 GHz and 2 GB main memory, running Scientific Linux 4. We used Sun's Java Platform, Standard Edition 6 as the runtime environment. We performed the experiments using the following sets of graphs:

Connected Directed Random Graphs: We generated a total number of 3000 connected directed random graphs $G = (V, E)$, i.e., 100 graphs for each combination of the following parameters:

$$\begin{aligned} |V| &\in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\} \\ |E|/|V| &\in \{1.5, 2.0, 2.5\} \end{aligned}$$

The graphs were constructed as follows: To obtain a connected graph with n vertices and m edges, we first construct a random tree with n vertices. Therefore, we start with a single-vertex graph $G = (\{v\}, \emptyset)$, iteratively add a new vertex w to it and connect w to a randomly chosen vertex already in G. Obviously, after $n-1$ iterations we obtain a connected tree with n vertices and $n-1$ edges. Now, we add the $|m|-|n|+1$ outstanding edges to G by randomly choosing two vertices and connecting them.

Execution Graphs: This graph set contains five different graphs that represent program runs of a parallel SAT solver as described in the preceding section.

Rome Graphs: The Rome graph test suite [48] is a well-known graph collection that contains about 11000 undirected graphs[1] with number of vertices ranging from 10 to 100. These graphs have been generated from a core set of 112 graphs used in "real life" software engineering and database applications. The density of the Rome graphs is between 1 and 2 with an average value of 1.4.

[1] `http://www.dia.uniroma3.it/people/gdb/wp12/LOG.html`

7.3.2 Test Results for Fast-Sugiyama

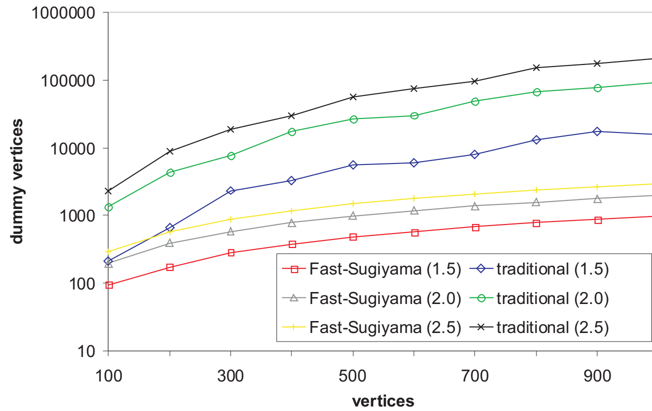
In the following, we compare the results of our `Fast-Sugiyama` implementation to the results of a traditional implementation. Both implementations use a layer-by-layer sweep with the barycenter heuristic. To obtain comparable results, we fix the number of layer sweeps. More precisely, for both algorithms we perform 12 iterations, where each iteration consists of an up and a down sweep (a single sweep performs the two-layer crossing minimization for each pair of adjacent layers). Furthermore, we randomly change the processing order of the graph elements after every third iteration. Both algorithms start with exactly the same layering produced by the simplex layering approach described in Section 2.4.2. Recall that this approach gives a solution which minimizes the overall edge length.

For the experiments, we use the connected random graphs. We use the feedback arc set heuristic described in Section 2.4.1 to make these graphs acyclic. As shown in Fig.7.13(a), our `Fast-Sugiyama` implementation leads to an enormous reduction of the dummy vertices inserted during the normalization. Note that the diagram uses a logarithmic scale on the y-axis. The reduction also leads to a significant improvement of the crossing minimization time (Fig. 7.13(c)). Our improvements make it possible to handle graphs that could not be handled before, due to high memory consumption and runtime of the traditional implementation. Fig. 7.13(b) shows that the number of crossings is almost the same for both algorithms. Slight differences are caused by the randomization as well as different refinements used by the algorithms.

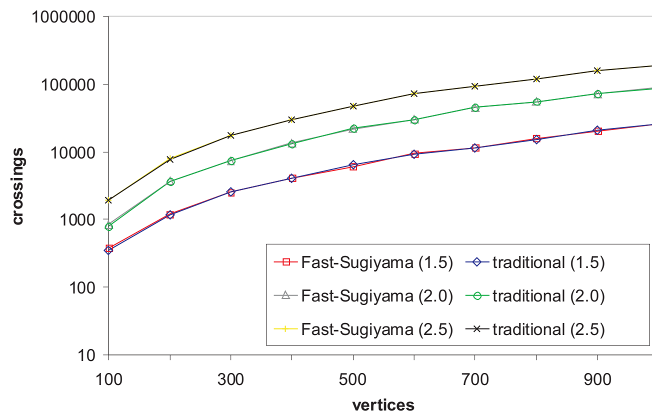
In the second experiment, we applied `Fast-Sugiyama` to execution graphs as described in Section 7.2.4.1. We test the implementation using five execution graphs of different size. The parallel computations were performed using 16 processors (modeled as vertical partitions). For the layering we use a longest path strategy with time interval length $\delta = 1\text{s}$. As shown in Table 7.1, our implementation performed well. Again it leads to a significant reduction of the number of required dummy vertices.

# Vertices / Edges	Time (ms)	# Dummy Vertices		# Crossings
		`Fast-Sugiyama`	traditional	
1952 / 2282	322	945	13612	1947
2690 / 3121	615	1280	23068	3129
4933 / 5732	1095	2259	46475	6363
11330 / 13259	3968	9128	133877	17480
17433 / 20460	7125	14802	244156	39140

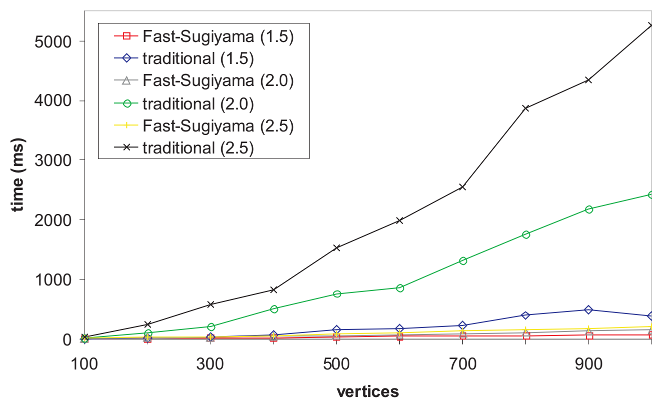
Table 7.1: Results of the experiments for execution graphs.



(a) number of required dummy vertices



(b) number of crossings



(c) crossing minimization time

Figure 7.13: Comparing two different implementations of Sugiyama's algorithm for connected random graphs with density 1.5, 2.0 and 2.5. The number of vertices is shown on the x-axis.

7.3.3 Test Results for Constraint-Kandinsky

In the following, we use the Rome graph collection to test the quality of the `Constraint-Kandinsky` algorithm with respect to the different drawing constraints. We measure the number of crossings in the final drawing, the number of crossings reduced by the rerouting step, the number of bends as well as the runtime of the planarization and orthogonalization phase. For our experiments on the different constraints, the compaction time never exceeds 50 ms and is always below 10% of the overall runtime. Note that the results presented here may be specific to our algorithm/implementation as well as to the chosen graph set and, hence, do not give a general insight into the complexity of the constraints. For all tests we use the following general configuration: The planarization uses our `Fast-Sugiyama` implementation with the barycenter heuristic and 12 iterations of the layer-by-layer sweep. The rerouting step performs 3 randomized passes.

When we apply a state-of-the-art TSM-based layout approach, i.e., the "OrthogonalLayouter" of the yFiles library [156], to the Rome graphs without considering any constraints, the average number of crossings for the instances with 100 vertices is 39.2. The average number of bends for these instances is 39.1 and the average layout time is 61.2 milliseconds.

7.3.3.1 Mixed Upward Drawings

In this subsection we present the results of the `Constraint-Kandinsky` algorithm with respect to constraint FLOW. First, we compare our new layering strategy described in Section 4.2.1 to traditional layering strategies. We randomly assign directions to the edges and then remove cycles using the feedback arc set heuristic described in Section 2.4.1. We apply our `Fast-Sugiyama` implementation to the different layering strategies. We compared our new layering to the longest path layering and the simplex layering (both described in Section 2.4.2) as well as to a topological layering. The topological layering calculates a topological order π of the vertices and then assigns each vertex v to layer $\pi(v)$. Fig. 7.14 shows the number of crossings for the different layering strategies. The x-axis gives the number of vertices of the graph instances. Our new layering significantly reduces the number of crossings in the resulting drawing, i.e., compared to the simplex layering it produces up to 20% fewer crossings. The results of the topological layering indicate that our improvement does not depend solely on the sparse layers. The runtime for calculating a result did never exceed 10 milliseconds.

In the second experiment, we compare `Constraint-Kandinsky` to the UML-Kandinsky approach described in [69]. For the UML-Kandinsky approach we use the parameter set proposed in [64]. For the experiment we use three different densities of upward edges, i.e., $|E^{\uparrow}|/|E| \in \{0.3, 0.6, 0.9\}$. The

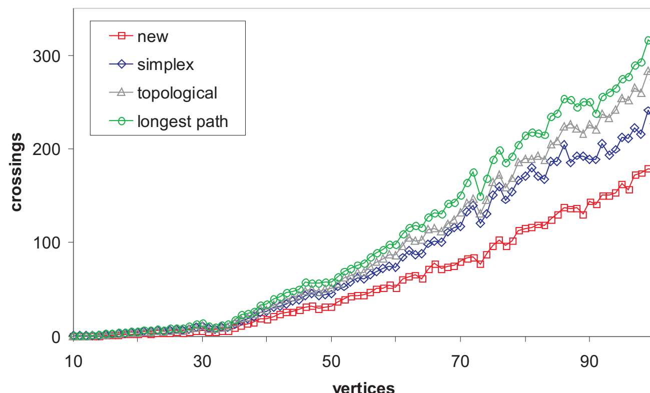


Figure 7.14: Number of crossings induced by different layering strategies.

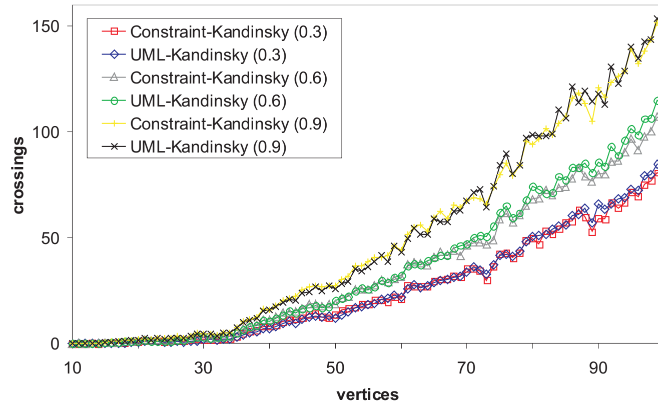
edges of $E^{\uparrow}$ are chosen randomly. Cycles in the graph $(V, E^{\uparrow})$ are removed using the feedback arc set heuristic described in Section 2.4.1.

The results shown in Fig. 7.15 and Fig. 7.16 lead to the following observations:

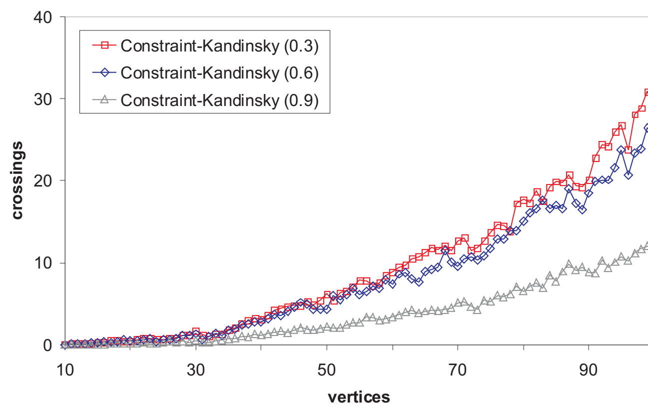
- For both approaches, the number of crossings increases with the number of upward edges. `Constraint-Kandinsky` produces slightly fewer crossings than UML-Kandinsky.
- The planarization of `Constraint-Kandinsky` is significantly faster than the planarization of UML-Kandinsky.
- For both approaches the number of bends increases with increasing number of crossings. This can be attributed to the increasing size of the planarized graph. This also leads to an increasing size of the Kandinsky network and, thus, to an increasing orthogonalization time.
- For `Constraint-Kandinsky` the number of crossings reduced by the rerouting step decreases with the number of upward edges. This is not surprising since we do not reroute upward edges.
- The rerouting step of `Constraint-Kandinsky` is very fast and never exceeds 4 milliseconds.

7.3.3.2 Bimodal Drawings

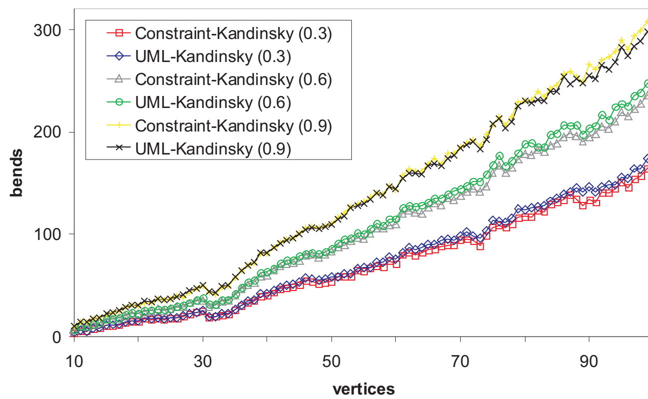
We test constraint BIMODAL using a similar setting as that for constraint FLOW, i.e., for $|E_D|/|E| \in \{0.3, 0.6, 0.9\}$. We assume that all vertices of V are of type `bimodal`. The directed edges are chosen randomly.



(a) overall number of crossings

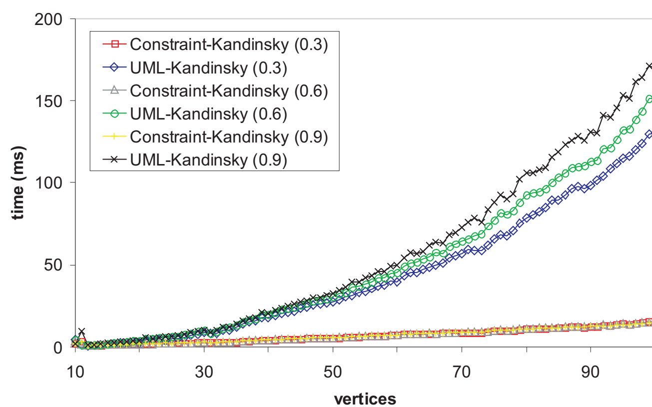


(b) crossings reduced by rerouting step

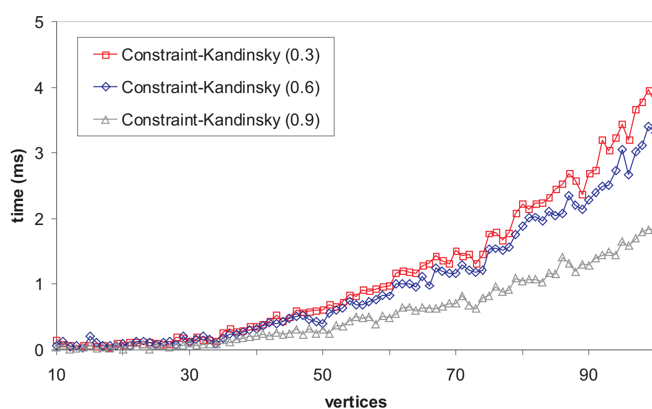


(c) number of bends

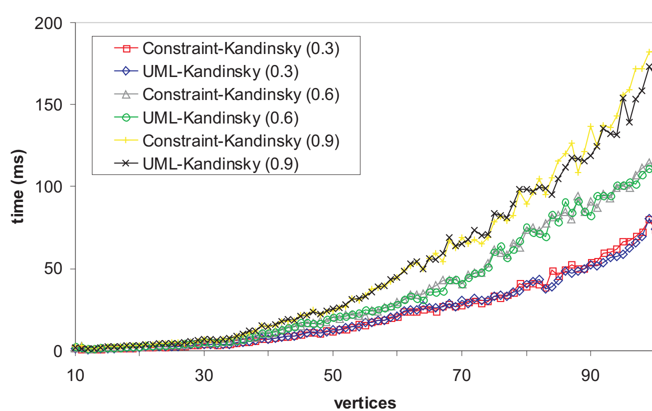
Figure 7.15: Test results for constraint FLOW.



(a) planarization time

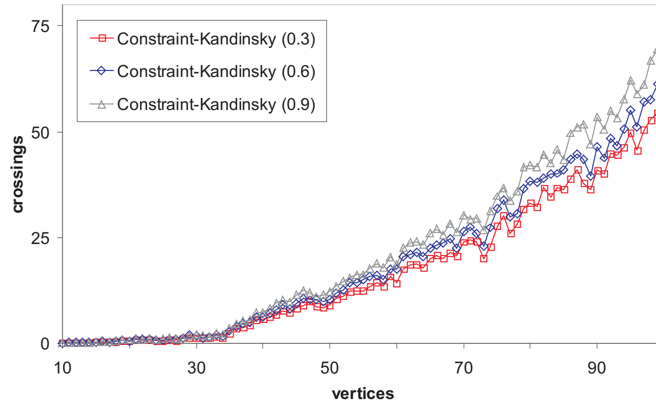


(b) rerouting time

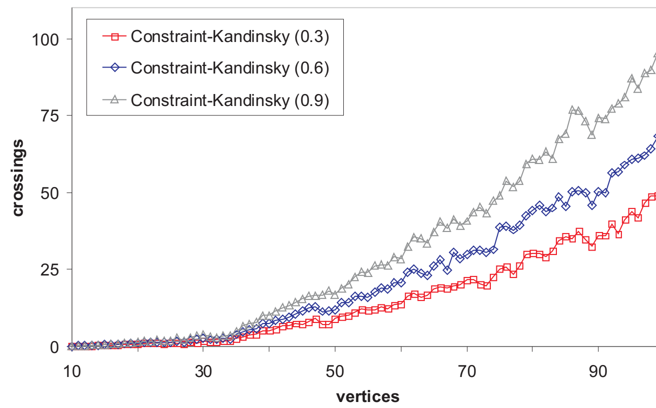


(c) orthogonalization time

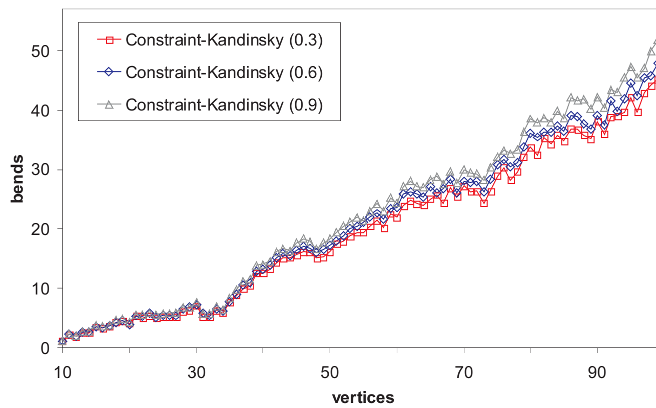
Figure 7.16: Runtimes for constraint FLOW.



(a) overall number of crossings

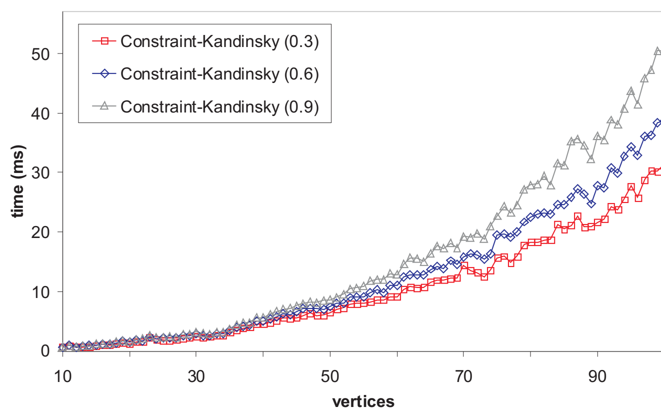


(b) crossings reduced by rerouting step

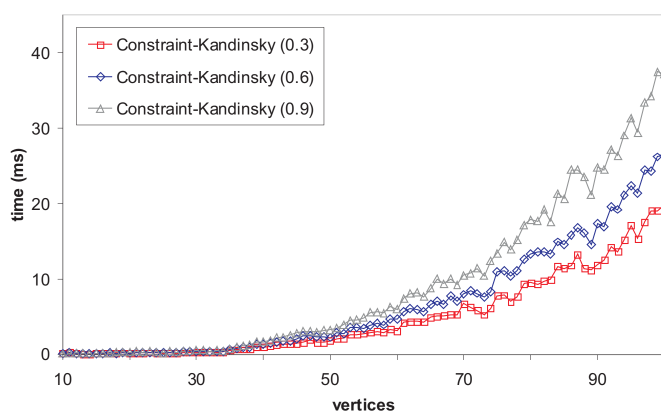


(c) number of bends

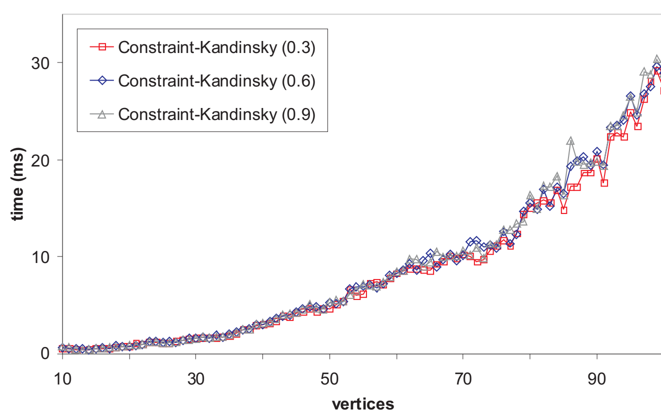
Figure 7.17: Test results for constraint BIMODAL.



(a) planarization time



(b) rerouting time



(c) orthogonalization time

Figure 7.18: Runtimes for constraint BIMODAL.

The results shown in Fig. 7.17 and Fig. 7.18 lead to the following observations:

- The number of crossings increases with the number of directed edges, i.e., edges that have to be embedded bimodally.
- The number of crossings reduced by the rerouting step increases with the number of directed edges. This is not surprising since in the initial drawing all directed edges point in the same direction (the layering is more restrictive for directed edges). However, we are allowed to reroute such edges.
- The number of bends slightly increases with the number of crossings.
- The results validate that constraint BIMODAL is less restrictive than constraint FLOW since it produces fewer crossings and bends. Note that for acyclic graphs, the number of crossings in the initial drawing is about the same for both approaches. However, for constraint BIMODAL we can reroute all edges, which significantly decreases the number of crossings.
- The runtime of the planarization is dominated by the rerouting step.
- The rerouting time (and thus the planarization time) increases with the number of reduced crossings. This can be attributed to the time needed for updating the data structure after finding a more suitable route for an edge. Furthermore, the size of the routing graph (and thus the rerouting time) increases with the number of crossings in the initial drawing (the drawing calculated with Sugiyama's algorithm).
- The orthogonalization times for the different configurations do not differ significantly. We attribute this to the similar size of the planarized graphs.

7.3.3.3 Cluster Drawings

Below, we state the results for constraint CLUSTER. We tested `ConstraintKandinsky` for five different configurations "(a/b)" varying in the number of compound vertices as well as the number of vertices assigned to clusters. For an input graph $G = (V, E)$, the first value a of a configuration states the number of compound vertices which is $|C| = a \cdot |V|$. The second value b states the number of vertices $x = (|C| + |V|) \cdot b$ assigned to compound vertices. The vertices are assigned to compound vertices as follows: First, for each compound vertex $c \in C$, we randomly choose a vertex of V and assign it to c. This guarantees that each cluster contains at least one vertex. Then, while the number of vertices assigned to compound vertices is less than x, we randomly choose an unassigned vertex v of $C \cup V$ and assign

it to a randomly chosen vertex c of C (if c is not already a successor of v with respect to the inclusion tree). Note that the crossing number does not include crossings of common edges with edges representing the cluster regions.

The results shown in Fig. 7.19 and Fig. 7.20 lead to the following observations:

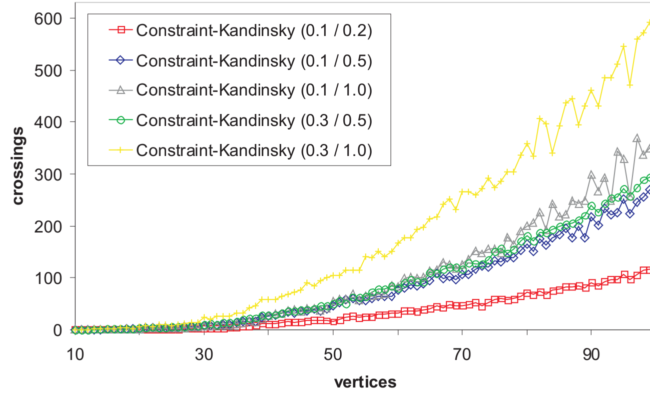
- The number of crossings increases with the number of clusters as well as the number of vertices assigned to clusters.
- The number of bends increases with the number of crossings.
- The planarization time is clearly dominated by the rerouting step and increases with an increasing number of crossings in the initial drawing.
- The rerouting time increases with the number of reduced crossings.
- The orthogonalization time increases with the number of crossings.

7.3.3.4 Partitioned Drawings

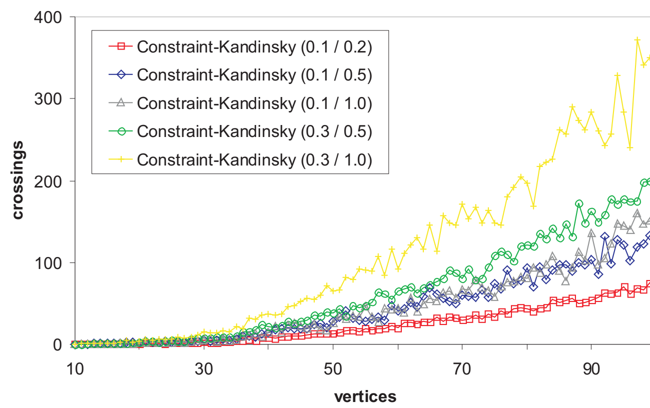
In our experiments for constraint PARTITION, we consider two-dimensional as well as one-dimensional partitions (swimlanes). We tested 8 configurations of partitions with different number of columns and rows (denoted with "*column* × *row* partition"). For the one-dimensional partitions, we test a configuration with ("opt") and one without an optimized swimlane order, calculated as described in Section 4.4.4. For each configuration the vertices are randomly assigned to the partition cells. Note that the crossing number does not include crossings of common edges with edges of the partition grid graph.

The results shown in Fig. 7.21 and Fig. 7.22 lead to the following observations:

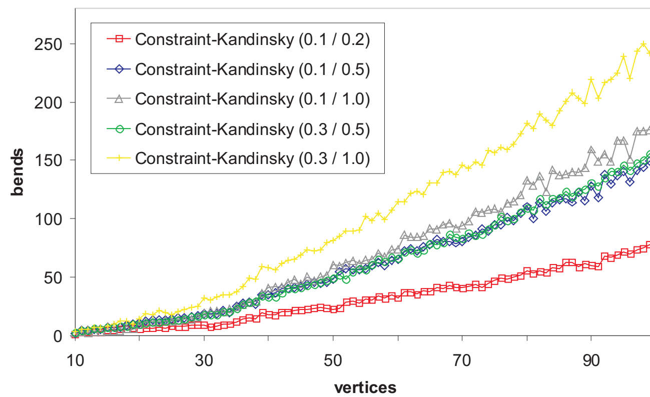
- The number of crossings increases with the number of partition columns and rows.
- The overall number of crossings does not indicate a positive effect of the swimlane order optimization heuristic. However, the number of crossings reduced during the rerouting step is higher for the configurations that do not use the optimization strategy. Hence, the number of crossings in the initial drawing is lower when we use it. Thus, applying the optimization is especially useful if we also have upward edges which cannot be rerouted.
- The rerouting step reduces the number of crossings by up to 50%.
- The number of bends increases with the number of crossings.



(a) overall number of crossings

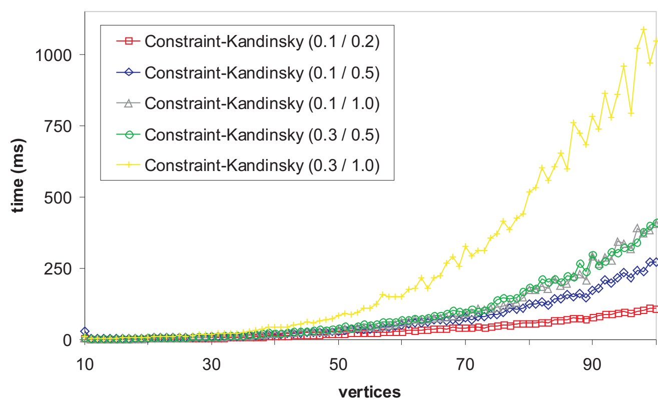


(b) crossings reduced by rerouting step

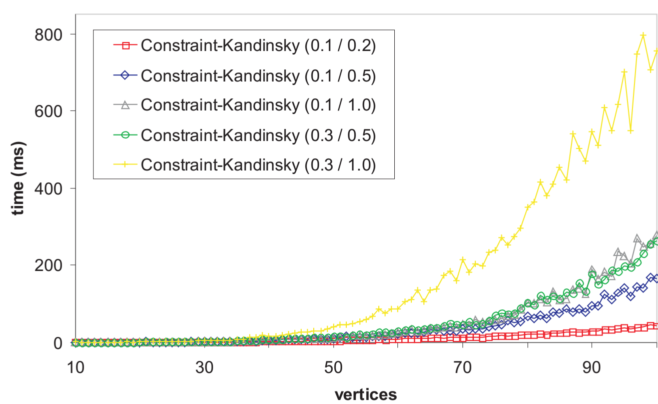


(c) number of bends

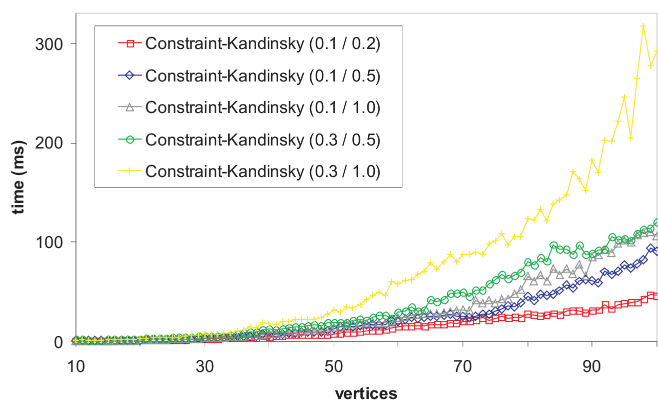
Figure 7.19: Test results for constraint CLUSTER.



(a) planarization time

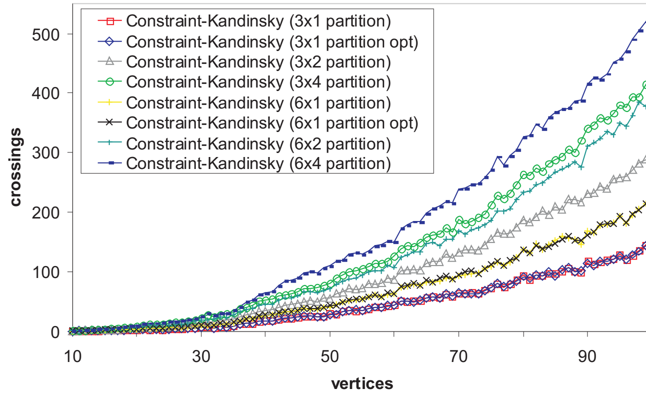


(b) rerouting time

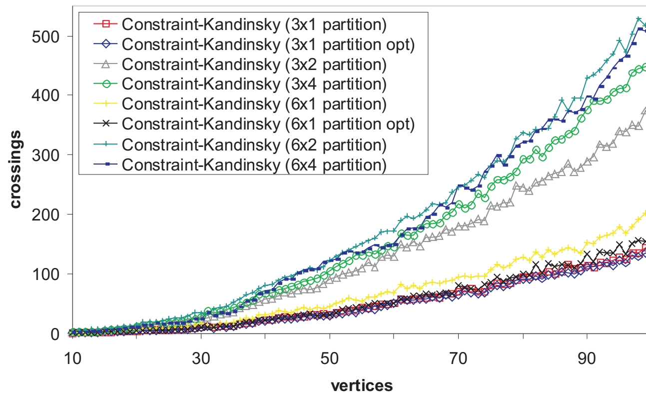


(c) orthogonalization time

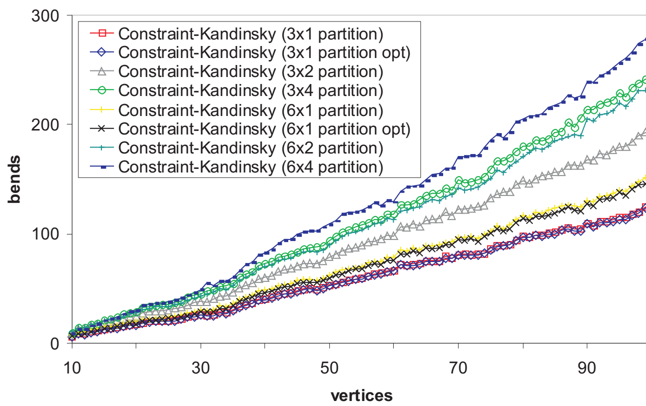
Figure 7.20: Runtimes for constraint CLUSTER.



(a) overall number of crossings

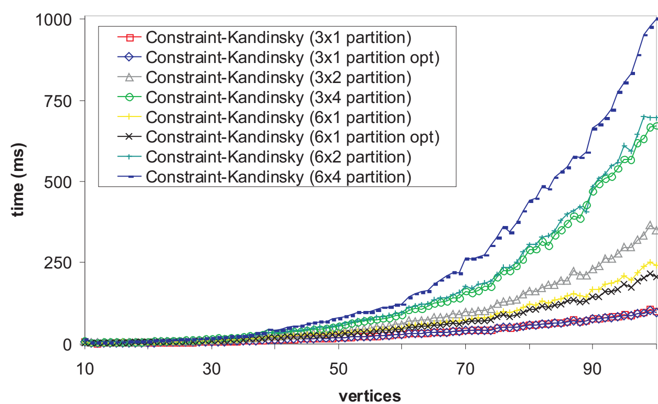


(b) crossings reduced by rerouting step

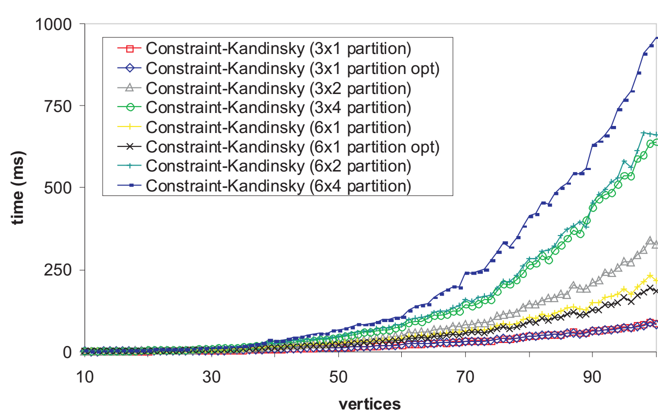


(c) number of bends

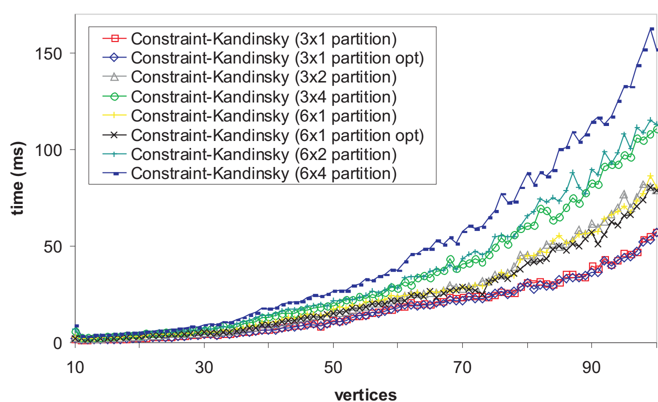
Figure 7.21: Test results for constraint PARTITION.



(a) planarization time



(b) rerouting time



(c) orthogonalization time

Figure 7.22: Runtimes for constraint PARTITION.

- The planarization time is clearly dominated by the rerouting step and increases with an increasing number of crossings in the initial drawing.
- The rerouting time increases with the number of reduced crossings.
- The orthogonalization time increases with the number of crossings.

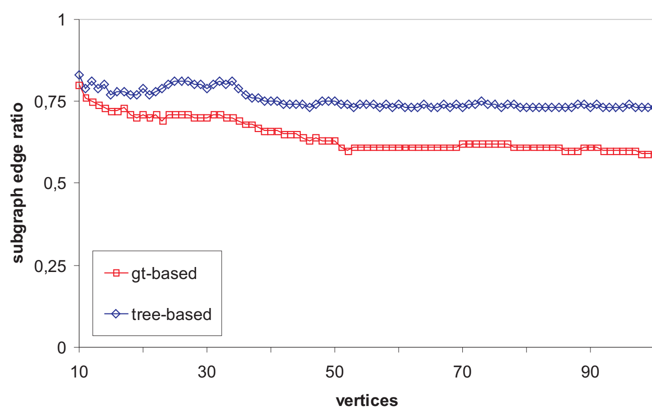
7.3.3.5 Port Constraint Preserving Drawings

Below, we compare different planarization and orthogonalization approaches that include port/side constraints. In the first experiment, we compare the `Constraint-Kandinsky` planarization with the alternative planarization approaches described in Section 6.2. We, therefore, independently consider the `sc`, `pc` and `mc` scenarios and measure the overall number of crossings, the number of edges in the planar subgraph (i.e., the number of these edges divided by the overall number of edges) as well as the overall runtime of the planarization. Note that for all alternative planarization approaches we use a modified shortest path routing for inserting the remaining edges.

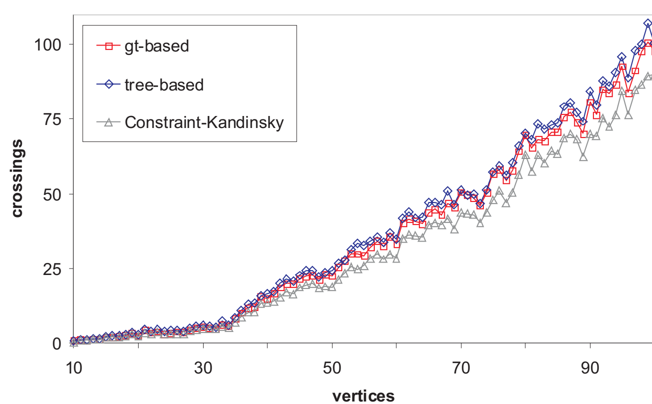
`sc` scenario Recall that in the `sc` scenario each edge has side constraints on both endpoints. Hence, for each endpoint of each edge we randomly choose one of the four possible sides. We compare the results of the `Constraint-Kandinsky` planarization to the results produced with the GT-based and the tree-based planarizations. As shown in Fig. 7.23(b), the `Constraint-Kandinsky` planarization produces slightly fewer crossings than the other approaches. However, to obtain this number of crossings, its rerouting step has to remove a lot of crossings, which results in a significantly higher runtime compared to the two other approaches (Fig. 7.23(c)). Hence, when we only consider constraint PORT/SIDE, approaches that are based on calculating a maximum planar subgraph may be superior to our approach.

Surprisingly, the planar subgraphs produced by the tree-based approach have more edges than the subgraphs produced by the GT-based approach (Fig. 7.23(a)). We attribute this to the relatively small density of the Rome graph collection. When the density is higher there are more candidates during the calculation of the maximum independent sets, which will improve the results of the GT-based planarization.

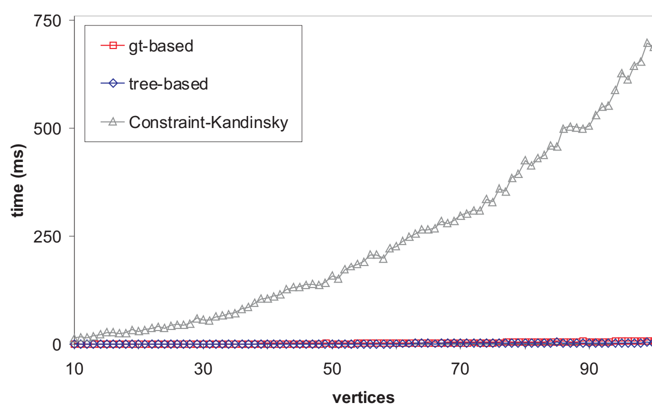
`pc` scenario In the `pc` scenario each edge has port constraints on both endpoints. For each endpoint of each edge we randomly choose one of the $8\kappa - 4$ possible pins. For the experiments we choose $\kappa = 10$. We compare the results of the `Constraint-Kandinsky` planarization to the results produced with the successive planarity testing approach and the tree-based planarization. As shown in Fig. 7.24(b), the number of crossings for all these approaches does not differ significantly. The successive planarity testing approach produces larger planar subgraphs than the tree-based planarization



(a) edges in planar subgraph



(b) overall number of crossings



(c) planarization time

Figure 7.23: Planarization results for constraint PORT/SIDE in the `sc` scenario.

(Fig. 7.24(a)). This is not surprising since it always calculates a maximal planar subgraph (not to be confused with a maximum planar subgraph).

For all approaches, the number of crossings in the `pc` scenario was only slightly higher than for the `sc` scenario (recall that the `pc` scenario is more restrictive). We attribute this to the relatively small density of the Rome graph collection, i.e., the average degree of a vertex is below three. Hence, there may be several vertex sides that are associated with only one edge. In this case, the port constraints can be handled like side constraints.

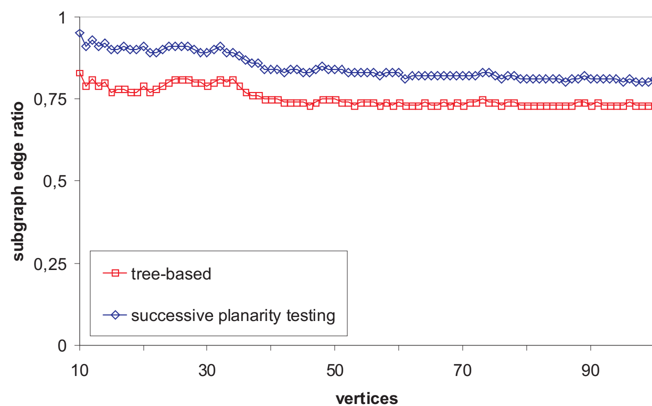
`mc` scenario First, for each endpoint of each edge we randomly choose, if the endpoint has a port, a side constraint or no constraint (all with the same probability). The side/pin is randomly chosen in the same way as for the `sc` and `pc` scenarios, respectively. We compare the results of the `Constraint-Kandinsky` planarization to the results produced with the tree-based planarization. As shown in Fig. 7.25(b), the `Constraint-Kandinsky` planarization produces slightly fewer crossings. Compared to the other scenarios, the number of crossings is fewer because about one third of the edges' endpoints have no constraints (the edge routing is less restrictive for these edges).

For the `mc` scenario we also test the different orthogonalization approaches, i.e., the skeleton-based network flow approach described in Section 5.4 (called "skeleton") and the network flow approach without fixing a skeleton (called "no-skeleton") described in Section 6.3.2. Both approaches are applied to the same port constraint preserving embedding. Surprisingly, the number of bends produced by the skeleton-based approach was significantly higher than that of the no-skeleton approach (Fig. 7.26(b)). We mainly attribute this to an unsuitable handling of the orientation problem. Recall that we insert up to $|V| - 1$ additional edges, i.e., skeleton edges, to fix the orientation of the vertices. Of course, this can lead to artifacts in the resulting drawings. The skeleton-based approach has a higher runtime since the additional skeleton edges as well as angle- and bend-constraints enlarge the Kandinsky network.

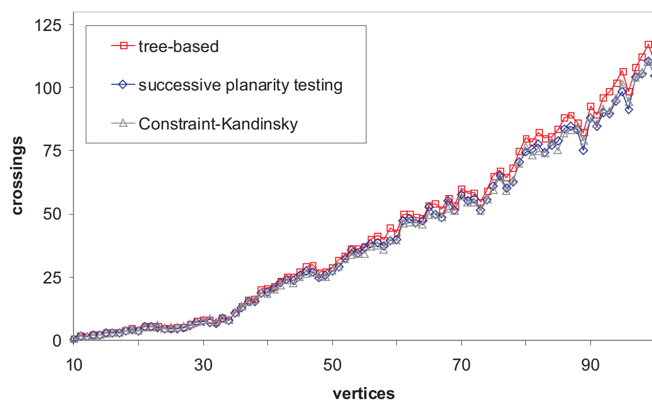
We also state the results obtained with the `odevs` approach described in Section 6.1.2. As shown in Fig. 7.26(c), the approach is very fast and produces significantly fewer bends. However, the number of crossings was extremely high compared to the two other orthogonalization approaches (Fig. 7.26(a)). Hence, we think that in order to use the `odevs` approach in practice we have to further reduce the number of crossings.

7.3.3.6 Summary

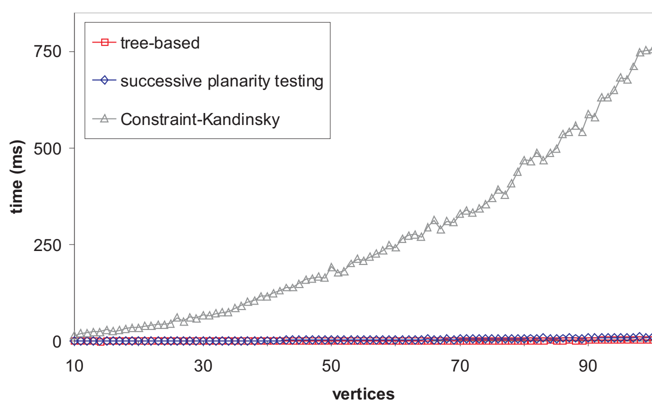
From the above results for `Constraint-Kandinsky`, we draw the following conclusions:



(a) edges in planar subgraph

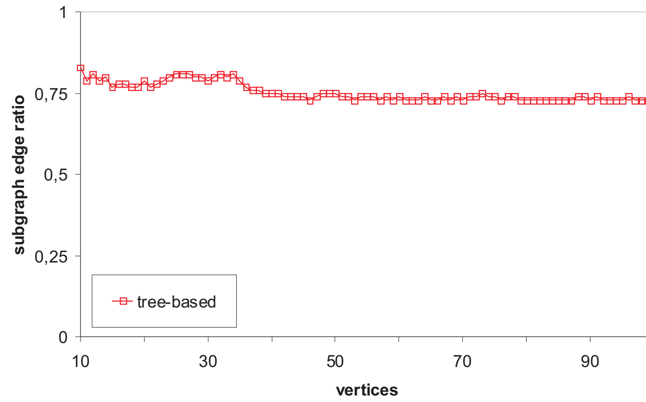


(b) overall number of crossings

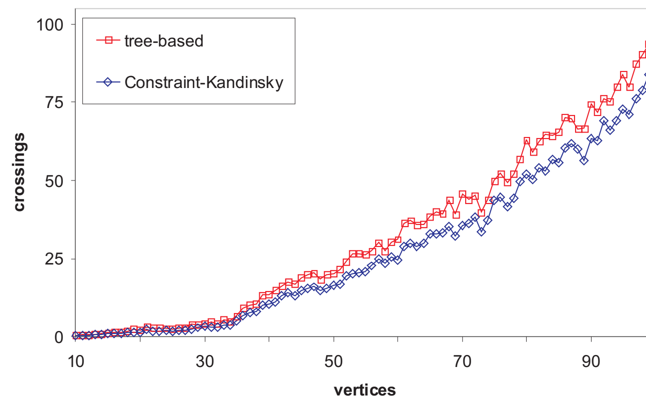


(c) planarization time

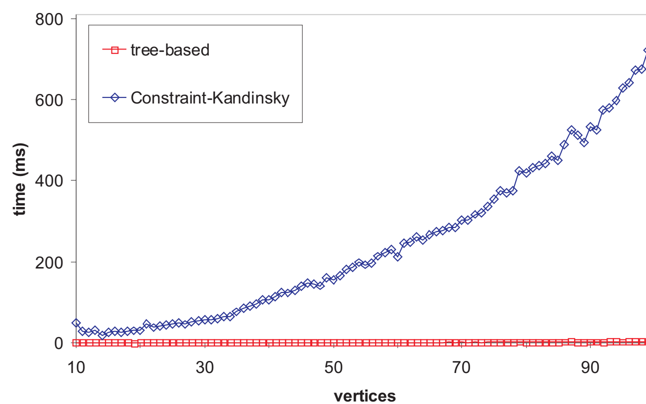
Figure 7.24: Planarization results for constraint PORT/SIDE in the `pc` scenario.



(a) edges in planar subgraph

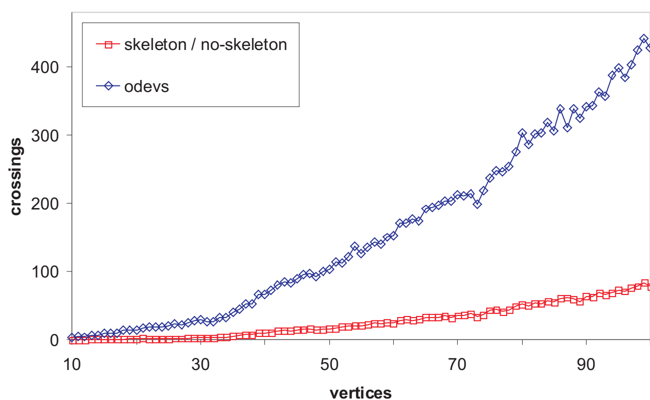


(b) overall number of crossings

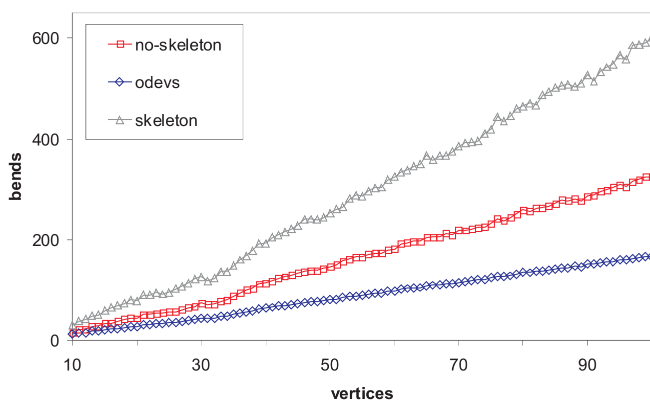


(c) planarization time

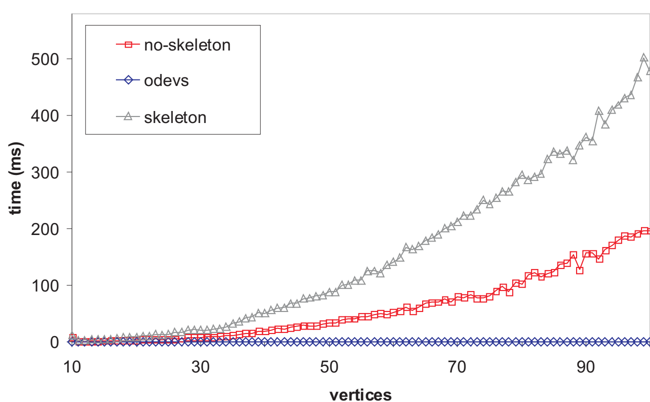
Figure 7.25: Planarization results for constraint PORT/SIDE in the `mc` scenario.



(a) overall number of crossings



(b) number of bends



(c) orthogonalization time

Figure 7.26: Orthogonalization results for constraint PORT/SIDE in the `mc` scenario.

- `Constraint-Kandinsky` is fast. In our experimental setting, it never exceeds a runtime of 1.5 seconds.

- The rerouting step always removes a lot of crossings (up to 50%). Hence, regarding aesthetic CROSSING, our TSM-based approach is clearly superior to Sugiyama-based approaches. Recall that our initial drawing is derived from a state-of-the-art Sugiyama implementation.

- The orthogonalization produces satisfying results in most cases. The number of bends for constraint PORT/SIDE is quite high. We attribute this to an unsuitable handling of the orientation problem.

CHAPTER 8

Conclusion

In this work, we addressed different drawing constraints which appear in diagrams used in application areas like software engineering, database modeling and VLSI design. We presented `Constraint-Kandinsky`, a new automatic layout algorithm which is based on the topology-shape-metrics approach and is able to consider all these constraints at the same time. `Constraint-Kandinsky` can be applied to several diagram types, e.g., UML class diagrams, entity-relationship diagrams, business process diagrams and many more. In our usability study we demonstrated how to apply `Constraint-Kandinsky` to UML activity diagrams. Furthermore, we presented a visualization method for execution graphs of parallel computations.

The experiments reveal the strengths and weaknesses of our layout approach. The results from our new planarization framework are especially convincing. In our experimental setting, `Constraint-Kandinsky` never exceeds a runtime of 1.5 seconds. For constraint PORT/SIDE, the handling of the orientation problem often leads to artifacts in the resulting drawings, i.e., edge routes with unnecessary bends. Nevertheless, we think that `Constraint-Kandinsky` is well suited for graphs of medium size and lower density (≤ 100 vertices, ≤ 2 density), which is sufficient for most diagrams used in the aforementioned applications. Our results prove that for these graphs the layout capabilities of TSM-based approaches regarding drawing constraints are competitive with those of layered and force-directed approaches. For larger graphs, the results produced by TSM-based approaches often suffer from phase arrangement, i.e., minimizing the number of crossings and bends is more important than minimizing the required drawing area. This strategy often leads to unsatisfying results for larger graphs since the drawing area is not used efficiently. For these graphs it is often desirable to switch to alternative layout approaches, e.g., to Sugiyama's approach, which we used for the visualization of execution graphs. Besides the size of the input graphs, the layout quality and runtime strongly depend on the number of constraints that have to be satisfied. However, this behavior is not surprising and we do not attribute it to a weakness in our approach.

In the following, we give a short overview of the main results presented in this work followed by a discussion about possible directions for future research.

8.1 Results

The main results presented in this work are the following:

- We reviewed different important drawing constraints which appear in several practical applications. These constraints restrict the cyclic order of edges around the vertices (BIMODAL and PORT/SIDE), group related vertices (CLUSTER and PARTITION) and draw edges monotonically in a prescribed direction (FLOW). Besides a formal description of these constraints, we also state important theoretical results as well as practical approaches carried out so far. Furthermore, for constraint PARTITION we showed that the corresponding planarity testing problem can be solved in linear time.

- We presented a generic planarization framework which is able to simultaneously include all the aforementioned constraints. It is based on a combination of the layered drawing approach of Sugiyama and a common rerouting strategy. As shown in this work, our framework is highly adaptable, robust and well suited for calculating planar embeddings subject to our set of drawing constraints.

- We presented `Fast-Sugiyama`, a fast implementation of the popular layered drawing approach of Sugiyama which produces drawings in the linear segments model. By a conceptually simple new technique, we are able to keep the number of dummy vertices and edges linear in the size of the input graph without increasing the number of crossings. Thus, we reduce the worst-case time complexity from $O((|V||E|)\log|E|)$ to $O((|V|+|E|)\log|E|)$ and the space consumption from $O(|V||E|)$ to $O(|V|+|E|)$.

- We described the necessary extensions for incorporating the different drawing constraints into the orthogonalization phase. Furthermore, we gave a brief description of how to handle self-loops as well as labels of graph elements.

- We presented alternative approaches for handling port and side constraints. This includes different planarization approaches which are based on common planarization strategies as well as two further orthogonalization approaches. We also introduced the `odevs` drawing model and showed that producing bend-minimum port constraint preserving drawings in this model is NP-hard. Based on this model we

developed a linear-time approach for handling port/side constraints which produces less than or equal to $3|E|$ bends and at most 4 bends per edge.

- We described how to apply `Constraint-Kandinsky` for drawing UML activity diagrams. We identified requirements and aesthetics for such diagrams and reviewed some popular UML tools with respect to their automatic layout capabilities.

- We presented an advanced visualization method for execution graphs of parallel computations which is based on our `Fast-Sugiyama` implementation. Our visualization is capable of concisely depicting several performance-relevant properties of task-parallel computations. The low runtime complexity of our visualization method enables highly interactive development workflows. We also studied the integration of our visualization method into a comprehensive development environment and realized a visually steered workbench for minimizing parallel overhead of irregular task-parallel computations.

8.2 Directions for Future Research

In this section we discuss possible directions for future research. This includes improvements of the presented algorithms, general extensions to include further drawing requirements as well as solving open theoretical questions. The following items depict promising improvements for our layout algorithms:

- During the planarization phase, we only reroute non-upward edges. As our experiments have shown, rerouting has a great impact on the crossing number. Hence, we can further improve quality by also rerouting upward edges. The approach for routing upward edges described in [69] can provide a suitable starting point for this purpose. Furthermore, the experiments point out that the runtime of the planarization phase is dominated by the rerouting step. We think that we can speed up the rerouting by using a more sophisticated data structure that allows faster updates of the planar embedding and the dual routing graph after changing the route of an edge.

- The experiments also indicate that a more adequate handling of the orientation problem may lead to a significant improvement of the layout quality. The current handling often leads to unsatisfying edge routes with many bends, especially for port/side constraints.

- A more appropriate handling of the `straight-line edge assignment issue` and the `no pin left issue` (both introduced in Section 5.4)

can further reduce the number of bends in drawings with port/side constraints.

- For graphs with port/side constraints, our `odevs`-based approach produces drawings with fewer bends than drawings produced by TSM-based approaches. However, to make the `odevs` approach more attractive for practical use, the number of crossings has to be further reduced. We believe that there is still some room for such an improvement. In addition, in practice it is more suitable to not restrict vertices to lie on general positions.

While our layout algorithms already cover important requirements demanded by several diagram types, there are still some relevant requirements left for future work:

- Future work might comprise an interactive version of `Constraint-Kandinsky` where the user can continuously change a diagram by adding or removing graph elements. When we apply our current approach to the modified graph structure, the resulting drawing can be significantly different to the previous one, even for small updates. The main objective of an interactive layout approach is to preserve the user's mental map [57] of a drawing, i.e., to produce a readable drawing that minimizes changes to the previous diagram. Here, the concepts of sketch-driven orthogonal layout developed in [26] fit into our framework. Furthermore, there are approaches for including interactivity in Sugiyama's approach; see, e.g., [20, 117]. However, these concepts require major modifications and extensions in order to be applicable to our set of drawing constraints.

- Another issue that often arises in interactive applications are "partial layouts", i.e., layouts where some of the vertices have fixed coordinates and some of the edges have specified routing points. The layout algorithm has to integrate the non-fixed graph elements into the layout of the fixed elements such that the resulting diagram is readable and the coordinates of the fixed elements do not change. Due to the fixed order of the phases of the TSM approach – coordinates first appear in the last phase – including this kind of constraint seems to be very difficult.

- To reduce complexity, it is often desirable to reorganize larger diagrams into several smaller ones [3]. Basically, there are two different strategies for how this task can be performed. We can take the input graph, subdivide it into several subgraphs according to semantic properties and then apply the layout algorithm to each subgraph separately. The second strategy applies the algorithm to the whole input graph and subdivides the resulting diagram into smaller pieces. Here,

the objective is to produce pieces that do not exceed a given size and that minimize the number of cut edges. An approach realizing this strategy is sketched in [60].

- To apply port/side constraints to a wider field of applications, we can include enhanced requirements like multi-candidates port/side constraints, limited number of pins and pin sharing as proposed in Section 6.4.

There are several open theoretical questions which we encountered in this work. Solving them may lead to new or improved algorithms. We think the most interesting questions are the following:

- It remains unknown whether bend minimization in the Kandinsky model is NP-hard. A further interesting question is whether the minimization problem can be solved if all edges are drawn upward. Note that this restricts the edge routes in a way such that the problem may become easier.

- There are several publications about c-planarity testing for clustered graphs; see [38, 39, 40, 47, 73, 91], however, the complexity of the problem is still unknown.

- Up to now, there has been no planarity test for graphs with port/side constraints, but the approach of Gutwenger et al. [92] offers a step towards such a test. It cannot, however, handle vertices which have both incident edges with and without constraints.

- Another open problem related to port/side constraints is whether crossing minimization in the `pc` scenario is still NP-hard. An alternative formulation of this problem is: Given a general graph $G = (V, E)$ and the cyclic order of the edges around each vertex, is crossing minimization subject to the given cyclic order still NP-hard?

In our opinion, working on the above topics is important to further improve the quality and benefits of automatic layout approaches as well as to produce adequate visualizations for complex and interactive applications.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] M. Ajtai, V. Chvátal, M. M. Newborn and E. Szemerédi. Crossing-free subgraphs. In *Theory and Practice of Combinatorics*, volume 60 of *North-Holland Math. Stud.*, pages 9–12. North-Holland, Amsterdam, 1982.

[3] S. W. Ambler. *The Elements of UML 2.0 Style*. Cambridge University Press, 2005.

[4] T. Asano, H. Imai and A. Mukaiyama. Finding a maximum weight independent set of a circle graph. *IEICE Tranactions*, E74:681–683, 1991.

[5] D. Auber. Tulip - a huge graph visualization framework. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 105–126. Springer-Verlag, 2003.

[6] W. Barth, M. Jünger and P. Mutzel. Simple and efficient bilayer cross counting. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 130–141. Springer-Verlag, 2002.

[7] M. Becker and I. Rojas. A graph layout algorithm for drawing metabolic pathways. *Bioinformatics*, 17(5):461–7, May 2001.

[8] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[9] P. Bertolazzi, G. Di Battista and W. Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE Transactions on Computers*, 49(8):826–840, 2000.

[10] P. Bertolazzi, G. Di Battista and W. Didimo. Quasi-upward planarity. *Algorithmica*, 32(3):474–506, 2002.

[11] P. Bertolazzi, G. Di Battista, C. Mannino and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, 27(1):132–169, 1998.

[12] T. C. Biedl. *Orthogonal Graph Visualization: The Three-Phase Method, With Applications*. PhD thesis, RUTCOR, Rutgers University, May 1997.

[13] T. C. Biedl, B. Madden and I. G. Tollis. The three-phase method: A unified approach to orthogonal graph drawing. *International Journal of Computational Geometry and Applications*, 10(6):553–580, 2000.

[14] W. Blochinger, M. Kaufmann and M. Siebenhaller. Visualizing structural properties of irregular parallel computations. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software visualization*, pages 125–134. ACM Press, 2005.

[15] W. Blochinger, M. Kaufmann and M. Siebenhaller. Visualization aided performance tuning of irregular task-parallel computations. *Information Visualization*, 5(2):81–94, 2006.

[16] W. Blochinger and W. Küchlin. The design of an API for strict multithreading in C++. In H. Kosch, L. Böszörményi and H. Hellwagner, editors, *Proc. of 9th Intl. Conf. Euro-Par 2003*, number 2790 in Lecture Notes in Computer Science, pages 722–731. Springer-Verlag, 2003.

[17] W. Blochinger, W. Küchlin, C. Ludwig and A. Weber. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

[18] W. Blochinger, C. Sinz and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

[19] R. D. Blumofe and C. E. Leiserson. Space efficient scheduling of multithreaded computations. In *Proc. of the Twenty Fifth Annual ACM Symp. on Theory of Computing*, pages 362–371, San Diego, CA, May 1993.

[20] K.-F. Böhringer and F. N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 43–51. ACM Press, 1990.

[21] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[22] F.-J. Brandenburg, M. Himsolt and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Proceedings of the International Symposium on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 1996.

[23] U. Brandes. Drawing on physical analogies. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, number 2025 in Lecture Notes in Computer Science, pages 71–86. Springer-Verlag, Berlin, Germany, 2001.

[24] U. Brandes, S. Cornelsen, C. Fieß and D. Wagner. How to draw the minimum cuts of a planar graph. *Comput. Geom.*, 29(2):117–133, 2004.

[25] U. Brandes, T. Dwyer and F. Schreiber. Visualizing related metabolic pathways in two and a half dimensions. In G. Liotta, editor, *Graph Drawing, Perugia, 2003*, pages 111–122. Springer-Verlag, 2004.

[26] U. Brandes, M. Eiglsperger, M. Kaufmann and D. Wagner. Sketch-driven orthogonal graph drawing. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2002.

[27] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In *Proceedings of the 9th Symposium on Graph Drawing (GD '01)*, volume 2265 of *Lecture Notes in Computer Science*, pages 31–44. Springer-Verlag, 2002.

[28] S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia and L. Vismara. Turn-regularity and optimal area drawings for orthogonal representations. *Computational Geometry: Theory and Applications*, 16(1):53–93, 2000.

[29] R. Brockenauer and S. Cornelsen. Drawing clusters and hierarchies. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, number 2025 in Lecture Notes in Computer Science, pages 193–227. Springer-Verlag, Berlin, Germany, 2001.

[30] I. Bruß and A. Frick. Fast interactive 3-d graph visualization. In *Proceedings of the International Symposium on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 1996.

[31] T. Bubeck, M. Hiller, W. Küchlin and W. Rosenstiel. Distributed symbolic computation with DTS. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 231–248. Springer-Verlag, 1995.

[32] C. Buchheim, M. Jünger, A. Menze and M. Percan. Bimodal crossing minimization. In *COCOON*, volume 4112 of *Lecture Notes in Computer Science*, pages 497–506. Springer-Verlag, 2006.

[33] R. Castello, R. Mili and I. G. Tollis. Automatic layout of statecharts. *Software – Practice and Experience*, 32(1):25–55, 2002.

[34] R. Castello, R. Mili and I. G. Tollis. A framework for the static and interactive visualization of statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002.

[35] R. Cimikowski. An analysis of heuristics for the maximum planar subgraph problem. In *Proceedings of the 6th ACM-SIAM Symposium of Discrete Algorithms*, pages 322–331, 1995.

[36] M. K. Coleman and D. S. Parker. Aesthetics-based graph layout for human consumption. *Software – Practice and Experience*, 26(12):1415–1438, 1996.

[37] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[38] S. Cornelsen and D. Wagner. Completely connected clustered graphs. *Journal of Discrete Algorithms*, 4(2):313–323, 2006.

[39] P. F. Cortese, G. Di Battista, F. Frati, M. Patrignani and M. Pizzonia. C-planarity of c-connected clustered graphs. *Journal of Graph Algorithms and Applications*, 12(2):225–262, Nov 2008.

[40] E. Dahlhaus. A linear time algorithm to recognize clustered graphs and its parallelization. In C. L. Lucchesi and A. V. Moura, editors, *LATIN*, volume 1380 of *Lecture Notes in Computer Science*, pages 239–248. Springer-Verlag, 1998.

[41] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[42] C. Demetrescu and I. Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters*, 86(3):129–136, 2003.

[43] G. Di Battista, W. Didimo and A. Marcandalli. Planarization of clustered graphs (extended abstract). In *Proceedings of the 9th Symposium on Graph Drawing (GD '01)*, volume 2265 of *Lecture Notes in Computer Science*, pages 60–74. Springer-Verlag, 2002.

[44] G. Di Battista, W. Didimo, M. Patrignani and M. Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size. In

J. Kratochvil, editor, *Proceedings of the 7th International Symposium on Graph Drawing (GD'99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 297–310. Springer-Verlag, 1999.

[45] G. Di Battista, W. Didimo, M. Patrignani and M. Pizzonia. Drawing database schemas. *Software–Practice and Experience*, 32(11):1065–1098, 2002.

[46] G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.

[47] G. Di Battista and F. Frati. Efficient c-planarity testing for embedded flat clustered graphs with small faces. *Journal of Graph Algorithms and Applications*, 13(3):349–378, 2009.

[48] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7:303–325, 1997.

[49] G. Di Battista, G. Liotta and F. Vargiu. Spirality and optimal orthogonal drawings. *SIAM Journal on Computing*, 27(6):1764–1811, 1998.

[50] W. Didimo and G. Liotta. Computing orthogonal drawings in a variable embedding setting. In K.-Y. Chwa and O. H. Ibarra, editors, *Proceedings of the 9th Annual International Symposium on Algorithms and Computation (ISAAC'98)*, volume 1533 of *Lecture Notes in Computer Science*, pages 79–88. Springer-Verlag, 1998.

[51] R. Diestel. *Graph Theory (Graduate Texts in Mathematics).* Springer-Verlag, third edition, August 2005.

[52] P. Duchet, Y. O. Hamidoune, M. L. Vergnas and H. Meyniel. Representing a planar graph by vertical lines joining different levels. *Discrete Mathematics*, 46(3):319–321, 1983.

[53] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[54] P. Eades, R. F. Cohen and M. L. Huang. Online animated graph drawing for web navigation. In G. D. Battista, editor, *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 330–335. Springer-Verlag, 1997.

[55] P. Eades, Q.-W. Feng and H. Nagamochi. Drawing clustered graphs on an orthogonal grid. *Journal of Graph Algorithms and Applications*, 3(4):3–29, 1999.

[56] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combin.*, 21.A:89–98, 1986.

[57] P. Eades, W. Lai, K. Misue and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics '91*, pages 24–33, 1991.

[58] P. Eades, X. Lin and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[59] P. Eades and N. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[60] P. Effinger, M. Kaufmann and M. Siebenhaller. An interactive layout tool for BPMN. *IEEE Conference on Commerce and Enterprise Computing*, pages 399–406, 2009.

[61] H. Eichelberger. Evaluation-report on the layout facilities of UML tools. Technical Report 298, Institut für Informatik, Würzburg University, July 2002.

[62] H. Eichelberger. SugiBib. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *Lecture Notes in Computer Science*, pages 467–468. Springer-Verlag, 2002.

[63] H. Eichelberger. *Aesthetics and automatic layout of UML class diagrams*. PhD thesis, Universität Würzburg, Am Hubland, D-97074 Würzburg, 2005.

[64] M. Eiglsperger. *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach*. PhD thesis, Universität Tübingen, 2003.

[65] M. Eiglsperger, S. Fekete and G. Klau. Orthogonal graph drawing. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, Lecture Notes in Computer Science Tutorial, pages 121–171. Springer-Verlag, 2001.

[66] M. Eiglsperger, U. Fößmeier and M. Kaufmann. Orthogonal graph drawing with constraints. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 3–11, 2000.

[67] M. Eiglsperger, C. Gutwenger, M. Kaufmann, J. Kupke, M. Jünger, S. Leipert, K. Klein, P. Mutzel and M. Siebenhaller. Automatic layout of UML class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004.

[68] M. Eiglsperger and M. Kaufmann. Fast compaction for orthogonal drawings with vertices of prescribed size. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2265, pages 124–138. Springer-Verlag, 2002.

[69] M. Eiglsperger, M. Kaufmann and F. Eppinger. An approach for mixed upward planarization. *Journal of Graph Algorithms and Applications*, 7(2):203–220, 2003.

[70] M. Eiglsperger, M. Kaufmann and M. Siebenhaller. A topology-shape-metrics approach for the automatic layout of UML class diagram. In *Proceedings of ACM 2003 Symposium on Software Visualization, SoftVis 2003*, pages 189–198. ACM Press, 2003.

[71] M. Eiglsperger, M. Siebenhaller and M. Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. *Journal of Graph Algorithms and Applications*, 9(3):305–325, 2005.

[72] L. Faria, C. M. H. de Figueiredo and C. F. X. Mendonça. Splitting number is NP-complete. *Discrete Applied Mathematics*, 108(1):65–83, 2001.

[73] Q.-W. Feng, R. F. Cohen and P. Eades. Planarity for clustered graphs. In *Proceedings of the 3rd European Symposium on Algorithms (ESA'95)*, volume 979 of *Lecture Notes in Computer Science*, pages 213–226. Springer-Verlag, 1995.

[74] M. Forster. Applying crossing reduction strategies to layered compound graphs. In *Proceedings of the 10th Symposium on Graph Drawing (GD'02)*, volume 2528 of *Lecture Notes in Computer Science*, pages 276–284. Springer-Verlag, 2002.

[75] M. Forster. *Crossings in Clustered Level Graphs*. Dissertation, University of Passau, 2004.

[76] M. Forster. A fast and simple heuristic for constrained two-level crossing reduction. In *Proceedings of the 9th Symposium on Graph Drawing (GD'04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 206–216. Springer-Verlag, 2005.

[77] U. Fößmeier. *Orthogonale Visualisierungstechniken für Graphen*. PhD thesis, Universität Tübingen, 1997.

[78] U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In *Proceedings of the International Symposium on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 254–266. Springer-Verlag, 1996.

[79] A. Frick. Upper bounds on the number of hidden nodes in Sugiyama's algorithm. In *Proceedings of the 4th International Symposium on Graph Drawing (GD'96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 169–183. Springer-Verlag, 1997.

[80] A. Frick, A. Ludwig and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of the 2nd International Symposium on Graph Drawing (GD'94)*, volume 894 of *Lecture Notes in Computer Science*, pages 388–403, 1995.

[81] T. M. J. Fruchterman and E. M. Reingold. Graph-drawing by force-directed placement. *Software — Practice and Experience*, 21(11):1129–1164, 1991.

[82] E. Gansner, E. Koutsofios, S. North and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[83] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$–Completeness.* W.H. Freeman & Co, 1979.

[84] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312–316, 1983.

[85] M. R. Garey, D. S. Johnson and L. J. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[86] A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In *Proceedings of the 4th International Symposium on Graph Drawing (GD'96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 201–216, 1997.

[87] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM Journal on Computing*, 31(2):601–625, 2001.

[88] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 7–18. ACM Press, 1987.

[89] O. Goldschmidt and A. Takvorian. An efficient graph planarization two-phase heuristic. *Networks*, 24:69–73, 1994.

[90] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert and P. Mutzel. A new approach for visualizing UML class diagrams. In *SOFTVIS*, pages 179–188, 217–218, 2003.

[91] C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan and R. Weiskircher. Advances in c-planarity testing of clustered graphs. In S. G. Kobourov and M. T. Goodrich, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2002.

[92] C. Gutwenger, K. Klein and P. Mutzel. Planarity testing and optimal edge insertion with embedding constraints. In M. Kaufmann and D. Wagner, editors, *Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 126–137. Springer-Verlag, 2006.

[93] C. Gutwenger, P. Mutzel and R. Weiskircher. Inserting an edge into a planar graph. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 246–255. Society for Industrial and Applied Mathematics, 2001.

[94] W. He and K. Marriott. Constrained graph layout. *Constraints*, 3(4):289–314, 1998.

[95] P. Healy and N. Nikolov. How to layer a directed acyclic graph. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 2002.

[96] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[97] M. T. Heath, A. D. Malony and D. T. Rover. The visual display of parallel performance data. *IEEE Computer*, 28(11):21–28, 1995.

[98] P. Hliněný. Crossing number is hard for cubic graphs. *Journal of Combinatorial Theory Series B*, 96(4):455–471, 2006.

[99] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[100] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications (JGAA)*, 1(1):1–25, 1997.

[101] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[102] R. Karp. Reducibility among combinatorical problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[103] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[104] G. W. Klau, K. Klein and P. Mutzel. An experimental comparison of orthogonal compaction algorithms (extended abstract). In *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 2001.

[105] G. W. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report 98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[106] G. W. Klau and P. Mutzel. Optimal compaction of orthogonal grid drawings. In *Integer Programming and Combinatorial Optimization (IPCO'99)*, number 1610 in Lecture Notes in Computer Science, pages 304–319, 1999.

[107] E. Kraemer and J. Stasko. Creating an accurate portrayal of concurrent executions. *Concurrency*, 6(1):36–46, 1998.

[108] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[109] A. S. LaPaugh. *VLSI Layout Algorithms*, pages 23/1–23/26. Algorithms and Theory of Computation Handbook. CRC Press, Boca Raton, 1998.

[110] F. T. Leighton. *Complexity issues in VLSI: optimal layouts for the shuffle-exchange graph and other networks*. MIT Press, Cambridge, MA, USA, 1983.

[111] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Applicable Theory in Computer Science. Wiley-Teubner, 1990.

[112] D. Lüttke-Hüttmann. Knickminimales zeichnen 4-planarer clustergraphen. Master's thesis, Universität des Saarlandes, July 2000.

[113] A. D. Malony. Tools for parallel computing: A performance evaluation perspective. In J. Blazewicz, K. Ecker, B. Plateau and D. Trystram, editors, *Handbook on Parallel and Distributed Processing*, chapter 7. Springer-Verlag, 2000.

[114] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[115] W. Nagel and A. Arnold. Performance visualization of parallel programs - the PARvis environment. In *Proceedings 1994 Intel Supercomputer Users Group (ISUG) Conference*, 1994.

[116] W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[117] S. C. North. Incremental layout in dynadag. In *Proceedings of the International Symposium on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418. Springer-Verlag, 1996.

[118] Object Management Group (OMG). UML 2.2 superstructure specification (WWW document). `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/` (accessed July 2009).

[119] R. H. J. M. Otten and J. G. van Wijk. Graph representation in interactive layout design. In *IEEE Transactions on Circuits and Systems*, pages 914–918, 1978.

[120] J. Pach and G. Tóth. Graphs drawn with few crossings per edge. *Combinatorica*, 17(3):427–439, 1997.

[121] J. Pach and R. Wenger. Embedding planar graphs at fixed vertex locations. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 263–274. Springer-Verlag, 1998.

[122] M. Patrignani. On the complexity of orthogonal compaction. Technical Report RT–DIA–39–99, Dipartimento di Informatica e Automazione, Università degli Studi di Roma Tre, January 1999.

[123] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 1997.

[124] H. C. Purchase, R. F. Cohen and M. James. An experimental study of the basis for graph drawing algorithms. *ACM Journal of Experimental Algorithmics*, 2(4), 1997.

[125] K. H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[126] D. A. Reed and R. A. Aydt. Tools for performance tuning and debugging. In J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors, *Sourcebook of Parallel Computing*, chapter 15. Morgan Kaufmann, 2003.

[127] M. Resende and C. Ribeiro. A grasp for graph planarization. *Networks*, 29:173–189, 1997.

[128] G. Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, June 1996.

[129] G. Sander. Graph layout for applications in compiler construction. *Theoretical Computer Science*, 217(2):175–214, 1999.

[130] M. Sarrafzadeh and D. T. Lee. A new approach to topological via minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(8):890–900, 1989.

[131] F. Schreiber. *Visualisierung biochemischer Reaktionsnetze*. PhD thesis, Universität Passau, 2001.

[132] J. Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In G. DiBattista, editor, *Proc. Graph Drawing, 5th International Symposium, GD '97, Rome, Italy, September, 1997*, volume 1353. Springer-Verlag, 1997.

[133] S. Shavor, J. D'Anjou and S. Fairbrother. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2004.

[134] D. Shegogue and W. J. Zheng. Integration of the gene ontology into an object-oriented architecture. *BMC Bioinformatics*, 6:113, 2005.

[135] M. Siebenhaller. Partitioned drawings. In *Proceedings of the 14th Symposium on Graph Drawing (GD'06)*, volume 4372 of *Lecture Notes in Computer Science*, pages 252–257. Springer-Verlag, 2007.

[136] M. Siebenhaller and M. Kaufmann. Drawing activity diagrams. In *Proceedings of ACM 2006 Symposium on Software Visualization, SoftVis 2006*, pages 159–160. ACM Press, 2006.

[137] M. Siebenhaller and M. Kaufmann. Drawing activity diagrams. Technical Report WSI-2006-02, Wilhelm-Schickard-Institute, University of Tübingen, WSI, Sand 14, 72076 Tübingen, 2006.

[138] M. Siebenhaller and M. Kaufmann. Mixed upward planarization - fast and robust. In *Proceedings of the 13th Symposium on Graph Drawing*

(GD'05), volume 3843 of *Lecture Notes in Computer Science*, pages 522–523. Springer-Verlag, 2006.

[139] J. M. Six and I. G. Tollis. Automated visualization of process diagrams. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *Lecture Notes in Computer Science*, pages 45–59. Springer-Verlag, 2002.

[140] D. B. Skillicorn and D. Talia. Models and languages for parallel computing. *ACM Computing Surveys*, 30:123–169, 1998.

[141] D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[142] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[143] B. Steckelbach, T. Bubeck, U. Fössmeier, M. Kaufmann, M. Ritt and W. Rosenstiel. Visualization of parallel execution graphs. In *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*, pages 403–412. Springer-Verlag, 1998.

[144] K. Sugiyama and K. Misue. Visualisation of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, 1991.

[145] K. Sugiyama and K. Misue. Graph drawing by the magnetic spring model. *J. Vis. Lang. Comput.*, 6(3):217–231, 1995.

[146] K. Sugiyama, S. Tagawa and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, 1981.

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

[148] R. Tamassia, G. D. Battista and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, 1988.

[149] B. Topol, J. Stasko and V. Sunderam. PVaniM: A tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[150] L. Vismara, G. D. Battista, A. Garg, G. Liotta, R. Tamassia and F. Vargiu. Experimental studies on graph drawing algorithms. *Softw., Pract. Exper.*, 30(11):1235–1284, 2000.

[151] V. Waddle. Graph layout for displaying data structures. In J. Marks, editor, *Graph Drawing, Colonial Williamsburg, 2000*, pages 241–252. Springer-Verlag, 2001.

[152] V. Waddle and A. Malhotra. An E log E line crossing algorithm for levelled graphs. In *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 59–71. Springer-Verlag, 1999.

[153] F. Wagner and A. Wolff. A combinatorial framework for map labeling. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 316–331. Springer-Verlag, 1998.

[154] X. Wang and I. Miyamoto. Generating customized layouts. In *Proceedings of the International Symposium on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 504–515. Springer-Verlag, 1996.

[155] K. Wittenburg and L. Weitzman. Qualitative visualization of processes: Attributed graph layout and focusing techniques. In *Proceedings of the 4th International Symposium on Graph Drawing (GD'96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 401–408. Springer-Verlag, 1997.

[156] yWorks. yFiles - a java graph layout and visualization library (WWW document). `http://www.yworks.com` (accessed July 2009).