# Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach

**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. Markus Eiglsperger**
aus Wangen am Bodensee

**Tübingen**
**2003**

Tag der mündlichen Qualifikation: 26. November 2003
Dekan: Prof. Dr. Martin Hautzinger
1. Gutachter: Prof. Dr. Michael Kaufmann
2. Gutachter: Prof. Dr. Wolfgang Küchlin
3. Gutachter: Prof. Dr. Ulrik Brandes

# Zusammenfassung der Arbeit

In dieser Arbeit werden Verfahren zum automatischen und interaktiven Erstellen hochqualitativer Zeichnungen von UML Klassendiagrammen vorgestellt. Da Klassendiagramme auf Graphen zurückgeführt werden können, verwenden wir dazu Methoden und Techniken aus dem Bereich des Graphenzeichnens, und konzentrieren uns in dieser Arbeit insbesondere auf das „Topology-Shape-Metrics“ Paradigma zum Zeichnen von Graphen.

Klassendiagramme sind ein weitverbreitetes Hilfsmittel in der Modellierung und Analyse von objekt-orientierten Softwaresystemen. Ursprünglich in den sechziger Jahren entwickelt, erlebten objekt-orientierte Techniken in den neunziger Jahren eine Renaissance und sind heutzutage aus der modernen Softwareentwicklung nicht mehr wegzudenken. Zur Beschreibung von objekt-orientierten Softwaresystemen hat sich die Unified Modeling Language (UML), welche eine Vereinheitlichung verschiedener Modellierungssprachen darstellt, als lingua franca durchgesetzt. Die UML definert ein semantisches Modell eines objekt-orientierten Softwaresystems und eine Reihe von Diagrammarten, welche Teile eines semantischen Modells visualisieren. Klassendiagramme sind dabei die mit Abstand am meisten verwendete Diagrammart und beschreiben die statischen Beziehungen von Klassen und Objekten in dem Softwaresystem. Man unterscheidet dabei zwischen drei Arten von Beziehungen: Abhängigkeiten, Assoziationen und Vererbungen.

Im klassischen Anwendungsfall werden Klassendiagramme interaktiv von einem Benutzer mit Hilfe eines Werkzeugs erstellt, wobei die Anordnung der einzelnen Bestandteile des Diagramms vom Benutzer vorgegeben wird. Es gibt allerdings auch Anwendungsfälle, in denen zwar der Inhalt des Diagrams bekannt ist, aber nicht die Anordnung der einzelnen Elemente. Dies ist der Fall wenn ein Diagramm nicht von einem Benutzer erstellt wurde, sondern aus einer anderen Quelle stammt. Mögliche Quellen sind Programme zur automatischen Dokumentation oder Analyse-Werkzeuge zum Reverse-Engineering. Will ein menschlicher Benutzer aus diesen Diagrammen Erkenntnisse ziehen, ist es in diesen Fällen notwendig, eine übersichtliche Anordnung der Diagrammelemente zu bestimmen.

Um eine solche übersichliche Anordnung zu berechnen, entwickeln wir neue Verfahren, die auf dem „Topology-Shape-Metrics“ Paradigma zum Zeichnen von Graphen basieren. Bisher werden für praktische Anwendun-

gen des Graphenzeichnens hauptsächlich zwei andere Ansätze verwendet: der kräfte-basierte Ansatz für ungerichtete Graphen und der hierarchische Ansatz für gerichtete Graphen. Beide lassen sich in den Grundvarianten relativ einfach implementieren und sind sehr robust in Bezug auf Eingabedaten und Erweiterungen. Da die Vererbungsbeziehung eine gerichtete Substruktur in Klassendiagrammen bildet, basieren die meisten existierenden Verfahren zum automatischen Zeichnen von Klassendiagrammen sowohl in der wissenschaftlichen Literatur wie auch in der Implementierung von Software-Werkzeugen auf dem hierarchischen Ansatz.

Im Gegensatz dazu steht das Topology-Shape-Metrics Paradigma im Schatten dieser beiden dominierenden Ansätze. Es erfreut sich zwar in der Forschung großer Beliebtheit, konnte sich in praktischen Anwendungen aber bis jetzt noch nicht durchsetzen. Die Algorithmen, die auf diesem Paradigma basieren, gelten weder als leicht zu implementieren, noch als ausgesprochen robust oder erweiterungsfähig. Um diese Behauptungen zu widerlegen, wollen wir anhand einer Beispielanwendung zeigen, dass auch der Topology-Shape-Metrics Ansatz für praktische Probleme einsetzbar ist. Wir haben dazu das automatische Zeichnen von Klassendiagrammen ausgewählt, da trotz der großen praktische Bedeutung des Problems die meisten verfügbaren Implementierungen für automatisches Zeichnen von Klassendiagrammen nur sehr schlechte Ergebnisse liefern. Am Beispiel dieser Anwendung entwickeln wir den Topology-Shape-Metrics Ansatz zu einem praktisch nutzbaren Verfahren weiter, welches, wie unsere Experimente zeigen, für diese Anwendung den bisherigen Verfahren weit überlegen ist. Die Erweiterungen des Verfahrens sind dabei so allgemein gehalten, dass sie auch für andere Anwendungsgebiete relevant sind und somit die Anwendbarkeit des Topology-Shape-Metrics Ansatzes stark erweitern.

Die Komplexität des automatischen Zeichnens von Klassendiagrammen wird maßgebend von den Konventionen bestimmt, die für ihre Darstellung existieren und die zu berücksichtigen sind, damit ein durchschnittlicher Benutzer eine Zeichnung als übersichtlich erachtet. Die Vererbungsbeziehung nimmt aufgrund ihrer strukturbildenden Eigenschaft eine Sonderstellung in Klassendiagrammen ein. Um die von der Vererbungsbeziehung definierte Hierarchie zu verdeutlichen, sollen alle Kurven, die diese Beziehungen repräsentieren, monoton in eine Richtung gezeichnet werden, normalerweise im Diagramm von unten nach oben zeigend. Zusätzlich werden oft mehrere Kurven an einem Punkt zu einer Kurve zusammengefasst um die Übersichtlichkeit im Diagramm zu erhöhen. Diese Notation ist auch unter dem Namen „Hyperkanten-Notation“ bekannt. Weiterhin ist es üblich, Beziehungen als orthogonale Kurven darzustellen, d.h. als Linienzug, der alternierend aus horizontalen und vertikalen Segmenten besteht.

In Kapitel 2 werden diese Konventionen sowie weitere Ästhetikkriterien, welche die Lesbarkeit eines Diagramms beeinflussen, diskutiert. Dies ermöglicht es uns, das Problem des automatischen Zeichnens von UML Klassendiagrammen zu formalisieren und auf eine mathematische Grundlage zu stellen. Es werden die existierenden Verfahren für dieses Problem beleuchtet und ihre Stärken und Schwächen analysiert. Aufbauend auf diese Analyse werden die Grundzüge eines neuen Verfahrens für das automatische Zeichnen von Klassendiagrammen vorgestellt. Das Verfahren basiert auf dem Topology-Shape-Metrics Paradigma und besteht aus den drei Phasen *Planarisierung*, *Orthogonalisierung* und *Kompaktierung*. Die oben beschriebenen speziellen Anforderungen von Klassendiagrammen haben die Entwicklung neuartiger Algorithmen für alle drei Phasen notwendig gemacht, welche in den darauffolgenden Kapiteln behandelt werden.

Kapitel 3 beschreibt die Planarisierungsphase des Zeichnenalgorithmus. In der Planarisierungsphase wird für einen gegebenen Graphen eine planare Einbettung berechnet. Um Kantenrichtungen behandeln zu können, wurden die Konzepte Planarität und Aufwärtsplanarität zu dem neuen Konzept der *gemischten Aufwärtsplanarität* erweitert und ein neuer Planarisierungsalgorithmus entworfen.

In Kapitel 4 werden neue Orthogonalisierungsalgorithmen vorgestellt. Diese Algorithmen sind Erweiterungen des bekannten `Kandinsky` Algorithmus. Zuerst stellen wir eine Variante des `Kandinsky` Algorithmus vor, welche bestimmte Nebenbedingungen für Knicke und Winkel behandeln kann. Die speziellen Anforderungen von Klassendiagrammen, insbesondere das Richtungskriterium und die korrekte Behandlung von Hyperkanten, können mit Hilfe dieser Nebenbedingungen formuliert werden, was uns zu einem Orthogonalisierungsalgorithmus für Klassendiagramme führt. Zusätzlich analysieren wir die Komplexität verschiedener Knickminimierungsprobleme im `Kandinsky`-Modell und präsentieren neue heuristische Verfahren für diese Probleme.

In Kapitel 5 stellen wir einen neuen Kompaktierungsalgorithmus vor, der in der Lage ist, feste Knotengrößen zu respektieren. Dies ist eine der grundlegenden Anforderungen für das automatische Zeichnen von Klassendiagrammen. Der Algorithmus hat lineare Laufzeit, was eine drastische Verbesserung gegenüber existierenden Algorithmen für dieses Problem bedeutet.

Der in den bisherigen Kapiteln vorgestellte Algorithmus errechnet eine Zeichnung vollautomatisch, ohne Interaktion mit dem Benutzer. In Kapitel 6 wird ein Algorithmus vorgestellt, welcher interaktiv mit dem Benutzer eine Zeichnung erstellt. Der Algorithmus kann sowohl neue Elemente in ein Diagramm integrieren, ohne es allzu stark zu verändern, als auch auf Benutzerwünsche reagieren.

In Kapitel 7 wird die Implementierung der vorgestellten Algorithmen besprochen und ihre experimentelle Evaluierung vorgenommen. Wir vergleichen hierzu die Implementierung unseres Algorithmus mit *SugiBib*, der Implementierung eines Algorithmus zum automatischen Zeichnen von Klassendiagrammen, welche dem hierarchischen Paradigma folgt.

Wir beenden die Arbeit mit Kapitel 8, welches die Ergebnisse dieser

Arbeit zusammenfasst und einen Ausblick auf die weiteren möglichen Entwicklungen in diesem Feld gibt.

# Preface

Class diagrams are among the most popular visualizations for object oriented software systems and have a broad range of applications. There is a variety of tools available for creating and manipulating class diagrams, they all use the Unified Modeling Language as graphical notation. In many settings it is desirable that the placement of the diagram elements is determined automatically, especially when the diagrams are generated automatically. This is usually the case in reverse engineering. For this reason the automatic layout of class diagrams gained importance in the last years. However, current available tools perform the task of automatic layout only poorly, the results of the automatic layout algorithms of commercial or free case-tools are in the best case mediocre, but normally totally unacceptable. For this reason the automatic layout of UML diagrams was cited as one of the top challenges for automatic graph drawing at the last two graph drawing conferences, GD2001 in Vienna and GD2002 at Irvine.

We propose in this work a new algorithm for automatic layout of class diagram. The algorithm is an adaption of sophisticated graph drawing algorithms which have proven their effectiveness in many applications. The algorithm produces significantly better results and is more robust than state-of the art layout algorithms which are based on the hierarchical graph drawing paradigm.

The research would not have been possible without the support of the Deutsche Forschungsgemeinschaft (DFG), which contributed financial support by the grant Ka 812/8-1 and Ka 812/8-2. In the first place I want to thank Michael Kaufmann for initiating the research project and for all the support he gave to me, ranging from improving my personal fitness by cross-country skiing to giving me the possibility to attend various international workshops and conferences to meet other researchers. Especially the warm and productive atmosphere in his working group inspired me. Many other people contributed to this work, directly or indirectly, to whom I want to express my thanks. I want to thank Roland Wiese with whom I had endless discussions about the implementation and whom I want to thank that he gave me the opportunity to contribute to yFiles, which made the implementation of the presented algorithms possible. I want to thank Ulrik Brandes and Dorothea Wagner who initiated my work on sketch-driven graph-drawing and GraphML. Further I want to thank Holger Eichelberger for providing me with the implementation of SugiBib and the adaptions he made which made it possible to compare it to our algorithms. Many of the implementation work was done by Frank Eppinger and Martin Siebenhaller, without their help this work would not have been possible. Furthermore I want to thank Thomas Behr and Bernhard List for the work they have put in our testing environment, which I used for testing and evaluating the algorithms.

I want to thank the rest of the working group, namely Martin Schmollinger and Nina Lehmann for their supply of coffee and the nice hours drinking it, Marcus Schiesser for keeping the systems running, and Renate Hallmayer for proofreading this work. Finally I want to thank my parents for their support of my studies and my wife Claudia for her patience with me.

# Contents

**1 Introduction 1**

**2 Automatic Layout of Class Diagrams 7**

2.1 Preliminaries and Notion . . . . . . . . . . . . . . . . . . . . . 8
  2.1.1 Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.1.2 Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  2.1.3 Drawing of Graphs . . . . . . . . . . . . . . . . . . . . . 10
  2.1.4 Planarity . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 A Graph Based Model for Class Diagrams . . . . . . . . . . . 14
  2.2.1 Semantic Entities Mapping to a Vertex . . . . . . . . 16
  2.2.2 Semantic Entities Mapping to an Edge . . . . . . . . . 17
  2.2.3 Complex Symbols . . . . . . . . . . . . . . . . . . . . . 18
  2.2.4 Other Model Elements . . . . . . . . . . . . . . . . . . 20
2.3 The CLASS DIAGRAM LAYOUT Problem . . . . . . . . . . . 21
2.4 Applying the Topology-Shape-Metrics Approach to Class Diagrams . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.5 UML-`Kandinsky` . . . . . . . . . . . . . . . . . . . . . . . . 28
2.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
  2.6.1 Automatic Layout in UML-Tools . . . . . . . . . . . . 29
  2.6.2 The Seemann Algorithm and its Enhancements . . . . 30
  2.6.3 GoVisual . . . . . . . . . . . . . . . . . . . . . . . . . . 32

**3 Mixed Upward Planarization 35**

3.1 Mixed Upward Planarity . . . . . . . . . . . . . . . . . . . . . 36
3.2 Maximum Mixed Upward Planar Subgraph . . . . . . . . . . 38
  3.2.1 The Goldschmidt/Takvorian Planarization Algorithm 39
  3.2.2 The Algorithm for Mixed Graphs . . . . . . . . . . . . 41
3.3 Edge Insertion . . . . . . . . . . . . . . . . . . . . . . . . . . 44
  3.3.1 Insertion of Undirected Edges . . . . . . . . . . . . . . 44
  3.3.2 Insertion of Directed Edges . . . . . . . . . . . . . . . 46
3.4 Rerouting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.5 Complete Algorithm . . . . . . . . . . . . . . . . . . . . . . . 52

**4 Orthogonalization 55**
- 4.1 Tamassia's Algorithm . . . . . . . . . . . . . . . . . . . . . . . 56
- 4.2 Generalizations of Tamassia's Algorithm Using Reduction . . 61
- 4.3 `Kandinsky` . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
  - 4.3.1 The `Kandinsky` Model . . . . . . . . . . . . . . . . . . . 63
  - 4.3.2 The Network Flow Formulation . . . . . . . . . . . . . 67
- 4.4 Solving the `Kandinsky` Network Flow Problem . . . . . . . . 70
  - 4.4.1 Complexity of Solving Arc Partition Minimum Cost Flow Networks . . . . . . . . . . . . . . . . . . . . . . . 70
  - 4.4.2 The Negative-Cycle Approach to Solve the KANDINSKY BEND MINIMIZATION Problem . . . . . . . . . . 74
  - 4.4.3 A 2-Approximation Algorithm . . . . . . . . . . . . . . 78
  - 4.4.4 An Improved Heuristic . . . . . . . . . . . . . . . . . . 80
- 4.5 Constraints in `Kandinsky` . . . . . . . . . . . . . . . . . . . 81
- 4.6 Orthogonalization of UML Class Diagrams . . . . . . . . . . 89
  - 4.6.1 Mixed Upward Orthogonal Drawings . . . . . . . . . . 89
  - 4.6.2 Orthogonalization of the Upward Subgraph . . . . . . 90
  - 4.6.3 The Complete Algorithm . . . . . . . . . . . . . . . . . 94

**5 Compaction 97**
- 5.1 Previous Work . . . . . . . . . . . . . . . . . . . . . . . . . . 98
  - 5.1.1 Compaction of Orthogonal Shapes . . . . . . . . . . . 98
  - 5.1.2 Compaction in the `Kandinsky` Model . . . . . . . . . 99
  - 5.1.3 Label Placement . . . . . . . . . . . . . . . . . . . . . . 100
- 5.2 The Compaction Algorithm . . . . . . . . . . . . . . . . . . . 100
  - 5.2.1 Label Placement . . . . . . . . . . . . . . . . . . . . . . 101
  - 5.2.2 One-Dimensional Compaction . . . . . . . . . . . . . . 102
- 5.3 The Shape Graph Approach . . . . . . . . . . . . . . . . . . . 102
- 5.4 A Linear Time Compaction Algorithm . . . . . . . . . . . . . 106
  - 5.4.1 Compaction Shape . . . . . . . . . . . . . . . . . . . . 106
  - 5.4.2 Complete Shape Extensions of `Kandinsky` Shapes . . 109
  - 5.4.3 Computing the Length Complete Shape Extension . . 113
  - 5.4.4 Coordinate Assignment . . . . . . . . . . . . . . . . . 114
  - 5.4.5 Arbitrary Number of Edges at One Side . . . . . . . . 116

**6 Interactive Layout 119**
- 6.1 The Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 123
- 6.2 Interactive Planarization . . . . . . . . . . . . . . . . . . . . . 124
  - 6.2.1 The Straight Line Segment Intersection Problem . . . 125
  - 6.2.2 Valid Drawings . . . . . . . . . . . . . . . . . . . . . . . 126
  - 6.2.3 Determining the Valid Subgraph . . . . . . . . . . . . 127
- 6.3 Interactive Orthogonalization . . . . . . . . . . . . . . . . . . 128

**7 Experiments 131**
7.1 Implementation of UML-`Kandinsky` . . . . . . . . . . . . . . 131
7.2 Data and Experimental Setting . . . . . . . . . . . . . . . . . . 133
7.2.1 Class Diagrams . . . . . . . . . . . . . . . . . . . . . . . 133
7.2.2 Rome Graphs . . . . . . . . . . . . . . . . . . . . . . . . 133
7.2.3 Directed Rome Graphs . . . . . . . . . . . . . . . . . . . 135
7.2.4 Upward Planar Graphs . . . . . . . . . . . . . . . . . . . 135
7.2.5 Graphs With Limited Height . . . . . . . . . . . . . . . 135
7.3 Comparing UML-`Kandinsky` to SugiBib . . . . . . . . . . . . 135
7.3.1 Tests on Diagrams with up to 80 Classes . . . . . . . . 136
7.3.2 Tests on Diagrams with more than 80 Classes . . . . . 139
7.4 Planarization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
7.4.1 Class Diagrams . . . . . . . . . . . . . . . . . . . . . . . 140
7.4.2 Directed Graphs . . . . . . . . . . . . . . . . . . . . . . 140
7.5 Orthogonalization . . . . . . . . . . . . . . . . . . . . . . . . . 145
7.6 Compaction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
7.7 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147

**8 Conclusion 153**
8.1 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
8.2 Directions of Future Research . . . . . . . . . . . . . . . . . . . 155
8.2.1 Improvement of the Presented Algorithms . . . . . . . 156
8.2.2 Advanced Modeling Constructs . . . . . . . . . . . . . 156
8.2.3 New Application Domains . . . . . . . . . . . . . . . . . 158

# Chapter 1

# Introduction

In the last two decades *graph drawing* emerged as a new field in the area of information visualization. The central question this field tries to answer is: Given a set of entities and a set of binary relationships between these entities (this is usually referred to as a *graph*), find a "good" visualization of this data. What a good visualization is depends of course on the semantics of the data as well as on the beholder, but there are some principles which proved to work for a large set of different types of data almost independently of the spectator. The most common type of visualization found for graphs is the node-link model, in which the entities are represented as shapes and the relationships as paths between the shapes.

Two approaches for drawing graphs in this model have been extraordinarily successful in practice: the force directed approach and the hierarchical approach. The reason for their success is that they provide fairly good results while being robust with respect to the input, extensible with respect to special constraints in the resulting drawing and are nevertheless not too hard to implement.

Another approach, called the topology-shape-metrics approach, has been studied extensively in the research community and although it seems to be very promising it has never attracted the same attention in practice as the force-directed and the hierarchical approach. While it shares the first property of the other two approaches, providing fairly good results, it has the reputation to lack the robustness and the extensibility of the other two approaches while being hard to implement.

In this work we show that the topology-shape-metrics approach has this reputation wrongly by applying it to a complex real-world scenario: the automatic layout of UML class diagrams. UML class diagrams arise in object-oriented software engineering and because of their complexity are perfectly suited as a benchmark application for graph drawing algorithms. This is emphasized by the fact that the automatic layout of UML class diagrams was cited as one of the top challenges for automatic graph drawing at the last two graph drawing conferences, GD2001 in Vienna and GD2002 at Irvine.

Class diagrams are used to describe the static structure of an object-oriented system. According to the authors of the UML class diagrams are the most common diagram found in modeling object-oriented systems [12]. Object-oriented techniques are ubiquitous in software engineering today. Although introduced in the late 1960s object-oriented techniques became increasingly important in the 1990s and are today one of the most important tools in software engineering. Our presentation of class diagrams and object-oriented methods is based on [11].

The fundamental concept shared by all object-oriented techniques is the *object model*. The main parts of the object model are *objects* and *classes*. An object can be defined informally as a tangible entity that exhibits some well-defined behavior. Booch gives the following definition for an object [11]:

> An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms *instance* and *object* are interchangeable.

Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction. Booch gives the following definition for a class [11]:

> A class is a set of objects that share a common behavior and a common structure.

A single object is simply an instance of a class.

A class diagram depicts in a visual language a set of classes of an object-oriented system and the relationships between them. We distinguish between mainly three different types of relationships between classes: *generalizations*, *associations* and *dependencies*:

A generalization is a relationship between a general thing (called the *superclass*) and a more specific kind of that thing (called the *subclass*). Generalization is sometimes called an "is-a-kind-of" relationship: one thing is-a-kind-of another thing.

An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa.

A *dependency* is a relationship that states that a change in specification of one thing (called the *supplier*) may affect another thing that uses it (called the *client*).

If we assume that all relationships are binary in a class diagram it can be seen mathematically as a drawing of a graph. Figure 1.1 shows an example for a class diagram.

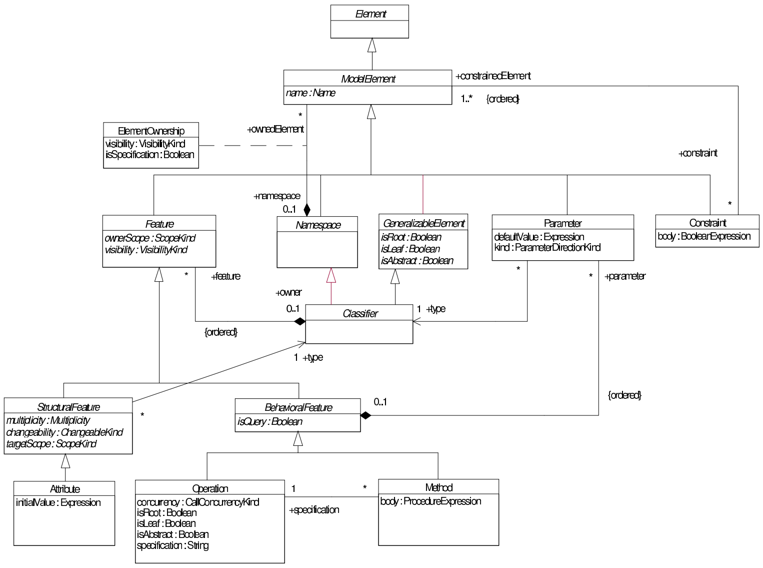


Figure 1.1: An example for an UML class diagram from [95].

If we have a closer look at the definition of class diagrams we discover that there is no one-to-one correspondence between the graph elements and the semantic entities of the diagram. There is a more complex mapping of semantic entities to graph elements: some semantic entities map not to single graph elements, they map to a collection of graph elements in the diagram. For example a class diagram may contain n-ary relationships and not only binary relationships. However binary relationships have no correspondence in graphs since a graph models only binary relationships. In section 2.2 we will discuss the graphical notation of the semantic entities and their mapping to graph elements in detail.

Object-orientation may play a role in each stage of a software engineering process, from analysis to implementation, and class diagrams may be used in each stage of this process, too. Prior to an implementation usually a *design* of the system is performed which provides the proper and effective structuring of the system and serves as a blueprint for the implementation. *Object-oriented design (OOD)* is a design method based in the object model. Booch gives the following definition for object-oriented design [11]:

> Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

The main philosophy of OOD is that every complex system is best approached through a small set of nearly independent views of a model. The model itself is an object-oriented decomposition of the system. It is believed that no single view is sufficient. Different views visualize different abstractions of the model. Therefore in OOD a model is defined by a set of consistent views, each focussing on another aspect of the system.

Each view should be described by a formal language to avoid ambiguities. The Unified Modeling Language is such a language. In our description of the UML we follow the UML specification [95]. For an introduction to the UML we point to [1, 12, 58]. According to the authors [95] the Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components. The UML is not intended to be a visual programming language, in the sense of having all the necessary visual and semantic support to replace programming languages. Furthermore it is process independent, i.e., it is not defining a standard process for software engineering.

Prior to the definition of the UML there were various different modeling languages, among them Booch, OMT, and OOSE. Each of them had a different focus and a different notation. UML was designed to unify these languages and to overcome weaknesses in the definition of them. The UML should be rather seen as a natural evolutionary step of Booch, OMT, and OOSE, than a radical departure of them. The UML 1.1 was adopted in November 1997 as an OMG standard and has become a defacto standard in software industry since then.

The UML follows the OOD paradigm that every complex system is best approached through a small set of nearly independent views of a semantic model. Each view visualizes a different abstraction of the model. UML supports the following views of a model:

- structural view (class diagram, object diagram)
- user view (use case diagram)
- implementation view (component diagram)
- behavioral view (sequence diagram, collaboration diagram, statechart diagram, activity diagram)
- environment diagrams (deployment diagram)

Each view defines a set of *diagrams*, which provide multiple perspectives of the system under analysis or development. The underlying semantic model integrates these perspectives so that a self-consistent system can be analyzed and built. To support this concept the UML specification is divided into UML Semantics and UML Notation Guide. The UML Semantics part contains the semantic definition of an UML Model. It defines the building blocks to define a model of a system. The UML Notation Guide defines the visualization of these building blocks.

But what are the benefits of an automatic layout algorithm for class diagrams, why is automatic visualization important ? In a first attempt one might say that automatic layout of class diagrams is not needed, since class diagrams are crafted by hand in an object-oriented design step. Since these diagrams stem directly from the designer who has already a rough sketch of it in his mind an automatic layout method is not needed. Automatic layout seems not appropriate since the algorithm can not guess this sketch.

However, we often have a different setting. Assume that a diagram is generated by a reverse engineering tool. In this case the diagram elements have to be placed by some algorithm since they are created automatically. Of course alternatively this can be done by hand by the user, but obviously this solution scales not very well with growing size of the reverse engineered project. Especially if one wants just to have a quick look at the system this does not seem to be the method of choice. Another example is the automatic documentation of software. In this scenario diagrams should be generated automatically and added to the documentation of a software project. Automation is desired since this ensures that the documentation is synchronized with the current state of the software.

Even when the diagram had been created by hand, automatic layout may play an invaluable role. Applying an automatic layout algorithm to a given diagram yields a new view of the diagram and may reveal properties of the model which had not been exhibited well in the original diagram. In the human creation process of diagrams it is also often observed that the user starts with a small diagram which grows over time. At a certain point the diagram becomes more and more unreadable, because the new elements do not fit well in the existing diagram. In this setting interactive layout can help to improve the drawing.

As the title of the work suggests we concentrate on the main diagram type the UML provides for structural views of a model, *class diagrams*. An algorithm for the automatic layout of class diagrams takes a class diagram as input and calculates a geometric representation for the elements of the diagram. It has to consider some constraints for the geometric representations, for example that classes are represented by rectangles of prescribed size or that some relationships must be drawn as a sequence of vertical and horizontal segments. Each graph drawing algorithm which can handle these constraints is therefore a potential layout algorithm for class diagrams. But not all graph drawing algorithms are equally suited for automatic layout of class diagrams since they may not consider the special conventions used in drawing class diagrams.

In Chapter 2 we will present a sound mathematical model for class diagrams and discuss the constraints, aesthetic criteria and conventions which apply for class diagrams. We will review the topology-shape-metrics approach, which consists of the three phases planarization, orthogonalization and compaction and discuss how the requirements of the layout of class diagrams affect the single phases. Based on these observations we give an outline of our new automatic layout algorithm UML-`Kandinsky` . We compare our approach to existing algorithms, most of them are based on the hierarchical approach. To meet the special requirements of class diagrams we have developed new algorithms for all three phases of the topology-shape-metrics approach. These new algorithms are discussed in the subsequent chapters.

In Chapter 3 we present a new planarization algorithm, which creates mixed upward planarizations of mixed graphs. To best of our knowledge this is the first time that planarization of directed and mixed graphs is discussed.

In Chapter 4 we present a new orthogonalization algorithm which can consider the special requirements of class diagrams, notably upward directed edges and hyperedges. We modified and adapted the very successful `Kandinsky` algorithm for this purpose.

In Chapter 5 we present a new compaction algorithm for orthogonal drawings with vertices of prescribed size, a constraint essential in class diagrams. Our algorithm is the first linear time algorithm for this problem and improves the running time for this problem drastically.

In Chapter 6 we discuss interactive layout of class diagrams. All algorithms covered so far do only allow very limited user interaction. We will present a new paradigm for interactive layout, the *sketch-driven* approach, and show how it can be applied to class diagrams.

In Chapter 7 we provide experimental results proving the efficiency of our new automatic layout algorithm UML-`Kandinsky` . We give a short description of the implementation of UML-`Kandinsky` and compare it to SugiBib, an automatic layout algorithm for class diagrams based on the hierarchical approach. Further we evaluate the algorithms for the three phases of UML-`Kandinsky` in detail.

We finish with Chapter 8, which contains a discussion of the presented work and an outlook to future directions of research related to the topics of this work.

All algorithms discussed in this work are implemented and example outputs in form of drawings are provided in many places of the work. Results of this work have already been published in [16, 47, 49, 50, 51, 118, 119].

# Chapter 2

# Automatic Layout of Class Diagrams

In this chapter we define the automatic layout problem for class diagrams and propose an algorithm based on the topology-shape-metrics approach to solve it. We will show why this approach is promising and discuss the challenges which arise when we want to apply the topology-shape-metrics approach to the automatic layout of class diagrams. This chapter follows partly [51] and is organized as follows:

We first review in Section 2.1 the basic mathematical concepts which we will use in this work before we propose in Section 2.2 a graph-based model for class diagrams.

There are several factors which influence the readability of class diagrams. In Section 2.3 we will review how the readability of class diagrams can be measured using *aesthetic criteria*. We will identify the relevant aesthetic criteria for class diagrams and define the CLASS DIAGRAM LAYOUT problem.

In Section 2.4 we review the topology-shape-metrics approach and discuss how this paradigm can be used for the visualization of class diagrams. We will especially examine how the different phases of the approach are affected by the special requirements for the layout of class diagrams.

In Section 2.5 we present an outline of UML-Kandinsky, our new algorithm for the automatic layout of class diagrams.

In Section 2.6 we review existing algorithms for the automatic layout of class diagrams. Most current automatic layout algorithms for class diagrams are based on the hierarchical approach for drawing graphs. Applying the hierarchical approach to the automatic layout of class diagrams has several drawbacks notably in the absence of generalizations in the diagram. We will see that our approach overcomes this and other drawbacks of the hierarchical approach and works well for all types of diagrams.

## 2.1 Preliminaries and Notion

In this section we will review the main mathematical concepts that we will use in the remainder of this work: strings, graphs, drawings of graphs, and planarity.

### 2.1.1 Strings

Let $\Sigma$ be a finite set. We will refer to $\Sigma$ as *alphabet* and to the elements of $\Sigma$ as *symbol.* A *string* over an alphabet $\Sigma$ is a sequence of symbols of $\Sigma$. The empty sequence is denoted by $\epsilon$. The length of a string $s$ is denoted by $|s|$. For a string $s$ and $a \in \Sigma$ we denote with $\#_a s$ the number of $a$ in $s$.

The result of appending a string $s_2$ to a string $s_1$ is denoted by $s_1 + s_2$. If the resulting string should carry the name $s_1$ after the concatenation, we denote it with $s_1 + = s_2$.

A *bit-string* is a string over the alphabet $\{0, 1\}$. The bit-wise complement of a bit-string $s$ is denoted by $\overline{s}$. The reverse of $s$ is denoted by $\overleftarrow{s}$.

The edit distance between two strings is defined as the smallest number of simple edit operations (insert, delete, and substitute) required to change one string into another.

We denote with $\mathrm{edit}_{(c_i,c_d)}(s_1, s_2)$ the version of the edit distance, which allows only the operations insert and delete, where $c_i$ denotes the cost for an insert operation and $c_d$ the cost for a delete operation. Note that the costs must be non-negative. The following properties of this type of edit distance will be useful in the remainder of this work:

**Lemma 2.1** *For two strings $s_1$ and $s_2$ and $a, b > 0$ holds:*

1. $c \cdot \mathit{edit}_{(a,b)}(s_1, s_2) = \mathit{edit}_{(ca,cb)}(s_1, s_2)$
2. $\mathit{edit}_{(a,b)}(s_1, s_2) = \mathit{edit}_{(a,0)}(s_1, s_2) + \mathit{edit}_{(0,b)}(s_1, s_2)$

**Proof:** Each sequence $o$ of operations which leads to an edit distance of $\mathrm{edit}_{(a,b)}(s_1, s_2)$ has the same number of insertions $i(o)$ and deletions $d(o)$. Assume that there are two sequences of operations $o_1$ and $o_2$ leading to the same edit distance with different number of operations. Since $|s_2| = |s_1| + i(o_1) - d(o_1)$ and $|s_2| = |s_1| + i(o_2) - d(o_2)$ it follows $i(o_1) - d(o_1) = i(o_2) - d(o_2)$. If $i(o_1) > i(o_2)$, then also $d(o_1) > d(o_2)$ and therefore $a \cdot i(o_1) + b \cdot d(o_1) > a \cdot i(o_2) + b \cdot d(o_2)$ which is a contradiction to the assumption that they have the same edit distance.

Let $o$ be a sequence of operations which leads to edit distance $\mathrm{edit}_{(a,b)}(s_1, s_2)$. Then the first claim follows from $c \cdot (i(o) + d(o)) = c \cdot i(o) + c \cdot d(o)$ and the second claim from $a \cdot i(o) + b \cdot d(o) = (a \cdot i(o) + 0 \cdot d(o)) + (0 \cdot i(o) + b \cdot d(o))$.
$\square$

**Lemma 2.2** *Let* $s_1$ *and* $s_2$ *be two strings, then*

$$a \cdot \mathit{edit}(s_1, s_2) + b \cdot (|s_1| - |s_2|) = \mathit{edit}_{(a+b,a-b)}(s_1, s_2) \ .$$

**Proof:** With $|s_1| - |s_2| = \text{edit}_{(1,0)} - \text{edit}_{(0,1)}$

$$\begin{aligned} & a \cdot \text{edit}(s_1, s_2) + b \cdot (|s_1| - |s_2|) = \\ & \text{edit}_{(a,a)}(s_1, s_2) - b(\text{edit}_{(1,0)}(s_1, s_2) - \text{edit}_{(0,1)}(s_1, s_2)) = \\ & \text{edit}_{(a+b,a-b)}(s_1, s_2) \end{aligned}$$

□

### 2.1.2 Graphs

In this section we introduce basic concepts from graph theory. For a comprehensive overview of graph theory we refer to [38].

A *graph* $G$ is denoted by a pair $(V, E)$, where $V$ is the set of vertices and $E \subseteq V \times V$ is the set of edges. A *graph element* is either a vertex or an edge. Sometimes we use the notion $V(G)$ for the vertices, resp. $E(G)$ for the edges, of a graph $G = (V, E)$. We denote with $\mathit{adj}(v)$ the set of edges adjacent a vertex $v \in V$. The degree $\delta_G(v)$ of a vertex $v \in V$ is the number of edges in $E$ adjacent to $v$. A graph is called *4-graph* if each vertex has degree smaller or equal 4. We say that $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. In this case we write $G' \subseteq G$. A graph isomorphism $f : V(G) \rightarrow V(H)$ is a bijection between the vertices of two graphs $G$ and $H$ with the property that any two vertices $u$ and $v$ from $G$ are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in $H$. If an isomorphism can be constructed between two graphs, then we say those graphs are isomorphic.

We call a graph *directed* if all pairs in $E$ are ordered and *undirected* if all pairs in $E$ are unordered. We call the first entry in a directed edge the *source* and the second *target*. Ignoring for every directed edge the order of its vertices, we get an undirected graph, which is called the *underlying graph*. For a vertex $v \in V$, we denote with $\mathit{in}(v)$ the set of edges in $E$ which have target $v$, and with $\mathit{out}(v)$ the set of edges with source $v$. The in-degree $\delta_G^-(v)$ denotes the number of edges in $\mathit{in}(v)$, and the out-degree $\delta_G^+(v)$ the number of edges in $\mathit{out}(v)$. We call a vertex with in-degree 0 a *source*, and a vertex with out-degree 0 a *sink*. A directed acyclic graph is called an *st-graph* if it has exactly one sink and one source.

If a graph contains both, directed and undirected edges, we call it a *mixed graph*. In this case we denote the set of directed edges with $E_d(G)$ and the set of undirected edges with $E_u(G)$. Often the shorter terms *digraph*, resp. *migraph*, are employed instead of the terms directed graph, resp. mixed graph.

An ordering $\pi : V \longrightarrow \mathbb{N}$ of a directed graph $G = (V, E)$ is called a *topological ordering* if for every edge $e = (v, w) \in E$ holds $\pi(v) < \pi(w)$.

A (simple) path $P = (v_0, e_1, v_1, \ldots, e_k, v_k)$ in a directed graph $G = (V, E)$ is an alternating sequence of vertices $V(P) = \{0, \ldots, v_k\} \subseteq V$ and edges $E(P) = \{e_1, \ldots, e_k\} \subseteq E$ such that $\{v_{i-1}, v_i\} = e_i$, $1 \leq i \leq k$, and $v_i \neq v_j$ for $0 \leq i \neq j \leq k$. The *length* of $P$ is $k$.

We denote with $v \overset{*}{\longrightarrow}_G w$ if there is a path between two vertices $v$ and $w$ in a graph $G$.

### 2.1.3 Drawing of Graphs

As already noted class diagrams can be modeled as drawings of graphs. In this section we will cover some basic definitions for drawings of graphs. For an introduction to graph drawing we point to [33, 81].

A *point drawing* $\Gamma$ of a graph $G = (V, E)$ maps each vertex $v \in V$ to a point $p(v)$ in the plane and each edge $e = (v, w) \in E$ to an open Jordan curve $c(e)$ such that $c(e)$ connects $p(v)$ with $p(w)$. A *rectangle drawing* $\Gamma$ of a graph $G = (V, E)$ maps each vertex $v \in V$ to a rectangle $r(v)$ in the plane and each edge $e = (v, w) \in E$ to an open Jordan curve $c(e)$ such that $c(e)$ connects $r(v)$ with $r(w)$. An *orthogonal drawing* of a graph is a point drawing in which the curve for each edge is a sequence of horizontal and vertical segments. In an *orthogonal grid drawing* the vertices and the bends along the edges have integer coordinates. Note that a graph admits an orthogonal grid drawing if and only if it is a 4-graph. An *orthogonal rectangle drawing* of a graph is a rectangle drawing in which the curve for each edge is a sequence of horizontal and vertical segments. In the corresponding grid drawing the boundaries of the rectangles and the bends along the edges have integer coordinates. For an illustration see Figure 2.1.

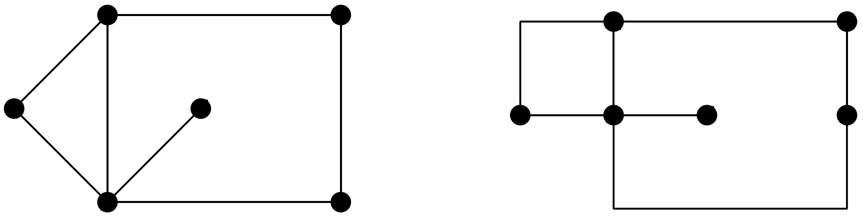
### 2.1.4 Planarity

We call a point drawing $\Gamma$ of a graph $G$ *planar* if no two edges in the drawing intersect except at common endpoints. A graph is *planar* if it has a planar point drawing.

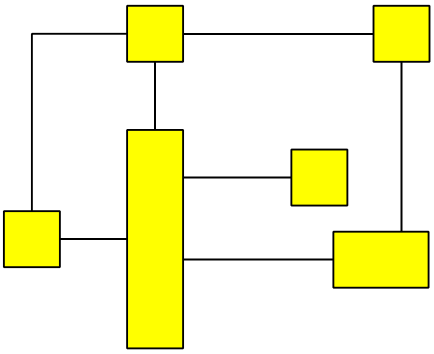
The above definition of planarity is based on geometric properties of the graph. Kuratowski found a pure combinatorial description of the class of planar graphs. To understand this description we need the concept of *subdivision*.

**Definition 2.1** *A* subdivision *of an edge $e = (v, w)$ in a graph $G = (V, E)$ can be obtained by adding a vertex $u$ to $V$, removing the edge $e$ from $E$ and adding the two edges $e_1 = (v, u)$, $e_2 = (u, w)$ to $E$. A graph $G' = (V', E')$ is called a subdivision of a graph $G = (V, E)$ if $G'$ can be obtained by a series of subdivisions of edges of $E$.*

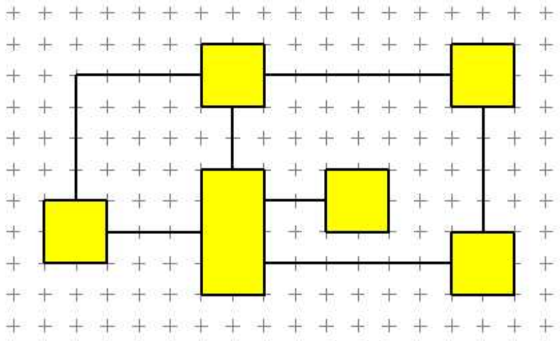
Thus, a subdivision of a graph is another graph where some edges of the original graph have been replaced by paths. For planar graphs the following theorem holds:



(a) Point drawing (b) Orthogonal point drawing



(c) Orthogonal rectangle drawing

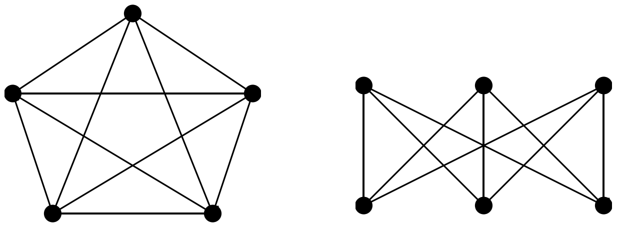


(d) Orthogonal grid rectangle drawing

Figure 2.1: Examples for different types of drawings.

**Theorem 2.1** *([87]) A graph $G$ is planar if and only if it contains no subgraph that is isomorphic to or is a subdivision of $K_5$ (the complete graph with 5 vertices) or $K_{3,3}$ (the complete bipartite graph with 3 vertices in each set).*

Whether a graph is planar or not can be tested in linear time [13, 73].



(a) (b)

Figure 2.2: The two minimal non-planar graphs $K_5$ (a) and $K_{3,3}$ (b).

If $\Gamma$ is a planar drawing, the set $\mathbb{R}^2 \setminus \Gamma$ is open and its regions are called the *faces* of $\Gamma$. Since $\Gamma$ is bounded, exactly one of the faces is unbounded. This face is called the *outer face* of $\Gamma$. The boundary of each face is a cycle in the graph.

A convenient encoding of a planar drawing is a *planar representation.* A planar representation $F$ of a planar graph $G = (V, E)$ defines for each edge $(v, w) \in E$, which might be directed or undirected, two *darts* $e = (v, w)$ and $e^R = (w, v)$. We say that $e^R$ is the reverse of $e$ and vice versa. We denote with $\bar{e}$ the reverse of a dart $e$. We denote with $\bar{E}$ the set of darts defined by $E$. The planar representation $F$ contains one cyclic list for each face, which contains the darts encountered by walking in clockwise ordering around the face. The first face in $F$ is by convention the outer face and is denoted by $f_{out}$. When we use the term face in the remainder of this work, we refer to the list of darts describing the face. For a dart $e$ we denote with $\mathit{face}(e)$ the face which contains $e$.

An *embedding* $\mathcal{E}$ of a graph is defined as the counter-clockwise cyclic ordering $\mathcal{E}(v)$ of the adjacent edges of each vertex $v$ of the graph. Each edge $e = (v, w) \in E$ appears twice in $\mathcal{E}$, namely as $(v, w)$ in $\mathcal{E}(v)$ and as $(w, v)$ in $\mathcal{E}(w)$. An embedding is *planar* if there is a planar drawing of the graph which preserves this ordering.

It is easy to obtain the planar representation from an embedding and vice versa. Given an embedding $\mathcal{E}$ we denote the planar representation induced by $\mathcal{E}$ with $F_{\mathcal{E}}$, and given a planar representation $F$ we denote the embedding induced by $F$ with $\mathcal{E}_F$. A graph with a given planar representation $F$ is called a *plane graph* and is denoted with $G = (V, E, F)$. We will omit the index $F$ in $\mathcal{E}_F$ and write just $\mathcal{E}$ if it is clear to which planar representation we refer.

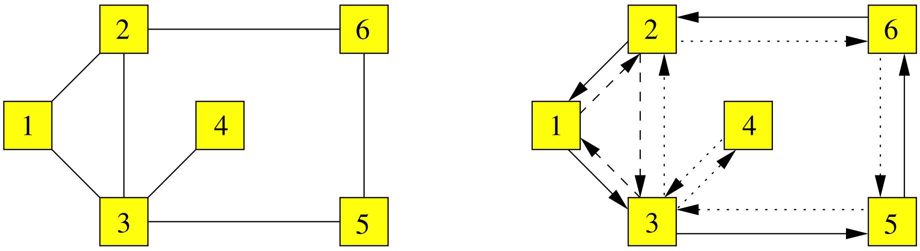


Figure 2.3: Example for a planar embedding.

Figure 2.3 illustrates the definition of planar representation on an example. The plane graph is defined by the planar representation $F = (f_0, f_1, f_2)$ where

$$\begin{aligned} f_0 &= \{(1,3),(3,5),(5,6),(6,2),(2,1)\}, \\ f_1 &= \{(1,2),(2,3),(3,1)\}, \\ f_2 &= \{(5,3),(3,4),(4,3),(3,2),(2,6),(6,5)\}\ . \end{aligned}$$

In Figure 2.3 (b) the face $f_0$ is denoted by the solid darts, the face $f_1$ by the dashed darts and the face $f_2$ by the pointed darts. The corresponding embedding $\mathcal{E}$ is:

$$\begin{aligned} \mathcal{E}(1) &= \{(1,2),(1,3)\}, \\ \mathcal{E}(2) &= \{(2,1),(2,3),(2,6)\}, \\ \mathcal{E}(3) &= \{(3,2),(3,1),(3,4)\}, \\ \mathcal{E}(4) &= \{(4,3)\}, \\ \mathcal{E}(5) &= \{(5,6),(5,3)\}, \\ \mathcal{E}(6) &= \{(6,2),(6,5)\}\ . \end{aligned}$$

The following famous theorem for planar graphs was first discovered by Euler around 1750:

**Theorem 2.2** *Let $G$ be a connected planar graph with $n$ vertices, $m$ edges, and $l$ faces. Then*

$$n - m + l = 2.$$

A direct consequence of this theorem is that any planar graph has at most a linear number of edges, moreover:

**Corollary 2.1** *A planar graph with $n \geq 3$ vertices has at most $3n-6$ edges.*

Every graph can be made planar by replacing edge crossings by dummy vertices. The planar graph obtained by this replacements is called *planarization*, Figure 2.4 shows a planarization of $K_5$.

**Definition 2.2** *Given a graph $G = (V, E)$, the graph $G' = (V \cup C, E', F)$ is a planarization of $G$ with crossing number $|C|$ if and only if*

- *$G'$ is a plane graph,*
- *$\delta_{G'}(v) = 4$ for all $v \in C$, and*
- *There is a mapping $\hat{p}$ from the edges in $E$ to paths in $G'$ with:*
  - *for each $e = (v, w) \in E$, $\hat{p}(e)$ is a path from $v$ to $w$ with $V(\hat{p}(e)) \setminus \{v, w\} \subseteq C$,*
  - *for each edge $e' \in E'$ there is an edge $e \in E$ with $e' \in \hat{p}(e)$, and*
  - *the edges of two paths are pairwise disjoint: $E(\hat{p}(e_1)) \cap E(\hat{p}(e_2)) = \emptyset$ for $e_1 \neq e_2 \in E$.*

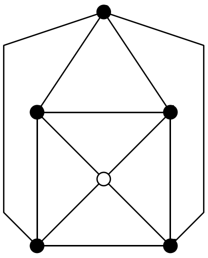


Figure 2.4: Planarization of $K_5$. The white vertex represents a crossing.

## 2.2 A Graph Based Model for Class Diagrams

In Chapter 1 we gave an informal introduction to class diagrams. In this section we will discuss class diagrams in greater detail and present a graph-based model for them.

In the UML specification [95] we find the following definition for a diagram:

> Most UML diagrams and some complex symbols are graphs containing nodes connected by paths. The information is mostly in the topology, not in the size or placement of the symbols (there are some exceptions, such as a sequence diagram with a metric time axis). There are three kinds of visual relationships that are important:
>
> 1. connection (usually of lines to 2-d shapes),
> 2. containment (of symbols by 2-d shapes with boundaries), and

> 3. visual attachment (one symbol being near another one on a diagram).
>
> These visual relationships map into connections of nodes in a graph, the parsed form of the notation. UML notation is intended to be drawn on 2-dimensional surfaces.

The UML Model itself does not distinguish the three types of visual relationships, it defines only the semantics of the relationships. The visual notation of the semantic entities of the UML Model is defined in the UML Notation Guide. It defines the mapping of the relationships in the model to one of the above visual relationships.

The diagram model, our data model for class diagrams, uses a mixed graph to model the first kind of visual relationship: connection. The graph is mixed since some relationships in the diagram are symmetric, for example undirected associations, while others are directed, for example dependencies.

We model the visual attachment relationship by *labels*. A label is in our setting a two-dimensional shape which refers to a graph element. In a diagram it should be clear from the placement of the label to which graph element it refers. We assume in the remainder that labels have rectangular shape.

To denote the semantic differences between the different kinds of semantic entities in a diagram we add type functions, which return for each element in the graph model the corresponding type in the UML model.

Furthermore the diagram model contains a size function which defines the size of the vertices and labels in the diagram. The size of these elements in the diagram normally depends on their content, for example for a class vertex the size depends on the attributes and operations of the class. However, we cannot derive the size of these elements directly from their content because the size in the diagram depends on the font type and font size, border width, and other parameters which are not defined on the UML model level. Therefore we have to add this information explicitly to the diagram model.

We call our diagram model *class diagram graph*, which is defined as follows:

**Definition 2.3** *A* class diagram graph *is defined as*

- *a mixed graph* $G = (V, E)$*,*
- *a set of labels* $L$*,*
- *a mapping* $refer : L \to E$*,*
- *a mapping* $vtype : V \to \{\texttt{class}, \texttt{interface}, \texttt{package}, \texttt{note}, \texttt{diamond}, \texttt{dummy}\}$*,*

- *a mapping etype* : $E \rightarrow \{\texttt{dependency}, \texttt{generalization}, \texttt{association}, \texttt{connector}\}$,
- *a mapping ltype* : $L \rightarrow \{\texttt{multiplicity}, \texttt{role}, \texttt{name}, \texttt{stereotype} \ \texttt{ordered}, \texttt{constraint}\}$,
- *a mapping size* : $V \cup L \rightarrow I\!N^2$.

A drawing of the class diagram graph defines a drawing for the class diagram. The *class diagram layout* is defined as follows:

**Definition 2.4** *A* class diagram layout *of a class diagram graph* $\mathcal{C}$ *is defined as a mapping* $\Gamma(\mathcal{C})$ *of the vertices and labels to rectangles of size as defined by the mapping size and the edges to open jordan curves.*

In the following we will discuss the visual notation of class diagrams and define the mapping of the diagram elements to graph elements in the class diagram graph. We can classify the semantic entities into three classes: those which map to a single vertex, those which map to a single edge with labels, and those which map to more complex sub-structures of the class diagram graph. We will discuss these classes in detail now.

### 2.2.1 Semantic Entities Mapping to a Vertex

The diagram elements class, interface, object, and package correspond directly to a vertex in the graph. The type of the vertex is the type of the diagram element.

#### Class and Interface

As already mentioned classes are the most important building block of any object-oriented system. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

Classes are represented by rectangles consisting of multiple compartments. Each compartment contains different features of the class, for example operations, attributes and the name in the first compartment.

An *interface* is a specifier for the externally-visible operations of a class without specification of internal structure.

Interfaces are represented as classes with the only difference that the word `interface` is printed above the name.

#### Package

A *package* is a grouping of model elements. Packages themselves may be nested within other packages.

Packages are represented as rectangles with the name attached in a small compartment above the left upper corner.

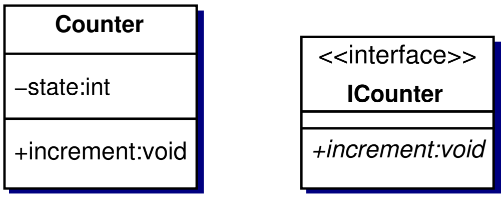


Figure 2.5: Example for a class and an interface in UML notation.

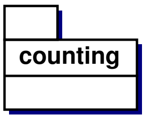


Figure 2.6: Example for a package in UML notation.

### 2.2.2 Semantic Entities Mapping to an Edge

The diagram elements dependency, generalization, binary link and binary association correspond directly to an edge in the graph. The type of the edge is the type of the diagram element. Each of the above diagram elements may specify a stereotype. Stereotypes map to a label of type `stereotype`.

#### Dependency

Dependencies are rendered as dashed lines with an arrow pointing to the client.

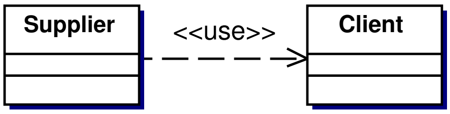


Figure 2.7: Example for a dependency with stereotype `use` in UML notation.

#### Generalization

Generalizations between classes are rendered as solid lines having as arrow head an empty triangle pointing to the superclass. A special type of generalization is the implementation of an interface by a class. Implementation relationships are rendered as dashed lines.

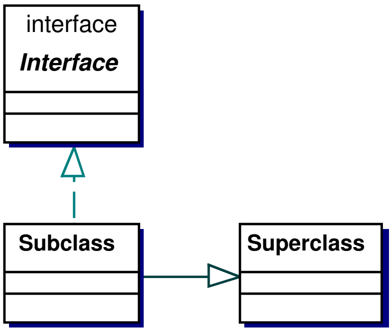


Figure 2.8: Two examples for generalization in UML notation.

**Association**

An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa. An *aggregation* is a special kind of association which models a whole/part relationship. Aggregation is sometimes called a ”has-a” relationship: meaning that an object of the whole has objects of the part.

Associations may have a name attached. This name maps to a label of type `name`. At each side of an association or link there may be a role specified. This role maps to a label of type `role`. Additionally associations may have a multiplicity assigned and have an ordered flag. The multiplicity maps to a label of type `multiplicity`, the ordered flag maps to a label of type `ordered`. Associations are rendered as solid lines. They may have arrowheads to indicate the navigability from one object to the other object. Aggregations have an empty diamond as adornment at the class representing the ”whole” in the whole/part relationship. Associations may be directed (like aggregations) or undirected.

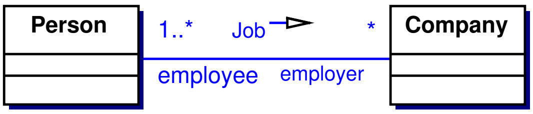


Figure 2.9: Example for association in UML notation.

### 2.2.3 Complex Symbols

The semantic entities association class, n-ary association, and note cannot be mapped directly to one vertex or edge.

### N-ary Association

An n-ary association is a relationship between more than two semantic entities. This is visualized by a diamond connecting to all these semantic entities. The diamond is modeled as a vertex of type `diamond` in the graph, the connections as edges of type `connector`.

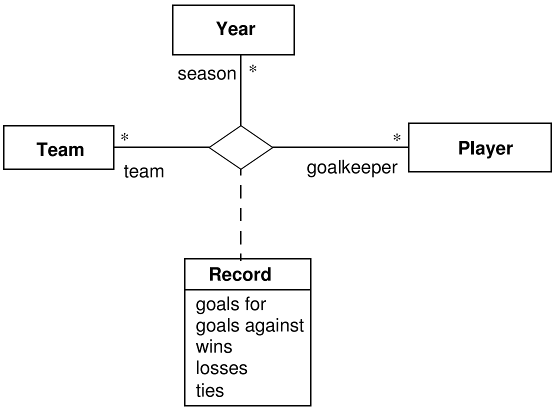


Figure 2.10: Example for an n-ary association in UML notation.

### Association Class

An association class is an association that also has class properties (or a class that has association properties). The notation is that of an association and a class connected by a dashed line. If the association is binary the representation of the association is a path of length greater than two and the class vertex is connected to a dummy vertex of this path with an edge of type `connector`. If the association is an n-ary association the class is connected to the diamond vertex of the representation of the n-ary association with an edge of type `connector`.

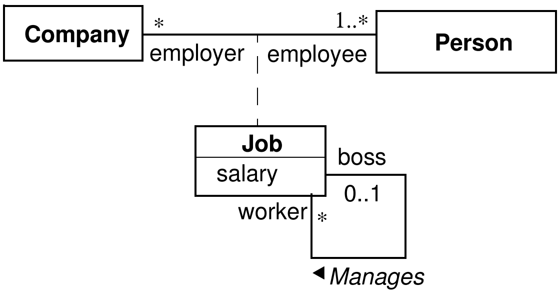


Figure 2.11: Example for an association class in UML notation.

### Note

A note is represented by a rectangle with a "bent corner" in the upper right corner. It contains arbitrary text. A note may be associated with each type of semantic entity by *note links.* Note links are denoted by dashed lines linking to these entities. The note itself is modeled as a vertex of type `note`. The note links as edges of type `connector`.

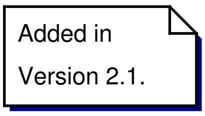


Figure 2.12: Example for a note in UML notation.

### 2.2.4 Other Model Elements

The UML specification contains much more semantic entities relevant for class diagrams which have not been discussed yet. Most of them are not discussed because they do not affect the automatic layout of a class diagram. There are two reasons why they do not play a role in the automatic layout:

**Embedded Elements.** These entities do not affect the topology of the graph and are exclusively rendered in the boundary of another semantic entity. Examples are the entities *Operation* and *Attribute.* These entities are rendered inside the rectangle visualizing the class they are belonging to. For example in Figure 2.5 the attribute state of the class `Counter` is contained in the middle compartment of the rectangle.

**Stereotypes.** The second types of entities are semantic entities whose visualization is similar to the visualization of one of the entities discussed and its semantics does not require special treatment in the visualization. Examples are the entities *Metaclass* and *Utility.* They are just stereotypes of class. A special role play the semantic entities *Object* and *Link.* Formally class diagrams can contain objects and links. A class diagram consisting purely of objects and links is called *object diagram* and can be seen as a special kind of class diagram. Class diagrams containing both, objects and classes, are rarely used, we therefore treat objects as classes and links as associations. Although this is not semantically correct it makes no difference for the visualization.

## 2.3 The **CLASS DIAGRAM LAYOUT** Problem

The problem of automatic layout of a class diagram can be stated as follows:

> Given a class diagram graph $\mathcal{C}$, find a class diagram layout $\Gamma$ for $\mathcal{C}$, that reveals the information in the diagram best.

Each graph drawing algorithm which can handle labels and respects prescribed vertex sizes is therefore a potential layout algorithm for class diagrams. But not all graph drawing algorithms are equally suited for automatic layout of class diagrams. This is due to the fact that the special aesthetic criteria of class diagrams are not necessarily covered by a graph drawing algorithm. In the remainder of this section we will discuss these aesthetic criteria and derive from them a formulation of the class diagram layout problem. Since class diagrams are drawings of graphs, we will first discuss aesthetics of drawings of graphs and then treat the special requirements of class diagrams.

There is no mathematical definition of aesthetics for a drawing of a graph, it can be defined informally that a drawing of a graph is more aesthetic than another drawing if it is ”nicer” or ”more readable”. To mathematically describe aesthetics the concept of *aesthetic criterion* is used. An aesthetic criterion measures one isolated mathematically defined property of the drawing and defines rules for the values of this property. Examples for aesthetic criteria are [33]:

- minimize number of edge crossings (CROSSING),
- minimize number of bends (BEND),
- minimize number of vertex and edge overlap (OVERLAP),
- maximize number of orthogonal edges (ORTHOGONAL),
- maximize angular resolution (RESOLUTION),
- minimize edge length (EDGE LENGTH),
- minimize area (AREA),
- maximize rectangular aspect-ratio (ASPECT RATIO),
- maximize number of edges respecting flow (FLOW),
- maximize symmetry (SYMMETRY).

Some of the above criteria are contradicting, e.g., area and crossing minimization [33]. Therefore finding an aesthetic drawing of a graph can be seen as solving a multi-objective optimization problem, the objective function being a set of aesthetic criteria. Which aesthetic criteria apply to a

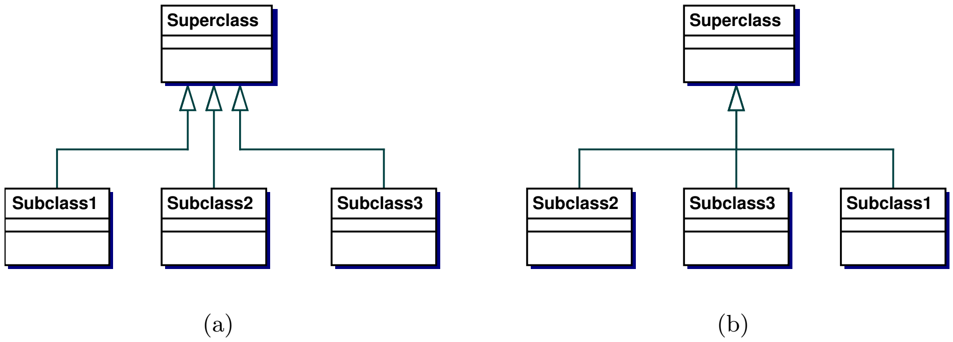


Figure 2.13: Generalization with distinct paths (a), hyperedge notation (b).

given drawing depends on the semantics of the graph and user preference. Aesthetic criteria more specific to class diagrams are discussed in [42]:

- use hyperedge notation for the generalization relation (HYPEREDGE),
- center the diamond vertex of n-ary association (CENTER),
- place notes and association classes near to the related model elements (PROXIMITY).

There are few empirical studies about aesthetic criteria, and how they affect the readability of a drawing. Purchase performed a series of experiments [102, 104, 105] about the impact of different aesthetic criteria for the readability of drawings. The first two experiments [102, 104] were performed for graphs without semantics, the experiment [105] was focused on class diagrams. In the experiments students had to answer questions about the graphs, resp. class diagrams, in a limited amount of time. The hypothesis is that if a given aesthetic criterion influences the readability in some sense, this should be reflected by the quality of the answers.

For graphs without semantics the aesthetic criteria BEND, CROSSING, RESOLUTION, ORTHOGONAL and SYMMETRY have been studied. In the experiments aesthetic criterion CROSSING was found to be by far the most important, aesthetic criteria BEND and SYMMETRY having lesser importance and ORTHOGONAL and RESOLUTION with no significant effect. For class diagrams, the situation is less clear. In [105] BEND and FLOW are evaluated to *decrease* the readability of a class diagram, ORTHOGONAL having no influence on the readability.

Another aspect of aesthetics of class diagrams is user preference. User preference is how an average user of class diagrams ranks aesthetic criteria. Note that user preference is only linked indirectly to the readability of a diagram, for example a user may prefer a certain visualization although it decreases readability, but in general it is assumed that a diagram is more readable for a user when it satisfies his personal preferences. In an experiment on user preferences for class diagrams [103] CROSSING ranked highest followed by BEND and HYPEREDGE.

As a conclusion we can say that it is not clear which is the most important aesthetic criterion for class diagrams. The aesthetic criteria CROSSING and BEND seem to play an important role. Although the importance for aesthetic criteria FLOW, HYPEREDGE and ORTHOGONAL is not backed by the above studies, partially because they have not been investigated, we think they play an important role for class diagrams and cannot be neglected. Because the situation is unclear we find it important for an automatic layout algorithm to be flexible enough to let the user choose the aesthetic criterion to optimize.

We introduce therefore the concept of *style*, which defines a mapping from a class diagram graph $\mathcal{C}$ to a layout graph $\mathcal{L}$ which is defined as:

- A mixed graph $G = (V, E)$,
- a subset $D \subseteq E$ denoting the directed edges,
- a subset $H \subseteq E$ denoting the edges which are considered to be part of a hyperedge,
- a set of labels $L$
- a function $T : L \rightarrow \{\texttt{source}, \texttt{center}, \texttt{target}\}$ denoting the preferred position of a label along the edge,
- a function $s : V \cup L \rightarrow \mathbb{N}^2$ denoting the size of the vertices, resp. labels, in the drawing.

In a style the semantics is taken away from the graph elements and is replaced by mathematical constructs. One possible style is to direct all generalization edges and draw them as hyperedges, while leaving all other types of edges undirected. In this case all edges $e$ with $etype(e) = \texttt{generalization}$ are contained in $D$ and $H$. Edges with different type are neither contained in $D$ nor in $H$. Another possibility is to direct the associations and keep the generalizations undirected. A style can distinguish between interfaces and classes, which is especially useful for visualizing class diagrams for Java programs. In a style to each label a preferred placement is assigned, which is normally the same for all styles: Labels of type `multiplicity` and `role` are always assigned to one end of an association. They receive therefore either the preferred placement `source` or `target`. All other labels receive the preferred placement `center`.

The styles described above should be rather seen as an example than as a definition. Through the use of styles as a intermediate transformation step we make it easy to integrate further model elements by enhancing the type table and the style mapping. The style defines the main aesthetic criterion for the diagram:

**Definition 2.5** *The* CLASS DIAGRAM LAYOUT *problem is defined as follows: Given a class diagram graph $\mathcal{C}$ and a style $\mathcal{S}$, find a class diagram layout $\Gamma(\mathcal{C})$ in which:*

- *there are no overlaps (OVERLAP),*
- *the edges of $D$ in $\mathcal{S}(\mathcal{C})$ point upward (FLOW),*
- *the edges of $H$ in $\mathcal{S}(\mathcal{C})$ are drawn as hyperedges (HYPEREDGE),*
- *the number of crossings is minimized (CROSSING), and*
- *the number of bends, the total area covered, and the total length of all edges in the drawing are minimal (BEND, EDGE LENGTH, AREA).*

The CLASS DIAGRAM LAYOUT problem is not tractable in general as we will see in Chapter 3. However, for the applications of automatic layout of class diagrams it is important to solve the problem in interactive time, in other words within a few seconds, on a common desktop computer. As typical size of class diagrams we can assume that the diagram contains less than 50 classes and that the diagrams are sparse. Sparse means that the number of relations in the diagram is only a constant factor higher than the number of classes in the diagram. This factor is for class diagrams usually below two. Diagrams with more classes than 50 are normally considered as too big by the user and split into multiple diagrams. The number of 50 classes is already a conservative upper bound, in practice diagrams with more 20 classes are hardly encountered in documentation of software systems. Diagrams which are not sparse reflect usually bad design [44] and usually do not have a good layout, only a not too bad layout.

## 2.4 Applying the Topology-Shape-Metrics Approach to Class Diagrams

In this section we review the topology-shape-metrics approach for drawing graphs and discuss how it can be applied to the automatic layout of class diagrams. The approach origins from the seminal paper of Tamassia [115], the name topology-shape-metrics approach was introduced in [33].

The topology-shape-metrics approach is one of the most popular graph drawing methods, it has been applied successfully to application domains like the visualization of data flow diagrams [5], database schemas [30, 31] and industrial plant schemas [37]. In a comparison of four graph drawing algorithms for orthogonal drawings, the one following the topology-shape-metrics approach was the clear winner [34].

The approach is motivated by the fact that for some applications the number of crossings is one of the most important aesthetic criteria, which is also true for class diagrams as we have seen in the previous section. The second aesthetic criterion in the topology-shape-metrics approach is the number of bends. Choosing the number of bends as optimization goal might surprise since it does not rank very high in the list of aesthetic criteria. However, drawings with a small number of bends tend also to be very compact having small total edge length and covering small area. Another advantage of optimization over the number of bends is, that for some special cases of the problem we can compute efficiently an optimal solution when the topology is fixed. For most other aesthetic criteria this is not possible.

The topology-shape-metrics approach is divided into the following three steps:

**Planarization** This step determines the topology of the drawing, which is described by a planar embedding. For non-planar graphs dummy vertices are inserted which represent crossings. Usually algorithms try to minimize the number of crossings.

**Orthogonalization** This step determines the angles and the bends in the drawing. Only multiples of 90° are assigned as angles which ensures that the drawing is orthogonal. Usually algorithms try to minimize the number of bends in this step.

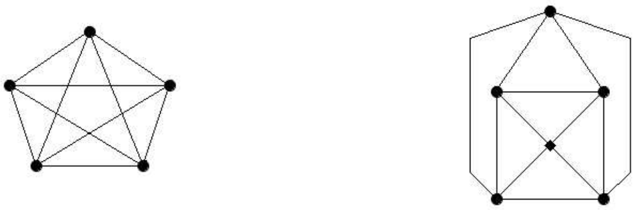
**Compaction** In this step the final coordinates are assigned to the vertices and to the edge bends. The dummy vertices introduced in the planarization step are removed. In this phase the main goal is to minimize the sum of the lengths of all edges and/or the area of the drawing.

The topology-shape-metrics approach avoids overlap and produces orthogonal drawings, therefore the aesthetic criteria ORTHOGONAL and OVERLAP are fulfilled. The algorithm is designed to optimize CROSSING, BEND, EDGE LENGTH and AREA.

If we want to apply the topology-shape-metrics approach to class diagrams we have to devise algorithms for the three phases which integrate the additional requirements of Section 2.3. We will now analyze these requirement with respect to the topology-shape-metrics approach.

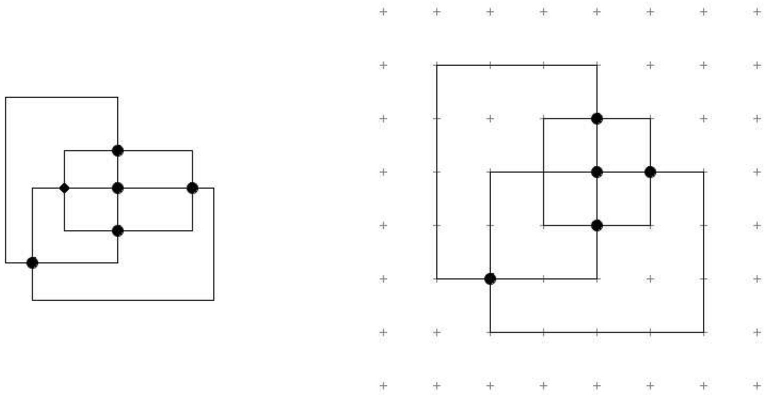
### FLOW Aesthetic Criterion

To satisfy the flow aesthetic criterion we have to take a look at the first and second phase. In the first phase an embedding of the graph must be calculated, which has a drawing in which all edges, for which the FLOW aesthetic



(a) input graph

(b) result of planarization



(c) result of orthogonalization

(d) result of compaction

Figure 2.14: The topology-shape-metrics approach.

criterion applies, point upward. The same holds for the orthogonalization step, it must calculate angles and bends accordingly.

An alternative approach are quasi-upward drawings as described in [7]. In quasi-upward drawings not all edges point upward, but upward directed edges emit from the upper half of the source vertex and connect to the lower half of the target vertex. In [7] a topology-shape-metrics approach for quasi-upward drawings with a minimal number of non-upward pointing edges is described.

The planarization algorithm we present assumes that the directed subgraph induced by the directed edges is acyclic and connected. If the directed graph induced by $D$ and $H$ contains directed cycles, not all directed edges can be drawn upward. We can therefore draw only a subset of the edges in $D$ and $H$ upward. Since the FLOW aesthetic criterion is very important, we want to minimize the number of edges we cannot draw upward. This is equivalent to the problem of finding the minimum number of edges we need to remove from $D$ and $H$, to make the subgraph induced by $D$ and $H$ acyclic. Unfortunately this problem, known as the *feedback arc set problem*, is NP-complete [80]. However, numerous heuristics have been proposed for this problem which work fairly well in practice. We use an algorithm based on depth-first-search as described in [62], and which has linear running time. If the subgraph induced by $D \cup H$ is not connected we add undirected edges to $D$ until $D$ is connected.

### HYPEREDGE Aesthetic Criterion

In order to handle hyperedges we can use a preprocessing technique. We substitute hyperedges by stars. All incoming edges of a vertex $u$ are removed from $u$ and connected to a *hyperedge vertex* $d$, which itself connects to $u$. Let $u \in V$ and $H(u) = \bigcup_{e=(v,u)\in E} e \in H$. For $H(u) = \{(v_1, u), \ldots, (v_k, u)\}$, we create a new vertex $d$, we substitute the edges $(v_i, u)$ with $(v_i, d)$, $1 \leq i \leq k$, and add the edge $(d, u)$. See figure 2.15 for an example.

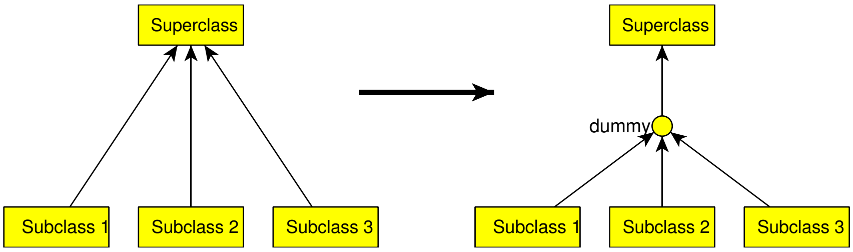


Figure 2.15: Substitution of edges forming a hyperedge by a star.

In the orthogonalization phase we must assign the correct angles and bends to the edges as shown in the example in Figure 2.13.

### Prescribed Node Size

To consider prescribed vertex sizes, the orthogonalization and the compaction phase must meet special requirements. The only methods that can guarantee prescribed vertex sizes rely on the `Kandinsky` model, also called podevsnef model. In this model the shape must conform to certain rules, the most important being that only one edge per side of the vertex has no bend. In the compaction phase the assigned vertex size must be realized.

### Labeling

Usually graph drawing systems perform labeling as a separate task after the drawings have been calculated and use a separate map labeling technique, for example [117]. Labeling can be integrated into the compaction step which usually yields better results than using a map labeling algorithm. Labeling is therefore handled in the compaction step of our algorithm.

### Dynamic Layout and Interactivity

The requirements dynamic layout and interactivity are somehow different to the other issues, and are therefore treated in a separate chapter. Interactivity requires that the user has some possibility to influence the result of the automatic layout algorithm to his wishes. The dynamic layout requirement states that if we have an automatically layouted diagram and change little of it, for example add an element to it, and then apply the automatic layout algorithm again, the resulting diagram still resembles the input diagram. Both requirements affect all three phases of the topology-shape-metrics approach and we will present a unified approach for this problem.

## 2.5 UML-`Kandinsky`

We are now able to outline UML-Kandinsky, our new algorithm for the automatic layout of class diagrams.

We assume that the directed graph induced by the predicate $H$ is acyclic. This assumption is justified, since the generalization relationship between classes is acyclic by definition and the hyperedge notation is restricted to generalization edges in the UML [95]. We assume furthermore that the input graph is connected. If this assumption is violated, we can divide the graph into its connected components and process each connected component separately by our algorithm. The diagrams of the connected components can then be arranged by a floor planning algorithm, for example [59].

This algorithm ignores interactivity, for interactive layout the algorithm for sketch-driven layout from Chapter 6 applies.

Algorithm 1: UML-`Kandinsky`

1. Preprocessing:
    (a) Remove edges from $D$ until the edges in $D \cup H$ induce an acyclic subgraph of $G$.
    (b) The edges in $H$ are substituted by stars.
    (c) If the edges in $D$ do not induce a connected subgraph add edges temporarily to $D$ to make this subgraph connected by using a minimum spanning tree algorithm.
2. Execute algorithm *mixed-upward-planarization* from Chapter 3 as planarization step.
3. Execute algorithm *mixed-upward orthogonalization* from Chapter 4 as orthogonalization step.
4. Execute algorithm *class diagram compaction* from Chapter 5 as compaction step.
5. Postprocessing
    (a) Remove all dummy vertices from the graph including: crossings, label vertices, hyperedge vertices, and artificial bends.
    (b) Place edge labels with preferred placement `source` or `target` with a map labeling algorithm.

## 2.6 Related Work

In this section we give a short overview over existing algorithms for the automatic layout of class diagrams.

### 2.6.1 Automatic Layout in UML-Tools

UML Tools, like TogetherJ from TogetherSoft or XDE from Rational, to mention the most popular ones, sometimes offer the possibility to automatically layout a diagram. Eichelberger evaluated 42 UML Tools in [43], including the above mentioned market leaders, and concludes that none of them provides automatic layout facilities which produce satisfactory results. In [70] some examples for major tools are presented, in which the automatic layout algorithm perform very badly. It is important to note that these examples are not some carefully constructed instances especially conceived to trouble the automatic layout algorithm of a certain tool. The examples are rather easy real world examples stemming from real world applications.

### 2.6.2 The Seemann Algorithm and its Enhancements

The Seemann algorithm and its enhancements are based on the hierarchical graph drawing approach. The hierarchical approach [114], also called Sugiyama approach, is an algorithmic framework for drawing directed acyclic graphs. It consists of three phases: *layer assignment*, *crossing minimization* and *vertex placement*. In the layer assignment phase each vertex $v$ in the graph is assigned a layer $l(v)$, such that all edges extend from a lower layer to a higher layer, in other words, $l(v) < l(w)$ for $(v, w) \in E$. In the second phase a permutation of each layer is computed, such that the number of crossings is minimized. The layering of the graph is usually normalized before this phase, which means that edges spanning more than one layer are split in a path by introducing dummy vertices, with each edge in the path spanning a single layer. In the third phase the coordinates for the vertices are determined. All vertices in a layer get the same y-coordinate and the x-coordinates are assigned according to the permutation calculated in the second phase. Coordinates of dummy vertices become bends of the original edge. Usually one tries to minimize the number of bends, area and the length of the edges in this phase.

The hierarchical approach optimizes the FLOW aesthetic criterion in the first place, and the CROSSING criterion in the second place. Criteria BEND, AREA and EDGE LENGTH are considered with lower priority. There are variants of the algorithm leading to orthogonal drawings and generally the algorithm generates no overlaps. Therefore OVERLAP is fulfilled and ORTHOGONAL can be fulfilled if desired.

In general in class diagrams not all edges need to be drawn according to aesthetic criterion FLOW, often only generalization edges are drawn according to FLOW. Especially some edges may be undirected, e.g. symmetric associations or links to association classes and notes. Therefore the hierarchical approach cannot be applied directly to class diagrams, modifications are needed to handle these observations.

The work of Seemann [111] was the first description of an enhanced version of the hierarchical approach which distinguishes between association and generalization edges. The algorithm works as follows: Nodes not adjacent to generalization edges are removed temporarily from the graph. Then the algorithm executes the first two phases of the hierarchical approach on the subgraph induced by the generalization edges. Then the removed vertices are inserted iteratively in the layers. As a last step vertex positions are calculated and the edges routed. Generalization edges are drawn as direct lines while association edges are drawn orthogonal. This algorithm is generalized by SugiBib [41, 42] to handle additional constraints like aesthetic criteria HYPEREDGE, CENTER and PROXIMITY and to our knowledge it

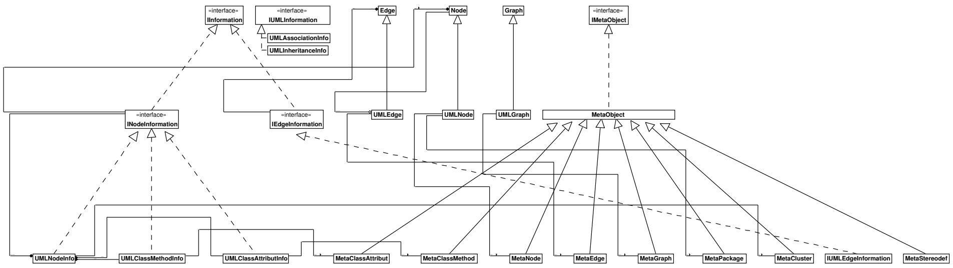


Figure 2.16: Example Layout of SugiBib taken from [41].

is the most sophisticated hierarchical layout algorithm for class diagrams at the moment. Furthermore it supports clustering and the edges denoting hierarchy are no longer restricted to generalizations. It rather divides the edges into hierarchical and non-hierarchical edges. Usually generalization edges are hierarchical and all others are not.

The reason to use the hierarchical approach for the visualization of class diagrams is based on two assumptions:

1. There is a large set of hierarchical edges in the graph defining a deep hierarchy, and

2. the users most important aesthetic criterion for the diagram is aesthetic criterion FLOW for the hierarchical edges.

These assumptions are often violated in practice. Class diagrams may contain no hierarchical structure at all, for example when the diagram contains only associations. Note that the extensive usage of inheritance in software engineering is discouraged. Gamma [61] notes:

> Favor object composition over class inheritance.

Even diagrams with a lot of hierarchical edges may be a problem for the hierarchical approach, since these hierarchies are usually not deep. Software with deep inheritance hierarchies tend to be difficult to understand and hard to maintain. Booch [11] reports that as a rule of thumb inheritance hierarchies in object-oriented software systems have a maximum depth of 7±2. Empirical studies show that the value is even smaller, in [23] it is shown that a value of three is more realistic in practice. This has as a consequence that only few layers are allocated by the above algorithms and when the size of the diagrams grow, the diagrams get wider but not higher resulting in diagrams with bad aspect ratio violating aesthetic criterion ASPECT RATIO. This can be avoided by using advanced layering algorithms which take the aspect ratio of the diagram into account. It is not clear how such a layering algorithm would look like, there are some approaches solving this problem, for example [27, 72], but it is not evident how they integrate in the above layout algorithm for class diagrams.

Even if there are enough hierarchical edges in the diagram, the user might not want to optimize aesthetic criterion FLOW, therefore assumption 2 is often violated. As noted in Section 2.3 aesthetic criterion CROSSING might influence the readability more than aesthetic criterion FLOW, for example the user might prefer a drawing with less crossings over a drawing with inheritance edges pointing upward. The hierarchical layout algorithms lack this flexibility by design.

A more technical argument is that for upward directed drawings the hierarchical graph drawing algorithm may produce more crossings than the upward planarization approach we suggest. In a study comparing the two algorithms [50] the upward planarization approach produced less crossings, especially for graphs with limited height. Note that we compared in this study the graph drawing algorithms and not the algorithms for drawing class diagrams, so these results have to be interpreted with care, but nevertheless they suggest that the upward planarization approach might be superior to the hierarchical approach to minimize the number of crossings.

### 2.6.3 GoVisual

The GoVisual tool for the automatic layout of class diagrams has been published in parallel to our work and is also based on the topology-shape-metrics approach. There is only a superficial description [69, 70] of the algorithm available. While the planarization phase is described in some detail, the orthogonalization and compaction phase are only roughly sketched.

The planarization algorithm is based on the concept of mixed upward planarity we introduced in [47]. The GoVisual algorithm calculates an upward embedding for each connected component of the subgraph induced by the directed edges. First an upward planarity test [8] for single-source digraphs is applied. If this test fails, a planarization technique is applied. The authors refer to our approach [47] and to techniques stemming from the hierarchical approach. After computing an upward planarization, the undirected edges between vertices in the same subgraph are inserted. Finally the edges having vertices in two different subgraphs are inserted using techniques for planarization of clustered graphs [29]. The algorithms used in the planarization phase make strong usage of advanced data-structures like SPQR-trees [36] and PQ-trees [13].

The other two phases use a reduction approach of Tamassia's algorithm as described in Section 4.2, but no details are revealed how the HYPEREDGE and FLOW aesthetic criteria, prescribed vertex sizes and label placement, are treated in these phases. Only pointers to basic algorithms not covering the above challenges are given, and it is noted in [70]:

> The second step performing the computation of the orthogonal

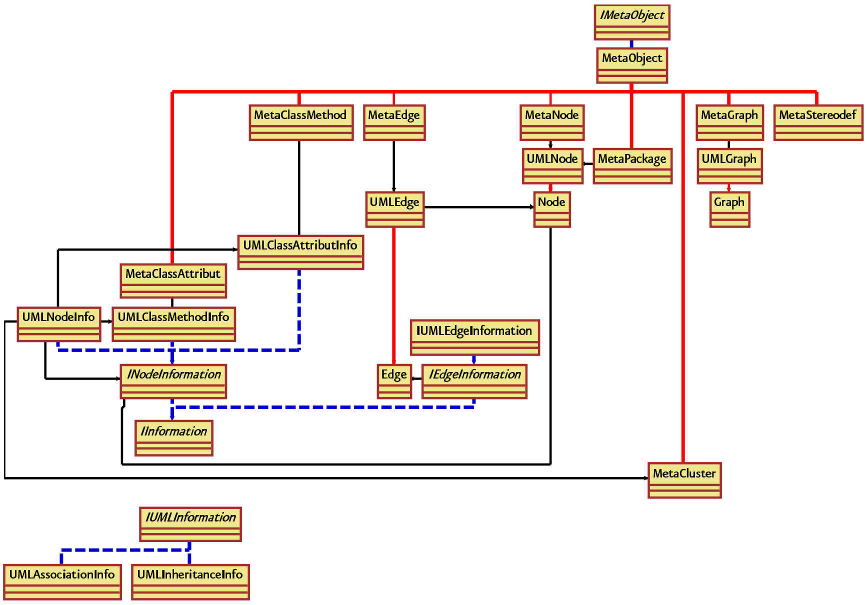


Figure 2.17: Example of GoVisual taken from [70].

> layout is more tedious due to complex implementation details, and we therefore give more room to the first part.

In the GoVisual approach only directed edges which are in the same connected component point in the same direction. Therefore the FLOW aesthetic criterion is only adhered locally for each connected component, but not globally for the entire diagram. However, the approach guarantees that no two class hierarchies are nested within each other: A class hierarchy is not enclosed by a circle (in the undirected sense) of arcs of a different hierarchy.

Figure 2.17 shows the GoVisual layout of the class diagram from Figure 2.16. The layout is clearly an improvement of the original layout in terms of crossings, bends, total edge length and area. It underlines that the topology-shape-metrics approach is promising for the automatic layout of class diagrams. However, the fact that the GoVisual approach guarantees only locally consistent directions for directed edges is sometimes a disadvantage. In Figure 2.17, for example, there is a single generalization edge pointing from `UMLEdge` to `Edge`. In a zoom level in which the arrow head is not clearly visible, a user cannot infer the direction of the inheritance relation. Even worse, if the user assumes that the generalization relationship is always drawn upward, the user may conclude that the relationship has the wrong direction.

Apart from this drawback the GoVisual approach is a clear improvement of the hierarchical approach. However, there are two practical hurdles for people who want to implement this approach: First the algorithm relies on sophisticated data-structures which are very complicated to implement. Second a lot of details important for the implementation of the algorithm are concealed. A reason for this might be that GoVisual is a commercial product and the authors have no interest in revealing too many details of it.

# Chapter 3

# Mixed Upward Planarization

In this chapter we consider the problem of finding a planarization of a mixed graph for which a drawing exists in which all directed edges are represented by monotonically increasing curves and which has a low number of crossings at the same time. The edges in the graph are weighted according to model the difference of importance of different types of edges. Planarization has been only studied for undirected graphs until now and we present the first algorithm for the planarization of mixed graphs, which includes the important special case of planarization of directed graphs.

Our presentation follows partly [47, 50]. We define our problem formally in Section 3.1 which will lead us to the WEIGHTED MIXED UPWARD CROSSING MINIMIZATION problem. Our algorithm is based on a heuristic which is a popular technique for the planarization of undirected graphs:

1. Construct an embedded planar subgraph.
2. Insert the edges not contained in the subgraph, one by one.
3. Use rerouting to reduce the number of crossings.

Since the crossing minimization problem is NP-complete [64], we cannot hope to find an efficient exact algorithm.

In the first step of the heuristic a planar subgraph of the input graph is calculated. Unfortunately finding a planar subgraph with the maximum number of edges, which is called the maximum planar subgraph problem, is NP-hard [63] even in the undirected unweighted case. We will study this problem for weighted mixed graphs in Section 3.2 and present a heuristic for it.

In the second step, the edges which are not part of the subgraph are inserted incrementally into the embedding. In Section 3.3 we will review algorithms for the insertion of undirected edges into undirected graphs and present an algorithm for the insertion of directed edges into a directed graph.

In the third step some local optimizations on the resulting planarization are performed to improve the quality of it. This technique, known as *rerouting*, is covered in Section 3.4

In Section 3.5 we summarize the results of the preceding sections and present the complete algorithm for the planarization of mixed graphs.

## 3.1 Mixed Upward Planarity

In Section 2.1.4 we introduced planar graphs and planarization. These concepts are defined for undirected graphs. In this section we will generalize these concepts to mixed graphs.

We first review the well-known concept of *upward planarity*, which defines planarity for directed graphs, before we concentrate on the mixed case. An *upward drawing* of a directed graph is a drawing in which each edge is represented by a curve monotonically increasing in the vertical direction. Note that such a drawing exists if and only if the directed graph is acyclic.

A drawing of a directed graph is *upward planar* if the drawing is both, upward and planar. A directed graph is *upward planar* if it has an upward planar drawing. Note that there are graphs which have an upward drawing and a planar drawing but which have no upward planar drawing, see Figure 3.1 for an example.

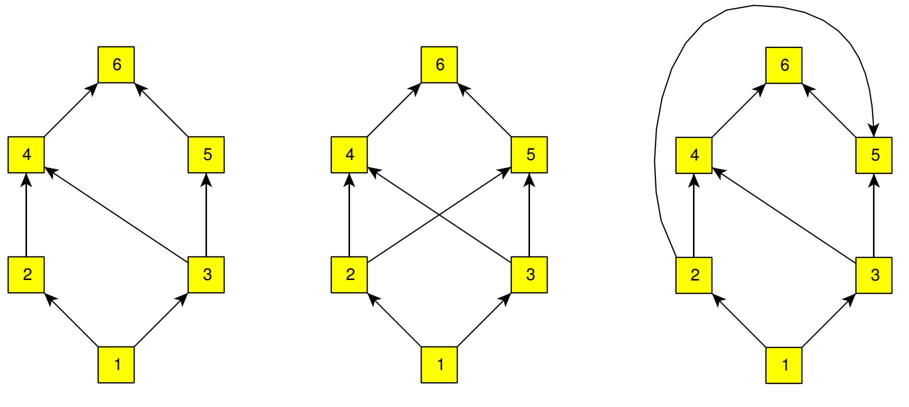
Upward planarity has been studied extensively and while it can be tested efficiently for some special classes of directed graphs, for example single source digraphs [8, 74], outerplanar graphs [96], or planar bipartite graphs [35], the problem is not tractable in the general case:

**Theorem 3.1 ([65])** *The decision problem if a directed graph is upward planar is NP-hard.*

When we look at the cyclic order of the edges around a vertex in an upward planar drawing we notice that the incoming and the outgoing edges form an interval. We call an ordering of the adjacent edges of a vertex with this property *bimodal.* If all vertices of an embedded directed graph $G$ are bimodal, then $G$ is bimodal.

An *upward embedding* of a directed graph is a *linear ordering* of the adjacent edges of each vertex of the graph which can be divided into two parts: the first part consisting of the outgoing edges and the second part of the incoming edges.

An upward embedding is *upward planar* if there is an upward planar drawing of the graph which preserves the corresponding ordering around each vertex. This means that the linear order is equivalent to the order that is obtained by ordering the edges according to the angle they form in the drawing with a ray leaving the vertex in direction of the positive x-axis.



(a) (b) (c)

Figure 3.1: An upward planar drawing of a directed graph (a). Adding the edge $(2,5)$ makes this graph non-upward planar although it has an upward drawing (b) and a planar drawing (c).

Planarity for mixed graphs has not been studied until now, previous work concentrated either on directed or undirected graphs. Since we are interested in mixed graphs, we introduce the concept of *mixed upward planarity* as a generalization of planarity and upward planarity.

**Definition 3.1** *A* mixed upward drawing *of a mixed graph $G$ is a drawing in which each directed edge of $G$ is represented by a curve monotonically increasing in the vertical direction. A mixed graph $G$ is called* mixed upward planar *if it has a planar mixed upward drawing.*

It follows directly from the definition that in the special case that all edges in the graph are undirected, resp. directed, a graph is mixed upward planar if, and only if, it is planar, resp. upward planar. Therefore mixed upward planarity is a natural generalization of planarity and upward planarity. Because the decision problem whether a directed graph is upward planar is a special case of the decision problem if a mixed graph is mixed upward planar it follows:

**Corollary 3.1** *The decision problem if a mixed graph is mixed upward planar is NP-hard.*

A *mixed upward embedding* of a mixed graph $G$ is a linear ordering of the adjacent edges of each vertex, which is an upward embedding for the directed subgraph of $G$. A mixed upward embedding is *mixed upward planar* if there is a mixed upward planar drawing of the graph which preserves the corresponding ordering. A *mixed upward planarization* of a mixed graph $G$ is a planarization of $G$ which is mixed upward planar.

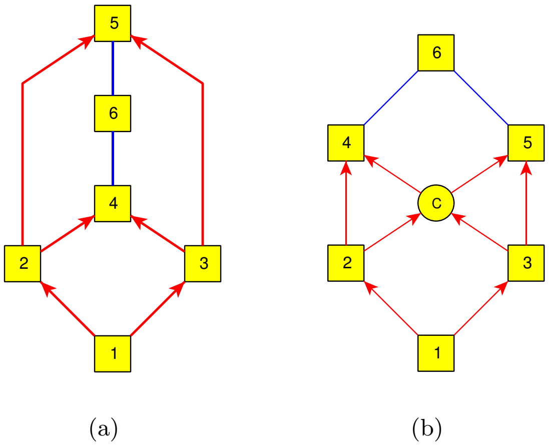


Figure 3.2: A mixed upward planar drawing of the mixed graph derived from undirecting the edges (4, 6) and (5, 6) of the graph from Fig. 3.1 (a). A mixed upward planarization of the same graph (b).

In a planarization of a weighted graph, the weight of a crossing is the product of the weight of the crossing edges. The weighted crossing number of a planarization of a weighted graph is the sum over the weights of the crossings in the planarization. Determining for a weighted mixed graph $G$ a mixed upward planarized graph with minimal weighted crossing number is called the WEIGHTED MIXED UPWARD CROSSING MINIMIZATION problem.

**Definition 3.2** *The WEIGHTED MIXED UPWARD CROSSING MINIMIZATION problem for a mixed graph* $G = (V, E_d \cup E_u)$ *and a weight function* $w : E_d \cup E_u \rightarrow I\!N$ *is to find a mixed upward planarization of* $G$ *such that the weighted sum over all crossings is minimal.*

Because the decision problem whether a mixed graph is mixed upward planar is a special case of this problem it follows:

**Corollary 3.2** *The WEIGHTED MIXED UPWARD CROSSING MINIMIZATION problem is NP-hard.*

## 3.2 Maximum Mixed Upward Planar Subgraph

In this section we cover the problem of finding a mixed upward planar subgraph of a weighted mixed graph with the maximum edge-weight. This problem is defined as follows:

**Definition 3.3** *The WEIGHTED MAXIMUM MIXED UPWARD PLANAR SUBGRAPH problem for a mixed graph* $G = (V, E_d \cup E_u)$ *and a weight* *function* $w : E_d \cup E_u \to I\!N$ *is to find sets* $E'_d \subseteq E_d$ *and* $E'_u \subseteq E_u$ *with the property that the mixed graph* $G = (V, E'_u \cup E'_d)$ *is mixed upward planar and the sum* $\sum_{e \in E'_d \cup E'_u} w(e)$ *is maximal.*

The WEIGHTED MAXIMUM MIXED UPWARD PLANAR SUBGRAPH problem is a natural generalization of the well studied maximum planar subgraph problem. Unfortunately already the maximum planar subgraph problem is NP-complete [63], and therefore also the WEIGHTED MAXIMUM MIXED UPWARD PLANAR SUBGRAPH problem.

**Corollary 3.3** *The WEIGHTED MAXIMUM MIXED UPWARD PLANAR SUBGRAPH problem is NP-hard.*

Several heuristics have been proposed for the solution of the maximum planar subgraph problem: [68, 76, 77, 79, 107]. One of them, [77], can also compute the optimal solution when no time limit is specified. Cimikowski [25] compared some of these algorithms empirically. In his comparison the algorithm of Jünger and Mutzel (JM) [77] performed best in solution quality, followed by the algorithm of Goldschmidt and Takvorian (GT) [68]. The fastest algorithm was the one based on PQ-trees [68], but its performance in terms of the solution quality was significantly lower than JM and GT. Resende and Ribero give a randomized formulation of GT in [107] . They show on the same test set as [25] that their formulation achieves better results as JM with the same running time performance, except for one family of graphs where JM performs better. Both algorithms, JM and GT, are able to consider edge weights.

However, the algorithm of GT is much easier to implement compared to the algorithm of JM, which is a branch-and-cut algorithm and is based on sophisticated algorithms for linear programming for this reason. Because of its performance and its implementation issues, we use GT as a starting point. We first review the GT algorithm and then show how it can be modified to calculate mixed upward planar subgraphs.

### 3.2.1 The Goldschmidt/Takvorian Planarization Algorithm

In this section, we review the main components of GT, the two-phase heuristic of Goldschmidt and Takvorian [68]. Our description follows the one in [107]. The first phase of GT consists of devising an ordering $\Pi$ of the set of vertices of $V$ of the input graph $G$. This ordering should possibly infer a Hamiltonian path. The vertices of $G$ are placed on a vertical line according to the ordering $\Pi$ obtained in the first phase, such that as many edges as possible between adjacent vertices can also be placed on the line. All other edges are drawn as arcs either right or left of the line.

The second phase of GT partitions the edge set $E$ of $G$ into subsets $\mathcal{L}$ (left of the line), $\mathcal{R}$ (right of the line), and $\mathcal{B}$ (the remainder) in such a way

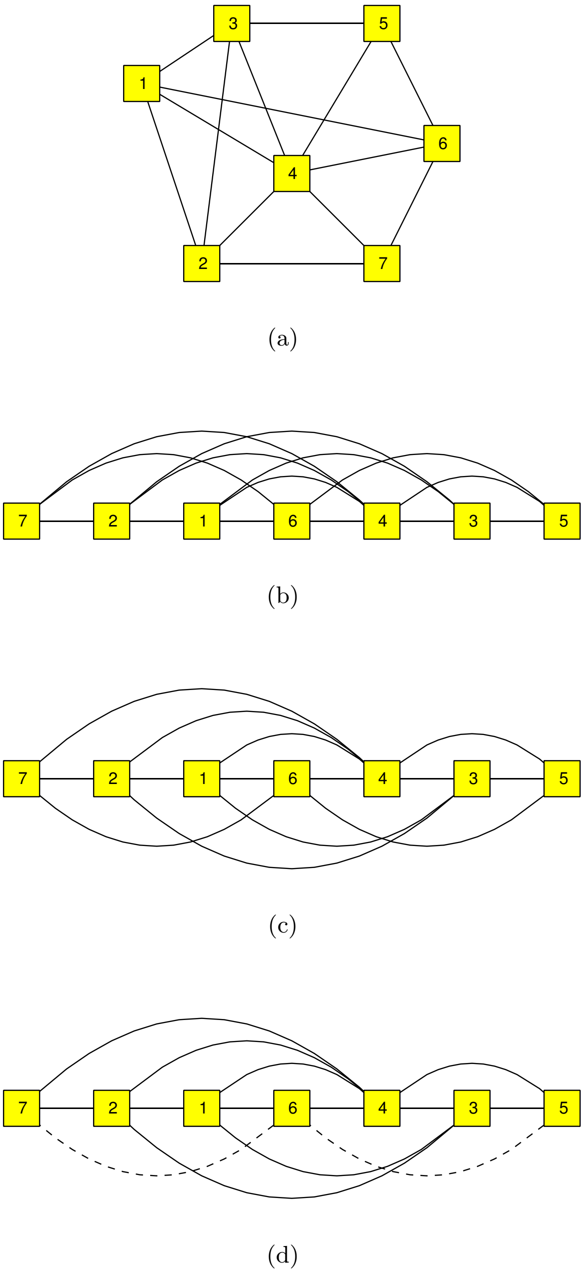


Figure 3.3: The above figures illustrate a run of the GT algorithm on an example input. The input graph $G$ is shown in Figure (a). The ordering $\Pi$ computed in the first phase of GT is $(7, 2, 1, 6, 4, 3, 5)$. Figure (b) depicts the corresponding drawing of $G$ for $\Pi$. The first independent set $\mathcal{L} = ((7, 4), (7, 2), (2, 1), (1, 6), (6, 4), (4, 3), (3, 5), (2, 4), (1, 4), (4, 5))$ is defined by the arcs above the line defined by the vertices in Figure (c). The second independent set $\mathcal{R} = ((2, 3), (1, 3))$ is defined by the solid arcs below this line in Figure (d). The remaining edges $\mathcal{B} = ((7, 6), (6, 5))$ are dashed.

that $|\mathcal{L}+\mathcal{R}|$ is large (ideally maximum) and that no two edges both in $\mathcal{L}$ or both in $\mathcal{R}$ cross with respect to the sequence $\Pi$ devised in the first phase.

Let $\pi(v)$ denote the relative position of vertex $v \in V$ within vertex sequence $\Pi$. Furthermore, let $e_1 = (a, b)$ and $e_2 = (c, d)$ be two edges of $G$, such that, without loss of generality, $\pi(a) < \pi(b)$ and $\pi(c) < \pi(d)$. These edges are said to *cross* if, with respect to sequence $\Pi$, $\pi(a) < \pi(c) < \pi(b) < \pi(d)$ or $\pi(c) < \pi(a) < \pi(d) < \pi(b)$.

The *conflict graph* has a vertex for every edge in $G$ and two vertices are adjacent if the corresponding edges cross with respect to $\Pi$. It follows directly from its definition that the conflict graph is an *overlap graph*, i.e. a graph whose vertices can be represented as intervals, and two vertices are adjacent if, and only if, the corresponding intervals intersect but none of the two is contained by the other.

An induced bipartite subgraph of the conflict graph represents a valid assignment of the edges in $G$ to the sets $\mathcal{L}$, $\mathcal{R}$ and $\mathcal{B}$. Since finding a maximum induced bipartite subgraph is NP-complete, even for overlap graphs, GT uses a heuristic. This heuristic calculates two disjoint independent sets of the conflict graph which, together, are a bipartite subgraph of the conflict graph. Figure 3.3 shows an example execution of the GT algorithm.

With the algorithm of Asano, Imai and Mukaiyama [3] a weighted maximum independent set of an overlap graph can be calculated in time $O(NM)$, where $N$ is the number of different interval endpoints and $M$ is the number of edges in the overlap graph. In our setting, $N \leq n$ and $M \leq m$, which leads to a running time of $O(nm)$. We refer to this algorithm as `MIS`. Since the above algorithm computes a weighted maximum independent, the GT algorithm can compute weighted planar subgraphs.

The performance of the whole algorithm can be improved by using randomization and local search techniques, as described in [107]. The randomization takes place in the first phase of the algorithm, where the vertex order is computed. Instead of calculating the planar subgraphs for only one ordering, we calculate planar subgraphs from a whole set of orderings and return the largest of them.

### 3.2.2 The Algorithm for Mixed Graphs

We now present our variant of the GT algorithm for mixed upward planar subgraph calculation. We call this variant *mixed GT* (`MGT`) to distinguish it from the original formulation.

In order to ensure that the planar subgraphs computed by the GT algorithm are mixed upward planar it suffices to consider the first step of GT, the construction of the vertex order. If no directed edge has a target vertex which is in the order before the source vertex, the resulting subgraph is upward planar.

**Lemma 3.1** *Let $G$ be a mixed graph. If the vertex order $\Pi$ in the first phase of the GT algorithm is a topological order of the subgraph $(V, E_d)$ of $G$, the result of GT is a mixed upward planar subgraph of $G$.*

**Proof:** Placing the vertices on a vertical line according to the ordering used by GT and drawing the edges in $\mathcal{L}$ as arcs on the left side of the line and the edges in $\mathcal{R}$ on the right side of the line yields an upward planar drawing of the subgraph calculated by GT. □

```
Algorithm 2: MVO
Input: A mixed graph $G = (V, E_d \cup E_u)$
Output: An ordering $\Pi$ on the vertices
$\bar{d} = \min_{\{v \in V | \delta^-_G(v) = 0\}} \{\delta_G(v)\}$
$CAND = \{v \in V | \delta^-_G(v) = 0 \text{ and } \delta_G(v) = \bar{d}\}$
$v_1 = \texttt{random}(CAND)$
$\mathcal{V} = V \setminus \{v_1\}$
$G_1$ = mixed graph induced on $G$ by $\mathcal{V}$
for $k = 1, \dots, |V| - 1$ do
    $\mathcal{U} = \{v \in \mathcal{V} | \delta^-_{G_k}(v) = 0\}$
    $CONN = adj(v_{k-1}) \cap \mathcal{U}$
    if $CONN \neq \emptyset$ then
        $\bar{d} = \min_{v \in CONN} \{\delta_{G_k}(v)\}$
        $CAND = \{v \in CONN | \delta_{G_k}(v) = \bar{d}\}$
    else
        $\bar{d} = \min_{v \in \mathcal{U}} \{\delta_{G_k}(v)\}$
        $CAND = \{v \in \mathcal{U} | \delta_{G_k}(v) = \bar{d}\}$
    $v_{k+1} = \texttt{random}(CAND)$
    $\mathcal{V} = \mathcal{V} \setminus v_{k+1}$
    $G_{k+1}$ = mixed graph induced on $G$ by $\mathcal{V}$
return $\Pi = (v_1, v_2, \dots, v_{|V|})$
```

The original version of GT [68] uses the following deterministic heuristic to find a Hamiltonian cycle, denoted by the ordering $\Pi$, in the input graph: The first vertex in $\Pi$ is a vertex with minimal degree in $G$. After the first $k$ vertices of the ordering have been determined, say $v_1, v_2, \dots v_k$, the next vertex $v_{k+1}$ is selected from the vertices adjacent to $v_k$ in $G$ having the least adjacencies in the subgraph $G_k$ of $G$ induced by $V \setminus \{v_1, v_2, \dots, v_k\}$. If there is no vertex in $G_k$ adjacent to $v_k$ in $G$, then $v_{k+1}$ is selected as a minimum degree vertex in $G_k$.

To use this algorithm for the mixed case we have to modify it in a way such that the resulting ordering is a topological ordering on the directed subgraph. We propose algorithm *mixed vertex order* (`MVO`), which uses an idea of a standard topological sorting algorithm to solve this problem: As in the original algorithm the ordering is constructed incrementally. After the first $k$ vertices of the ordering have been determined, say $v_1, v_2, \dots v_k$, we select a vertex $v_{k+1}$ with no incoming edge from a vertex in $G_k$. Again, as in the original algorithm, we prefer vertices, which are connected to $v_k$ and have minimal degree.

The algorithm sketched above adds the first vertex which fulfills the requirements to the sequence. We randomize the algorithm by first creating a set of candidate vertices which fulfill the requirements, and then choosing randomly one vertex from the candidate set.

Algorithm `MGT` computes *MaxIterations* vertex-orders and computes for each vertex order a mixed-upward-planar subgraph. The mixed-upward-planar subgraph with the greatest weight is returned as result. *MaxIterations* is a constant and therefore the randomization does not increase the running time of the algorithm.

**Lemma 3.2** *The vertex order* $\Pi$ *calculated by algorithm* `MVO` *is a topological order of the subgraph* $(V, E_d)$ *of* $G$*.*

**Proof:** For each vertex $u \in V$ holds that $u \in \mathcal{V}$ until iteration $\Pi(u) - 1$ and $u \in \mathcal{U}$ in iteration $\Pi(u) - 1$ of the for loop. Assume that there is an edge $e = (v, w) \in E_d$ with $\Pi(v) = l$, $\Pi(w) = k$, and $l > k$. Then $v, w \in \mathcal{V}$ at iteration $k - 1$ of the for loop and it follows $(v, w) \in G_{k-1}$. But then $\delta^-_{G_{k-1}}(w) > 0$ and it follows that $w \notin \mathcal{U}$ which is a contradiction to our first statement. □

From the sets $\mathcal{L}$ and $\mathcal{R}$ and the ordering $\Pi$, we can now easily obtain the mixed upward embedding: For each vertex $v \in V$ we sort the edges with source $v$ in $\mathcal{R}$ decreasing according to $\Pi$ and the edges with source $v$ in $\mathcal{L}$ increasing according to $\Pi$ and concatenate these two ordered lists to one. For the incoming edges, we first sort the edges with target $v$ in $\mathcal{L}$ increasing according to $\Pi$ and the edges with source $v$ in $\mathcal{R}$ decreasing according to $\Pi$ and append the result to the list of outgoing edges.

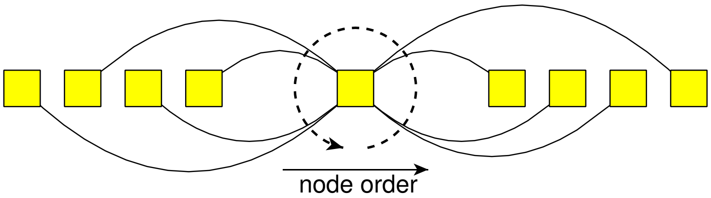


Figure 3.4: The order of the edges at a vertex can be derived directly from $\Pi$ and the sets $\mathcal{L}$ and $\mathcal{R}$.

We conclude the section with the following theorem:

Algorithm 3: Algorithm `MGT`

```
Input: Mixed graph $G = (V, E_d \cup E_u)$, weight $w : E_d \cup E_u \to \mathbb{N}$
Output: Mixed upward planar subgraph $G' = (V, E'_d \cup E'_U)$ of $G$,
        ordering $\Pi$
$\mathcal{L}_{max} = \emptyset$;
$\mathcal{R}_{max} = \emptyset$ ;
for $i = 0, \ldots, MaxIterations - 1$ do
    $\Pi = \mathtt{MVO}(G)$ ;
    $\mathcal{L} = \mathtt{MIS}(V, E_d \cup E_u, w)$ ;
    $\mathcal{R} = \mathtt{MIS}(V, (E_d \cup E_u) \setminus \mathcal{L}, w)$ ;
    if $|\mathcal{L} + \mathcal{R}| > |\mathcal{L}_{max} + \mathcal{R}_{max}|$ then
        $\mathcal{L}_{max} = \mathcal{L}$ ;
        $\mathcal{R}_{max} = \mathcal{R}$ ;
$E'_d = \{\mathcal{L}_{max} \cup \mathcal{R}_{max}\} \cap E_d$;
$E'_u = \{\mathcal{L}_{max} \cup \mathcal{R}_{max}\} \cap E_u$;
```

**Theorem 3.2** *Algorithm `MGT` computes an embedded mixed upward planar subgraph of a mixed graph in time $O(nm)$.*

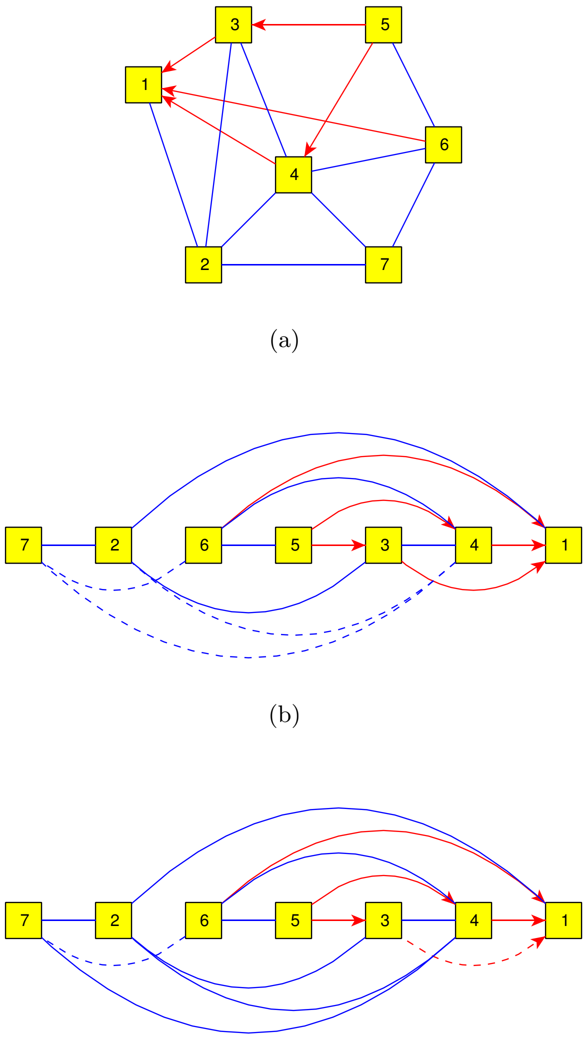
## 3.3 Edge Insertion

In this section we cover the problem of inserting an edge into an embedded graph. We first consider the well-known problem of inserting a single edge into an undirected plane graph and review a folklore algorithm for its solution. We then present our algorithm for inserting directed edges into an upward planar graph.

### 3.3.1 Insertion of Undirected Edges

In this section we treat the problem of inserting a single, undirected edge into a plane graph. There are two variants of the problem: in the first variant the embedding of the input graph is allowed to change, in the second variant the embedding is not subject to change. The first variant is more difficult and only recently a complicated linear time algorithm for this problem has been proposed [71]. However we cannot use this variant to insert an undirected edge into a mixed upward planar graph since it is not guaranteed that the resulting embedding is still mixed upward planar. This is not a flaw of the insertion algorithm, this results from the NP-completeness of the mixed upward planarization problem.

Therefore the second variant applies to us. It is well known that inserting a single undirected edge in a plane graph with the minimal number of



(a)

(b)

(c)

Figure 3.5: Example for `MGT`. Figure (a) shows the input graph, which is the example graph from Figure 3.3 with some edges oriented. The result of `MGT` with high edge weights for directed edges is depicted in Figure (b). If we use uniform edge weights, the result is different, only two edges are in $\mathcal{B}$ as shown in Figure (c).

crossings can be done in linear time, when the embedding is not allowed to change. We call the algorithm to solve this problem *undirected edge insertion* (`UEI`).

The algorithm is based on the *dual graph* of the input graph. The *dual graph* $G^*$ of a plane graph $G = (V, E, F)$ has a vertex for each face of $G$, and an edge $d(e) = (f, g)$ between two faces $f$ and $g$ for each edge $e$ that is shared by $f$ and $g$. It follows directly from the construction that the dual graph of a planar graph has linear size.

To insert an edge $(a, b)$, we construct from $G^*$ the *extended dual graph* $G^*_{(a,b)}$. This is done by adding two vertices $a'$ and $b'$ to $G^*$ and inserting an edge from $(a', f)$, resp $(b', f)$, into $G^*$ for each $f$ which contains an edge adjacent to $a'$, resp. $b'$. We assign to each edge $d(e)$ in the dual graph the product of the weight of edge $e$ and edge $(a, b)$.

From each path $e_0, \ldots, e_k$ from $a'$ to $b'$ in $G^*_{(a,b)}$ we can obtain a planarization of $G$ by subdividing the edges in $G$ corresponding to $e_1, \ldots, e_{k-1}$ and adding a path from $a$ to $b$ which uses the vertices introduced by the subdivision. A shortest path from $a'$ to $b'$ induces therefore a planarization of $G \cup (a, b)$ with minimal weighted crossing number without changing the embedding of $G$. The algorithm is illustrated in Figure 3.6.

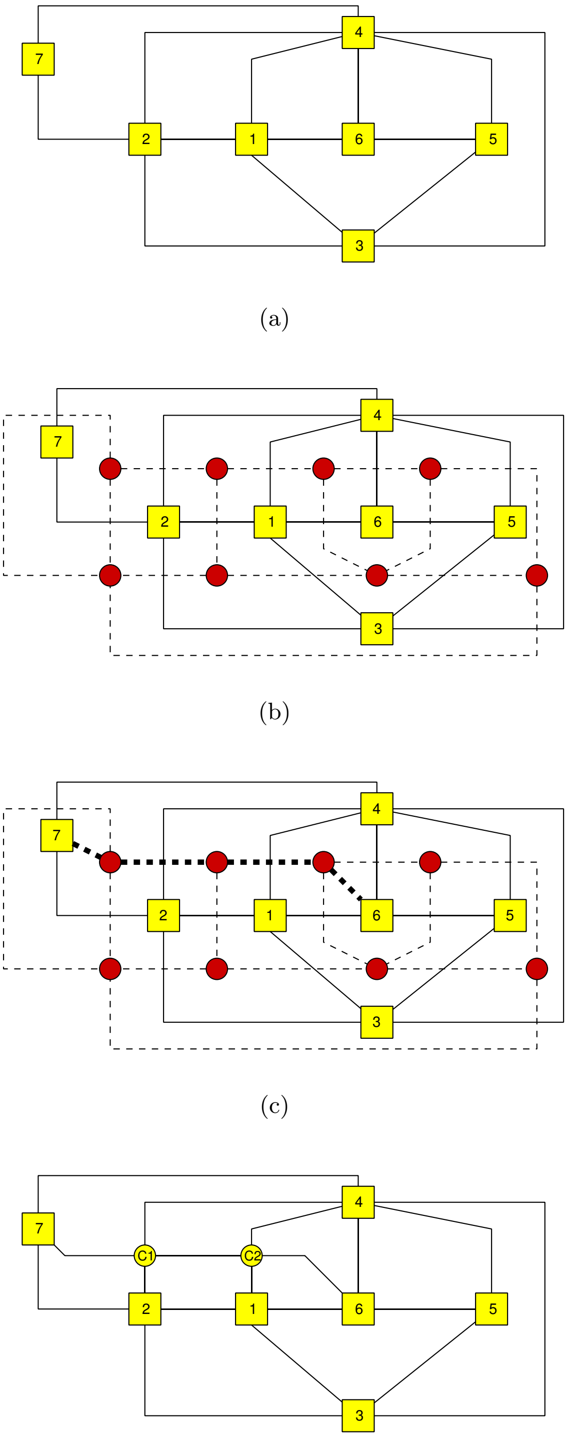
### 3.3.2 Insertion of Directed Edges

In this section we treat the problem of inserting directed edges into an upward embedded directed graph, which is a much harder problem than inserting undirected edges in a plane graph. This is due to the fact that in the undirected case the edges which are not part of the planar subgraph in the first step can be inserted independently of each other. This does not hold for the directed case. Here, we cannot insert an edge into the drawing without looking at the remaining edges which have to be inserted later. The reason for this is that the dummy vertices added to the graph in a planarization step introduce changes in the ordering of the vertices of the graph. Although the graph remains acyclic, this may introduce a directed cycle with an edge which will be added later. Then this edge can no longer be inserted.

For example assume that in Figure 3.7(a) the dashed edges have to be inserted, and we start by inserting edge $(5, 9)$. If we insert edge $(5, 9)$ as in Figure 3.7, we produce a crossing $C$ with edge $(1, 3)$. Then, it is no longer possible to introduce edge $(3, 4)$ without destroying the upwardness property because of the new directed cycle $5 - C - 3 - 4 - 5$.

For the remainder of the section, $G = (V, E, F)$ is an upward embedded directed graph, $R \subseteq V \times V$ denotes a set of edges to be inserted into $G$, and $w : E \cup R \to \mathbb{N}$ a weight function of the edges. Furthermore we assume that the graph $(V, E \cup R)$ is acyclic and that $G$ is an $st$-graph.

Since $G$ is an upward embedded $st$-graph, each face $f$ has exactly one



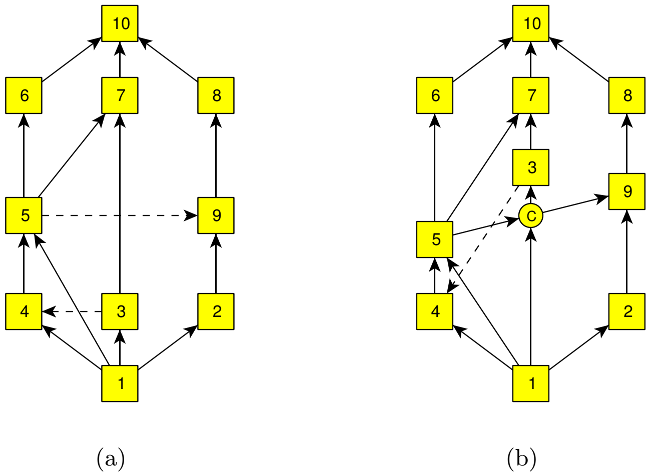
(a)

(b)

(c)

(d)

Figure 3.6: Insertion of edge $(7, 6)$ into the plane graph shown in (a). The dual $G^*$ is depicted in (b) and the extended dual $G^*_{(7,6)}$ in (c). The shortest path from vertex 7 to vertex 6, illustrated by the thick dashed edges. The resulting planarization is shown in (d).



(a) (b)

Figure 3.7: Critical configuration for the insertion of a directed edge.

vertex with two outgoing edges in $f$, called the *source* of $f$, or shorter $s(f)$. Analogously $f$ has exactly one vertex with two incoming edges in $f$, called the *sink* of $f$, or shorter $t(f)$. All remaining vertices which are adjacent to edges in $f$ have one incoming and one outgoing edge. Since each edge is represented by a monotonically increasing curve, the face on the left side, resp. right side of $e$ is the same in all upward drawings of the embedding. We denote with $lf(e)$, resp. $rf(e)$, the face on the left, resp., right side of $e$. We consider the outer face as two faces, the *left-outer face*, resp. the *right-outer face*, which denote the left, resp. right part, of the outer face.

As shown above, we have to avoid cycles when we insert edges. We avoid this by *layering* the graph. We define a valid layering $l$ as a mapping of $V$ to integers such that $l(v) > l(u)$ for each edge $(u, v) \in E \cup R$. Computing a layering for a directed acyclic graph is the first phase of the hierarchical graph drawing approach and has been studied extensively, for an overview see for example [4]. We use for our algorithm the longest path layering method, which calculates a layering of minimal height in time $O(n + m)$. Figure 3.8(a) shows a valid layering of the example in Figure 3.7, in which the y-coordinates of the vertices denote the layer numbers.

From the layered graph we construct the routing graph $R_{(a,b)}$ for the insertion of a directed edge $(a, b)$. The routing graph contains, for each face $f$ and for each layer $i$ that $f$ spans, a vertex $v(f, i)$. Two vertices lying in neighboring layers and representing the same face are connected by a directed edge of weight 0 in increasing layer order. Additionally, two vertices at the same layer $i$ of adjacent faces are connected by an edge if the source vertex of an edge separating these two faces is less than or equal to $i$

and the layer of the target vertex is greater than $i$. We assign to this edge the product of the weight of the edge separating the faces and the weight of edge $(a, b)$.

In this graph, there are no edges in decreasing layer order, in other words for each edge $(v(f, i), v(g, j))$ holds $i \leq j$. Each edge of positive weight represents one crossing. A shortest path in the routing graph represents, therefore, an insertion of an edge with minimal weighted crossing number with respect to the given layering.

We add a vertex $v(a)$ to the routing graph and connect it to faces which are adjacent to outgoing edges of $a$. Analogously we add a vertex $v(b)$ to the routing graph and connect it to faces which are adjacent to incoming edges of $b$. A path from $v(a)$ to $v(b)$ corresponds to a valid routing of the edge $(a, b)$. Figure 3.8(b) shows an example for a routing graph.

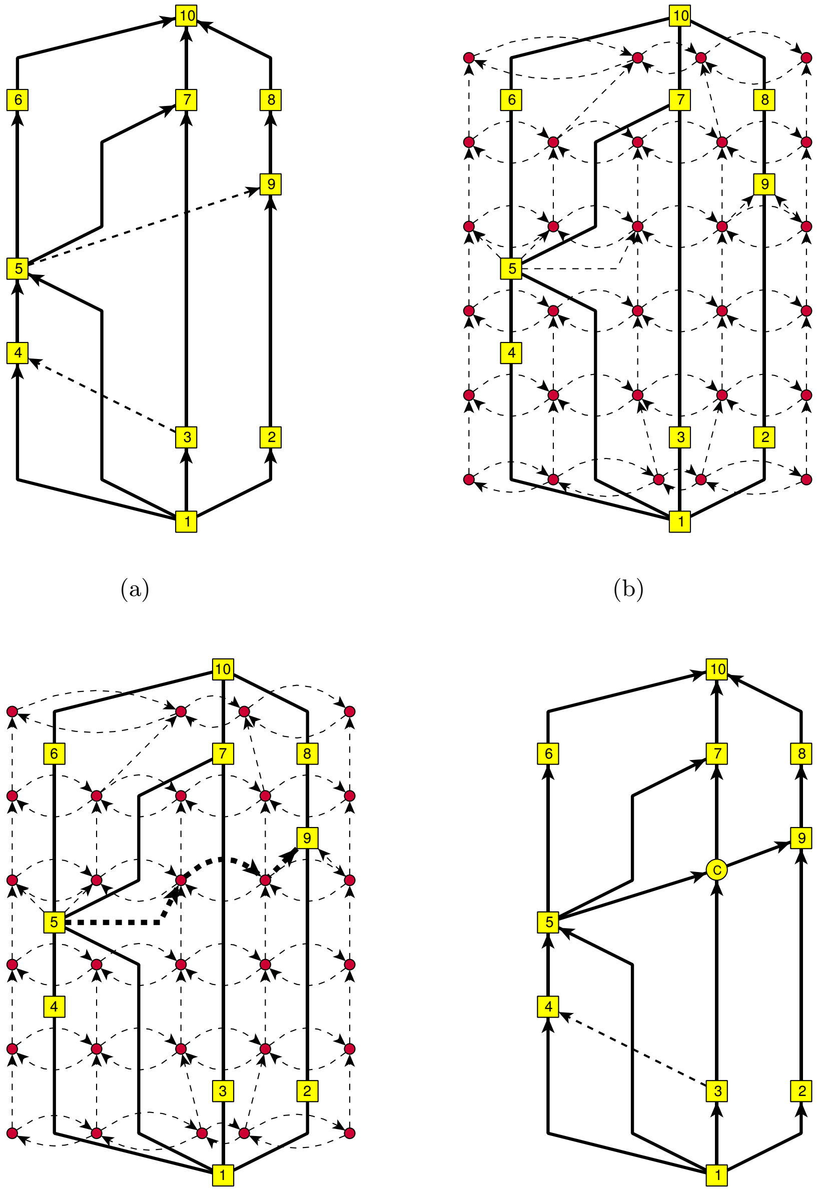
Algorithm *directed edge insertion* (`DEI`) summarizes the construction. It takes as input an upward embedded graph $G$, the set of remaining edges $R$ and one edge $e \in R$. The output $G'$ is a planarization of $(V, E \cup \{e\})$. It uses the subroutine `subdivide`$(G, e)$ which splits an edge $e = (a, b)$ into two edges $(a, w), (w, b)$, adds the vertex $w$ to $G$ and returns the created vertex $w$.

**Lemma 3.3** *The resulting graph $G' = (V', E')$ of algorithm `DEI` is an st-graph and upward planar.*

**Proof:** First we show that $G'$ is an *st*-graph. In the edge insertion step we do not decrease the in-degree or the out-degree of any vertex existing already in the input graph. Therefore, we only have to show that none of the inserted vertices is a sink or a source. But this is true, since each of these vertices has in-degree two and out-degree two. It remains to show that $G'$ is upward planar. Since the graph $G'$ is planar by construction, it remains to show that it is upward planar. We can define the $y$-coordinates of the vertices of $G'$ by assigning each vertex $v \in V \subseteq V'$ the value $2l(v)$, and each dummy vertex $w$ introduced by the subdivision of an edge $(w, u)$ the value $2l(w) + 1$. In the resulting drawing, each edge is pointing upward. □

**Lemma 3.4** *Algorithm `DEI` has time complexity $O(|V|^2)$,*

**Proof:** The faces of the graph can be computed in linear time from the embedding. A valid layering with a minimal number of layers can also be computed in linear time using a longest path algorithm. The maximum number of layers is linear, since a topological sorting is an upper bound for the number of layers. Hence, the number of vertices in the routing graph is $O(|V|^2)$ and, since each vertex has constant degree, the total size of the routing graph is $O(|V|^2)$. Because the graph is planar, we can compute the shortest path in linear time [86]. The insertion of the edge can also be done in linear time. □



(a) (b)

(c) (d)

Figure 3.8: Example for algorithm directed edge insertion. Figure (a) shows a valid layering of the input graph. The routing graph $R_{(5,9)}$ for the insertion of edge $(5,9)$ is depicted in Figure (b). The shortest path is highlighted by the thick edges in Figure (c). The resulting planarization is shown in Figure (d).

```
Algorithm 4: DEI
Input: Embedded upward planar $st$-graph $G = (V, E, F)$,
       $R \subseteq V \times V$, $e = (a, b) \in R$, $w : E \cup R \rightarrow \mathbb{N}$
Output: Embedded upward planarized $st$-graph $G'$ of $(V, E \cup \{e\})$
determine valid layering $l$ of $(V, E \cup R)$
Let $R$ be an empty directed graph
for every face $f$ in $F$ do
    create vertices $v(f, i)$ for $l(s(f)) \leq i < l(t(f))$ in $R$
    for $l(s(f)) < i < l(t(f))$ do
        create edge of weight 0 from $v(f, i-1)$ to $v(f, i)$ in $R$
for every edge $e' = (c, d)$ of $E$ do
    for $l(c) \leq i < l(d)$ do
        create edge of weight $w(e) \cdot w(e')$ from $v(rf(e'), i)$ to $v(lf(e'), i)$ in $R$
        create edge of weight $w(e) \cdot w(e')$ from $v(lf(e'), i)$ to $v(rf(e'), i)$ in $R$
create vertex $v(a)$ and $v(b)$ in $R$ representing $a$ resp. $b$
Insert edge of weight 0 from $v(a)$ to $v(f, l(a))$ in $R$ if $f$ is adjacent to $a$ and such a vertex exists
Insert edge of weight 0 from $v(b)$ to $v(f, l(b)-1)$ in $R$ if $f$ is adjacent to $b$ and such a vertex exists
Calculate shortest path $p$ from $v(a)$ to $v(b)$ in $R$
$E' = E$, $V' = V$, $G' = (V', E')$
Let $e_0, \ldots, e_n$ be the edges of weight $> 0$ in $p$
for $0 \leq i \leq n$ do
    $w_i$ =subdivide$(G', e_i)$
Add an edge between $a$ and $w_0$, $w_n$ and $b$, and $w_i$ and $w_{i+1}$ in $E'$
return $G'$
```

The following theorem summarizes the lemmas above.

**Theorem 3.3** *Let $G = (V, E)$ be an upward embedded st-graph, $R \subseteq V \times V$, $e \in R$, and the graph $(V, E \cup R)$ acyclic. Algorithm* `DEI` *computes an upward planarization $G' = (V', E')$ of $G = (V, E \cup \{e\})$ in time $O(|V|^2)$ so that the graph $(V', E' \cup \{R \setminus \{e\}\})$ is acyclic. The planarization $G'$ is again an st-graph.*

## 3.4 Rerouting

In this section, we review a local optimization method for reducing the number of crossings in a planarization. The method works for planarizations and upward planarizations.

One step of the method removes a path $p$ representing an edge from the planarization, and tries to route the edge with fewer crossings. Testing whether the edge can be routed with fewer crossings reduces again to a shortest path problem in the corresponding routing graph. We use the same routing graphs as for edge insertion, but instead of removing the edges in $p$ physically, we just give them weight zero. The advantage of this approach is that we do not have to change the planarization and in the case of upward planarizations the input graph remains an $st$-graph. If we succeed to find a better routing, we change the planarization according to the new routing, otherwise, we do not change the planarization.

We iterate this local optimization until we either do not make any further improvements, in other words there is no edge for which we can find a routing with less crossings. This is realized by defining a set of edges *Cand* which contains all edges of the original graph which have crossings in the planarization. In each iteration we choose an edge in *Cand* with maximum number of crossings and perform the local optimization step for the path defined by this edge. If the planarization has been improved we recalculate *Cand* and start again. We stop when *Cand* is empty. Since the local optimization is time consuming, we bound the total number of local optimizations steps by a constant.

## 3.5 Complete Algorithm

In this section we put the results of the previous sections together and present the complete algorithm for mixed upward planarization.

The first step of the algorithm consists of executing `MGT` to compute a maximum mixed upward planar subgraph $G'$ of the input graph. Next we want to execute algorithm `DEI` to insert the directed edges into the planarization, but this may not be possible since this algorithm assumes that the input graph is an $st$-graph. In general this condition is not fulfilled by $G'$, and we need to perform two additional steps after executing `MGT` to ensure this property. First we direct the undirected edges in $G'$ temporarily, so that the resulting planarization is upward planar. This is achieved by directing these edges according to the ordering $\Pi$ computed by algorithm `MVO`. Then we augment the graph to an upward planar $st$-graph. Again we use the ordering $\Pi$ for this task. Assume there is a sink $v$ in the graph with $\Pi(v) = k < n$. Then we insert a directed edge $(v, w)$ with $\Pi(w) = k + 1$ into $G'$. Analogously we insert a directed edge $(w, v)$ with $\Pi(w) = k - 1$ into $G'$ if $v$ is a source with $\Pi(v) > 1$. Clearly $G'$ is an $st$-graph after this step. We need to perform the $st$-completion only once before the edge insertion process since Lemma 3.3 guarantees that the graph remains an $st$-graph after inserting an edge. Note that this augmentation does not affect the worst-case running time of the algorithm, since it follows from Euler's theorem that the number of edges in the graph remains linear in the number of vertices.

After the augmentation we are ready to invoke algorithm `DEI` for each directed edge removed from $G$ in `MGT`. Edges in the routing graph representing an edge added in the augmenting step are assigned weight 0, because they do not introduce a real crossing. The removed directed edges are inserted in random order. After the routing the artifacts, which were introduced to make the input graph an *st*-graph, are removed and the temporarily directed edges are marked as undirected.

As a last step we insert the undirected edges into the planarization. We insert them one-by-one with algorithm `UIE`. After all undirected edges have been inserted, we perform rerouting on the undirected edges to improve the quality of the planarization.

Algorithm *mixed upward planarization* (`MUP`) summarizes the above description of the planarization algorithm for mixed graphs.

**Theorem 3.4** *Let* $G = (V, E_d \cup E_u)$ *be a mixed graph. Algorithm* `MUP` *creates an embedded mixed upward planarized graph of* $G$ *in time* $O(|V|(|E_d| + |E_u|) + (|V| + |C|)^2|E_d \setminus E'_d| + (|V| + |C|)|E_u \setminus E'_u|)$*, where* $C$ *is the set of crossings in the planarized graph.*

The running time of algorithm `MGT` depends therefore heavily on the input graph, especially on the size of the mixed-upward-planar subgraph found by algorithm `MGT` and the number of crossings in the planarization.

For the application of automatic layout of class diagrams we will see in Chapter 7 that we can safely assume that $|C|$ is $O(|V|)$ and the input graph is sparse. The above running-time formula evaluates then to $O(|V|^2 + (|V|)^2|E_d \setminus E'_d| + (|V|)|E_u \setminus E'_u|)$, which is $O(|V|^3)$.

In our tests on automatically generated class diagrams in Chapter 7 we experienced small values for $|E_d \setminus E'_d|$, too. The value of $|E_d \setminus E'_d|$ was never greater than 16 for class diagrams with up to 80 classes. Therefore one can expect that on real world data for class diagrams algorithm `MGT` performs with quadratic running time.

Algorithm 5: `MUP`

```
Input: Mixed graph $G = (V, E_d \cup E_u)$, weight $w : E_d \cup E_u \to \mathbb{N}$
Output: Mixed upward planarization $\Gamma'$ of $G$.
MGT($G, w, G' = (V, E'_d \cup E'_u), \Pi$)
$I_d = E_d \setminus E'_d$
$I_u = E_u \setminus E'_u$
// Direct undirected edges
for $\{v, w\} \in E'_u$ do
    if $\Pi(v) < \Pi(w)$ then $E'_d = E'_d \cup \{(v, w)\}$
    if $\Pi(v) > \Pi(w)$ then $E'_d = E'_d \cup \{(w, v)\}$
    $E'_u = E'_u \setminus \{v, w\}$
// Augment $G'$ to an st-graph.
$T = \emptyset$
for $i = 0, \dots, n$ do
    $v = \Pi^{-1}(i)$
    if $(\delta^+_{G'}(v) = 0)$ and $i < n$ then
        $T = T \cup \{(v, \Pi^{-1}(i + 1))\}$
        $E'_d = E'_d \cup \{(v, \Pi^{-1}(i + 1))\}$
    if $\delta^-_{G'}(v) = 0$ and $i > 0$ then
        $T = T \cup \{(w, \Pi^{-1}(i - 1)\}$
        $E'_d = E'_d \cup \{(w, \Pi^{-1}(i - 1))\}$
// Insert directed edges
Compute embedding of $G'$ from $\Pi$
for $e \in I_d$ do
    $G' =$ DEI($G', I_d, e, w$)
    $I_d = I_d \setminus \{e\}$
Reroute directed edges
// Remove artefact
for $e \in T$ do
    remove representation of $e$ from $G'$
// Undirect temporarily directed edges
for $e \in E_u \setminus I_u$ do
    $p(e)$ is the representation of $e$ in $G'$
    $E'_d = E'_d \setminus p(e)$
    $E'_u = E'_u \cup p(e)$
// Insert undirected edges
for $e \in I_u$ do
    $G' =$ UEI($G', e$)
Reroute undirected edges
```

# Chapter 4

# Orthogonalization

In this chapter we describe an orthogonalization algorithm which tries to minimize the number of bends while not violating the special requirements of UML diagrams, notably that vertices have prescribed size, directed edges point upward and some marked subgraphs are layouted as hyperedge. The result of the algorithm is an absolute `Kandinsky` shape which is computed from a mixed upward planarization.

Our orthogonalization algorithm is based on the `Kandinsky` algorithm, introduced by Kaufmann and Fößmeier. The `Kandinsky` algorithm [54, 56] is an extension of Tamassia's algorithm, which computes bend-minimal point drawings of plane 4-graphs [115]. Tamassia's algorithm is reviewed in Section 4.1.

The `Kandinsky` algorithm removes the severe limitation of Tamassia's algorithm that it is restricted to point drawings and therefore limited to 4-graphs. This restriction is prohibitive for use in many real world applications, especially for UML class diagrams. UML class diagrams are orthogonal rectangle drawings and vertices may have arbitrary degree. The `Kandinsky` algorithm is chosen since other orthogonalization algorithms which produce rectangle drawings cannot guarantee prescribed vertex sizes. The latter algorithms use Tamassia's algorithm as subroutine to produce orthogonal rectangle drawings of graphs with non-bounded degree. We review these algorithms in Section 4.2.

The `Kandinsky` approach is discussed in Section 4.3. We first review the different `Kandinsky` models and then define the KANDINSKY BEND MINIMIZATION problem. To model this problem we introduce *arc partition minimum cost flow networks* as a natural generalization of minimum cost flow networks. The core of the `Kandinsky` approach is a generalization of the Tamassia's minimum cost network flow formulation for 4-graphs to graphs of arbitrary degree using arc partition minimum cost flow networks. We call the generalized network the ***Kandinsky*** *network*.

We will show in Section 4.4 that solving arc partition minimum cost flow networks to optimality is NP-hard in general even in very restrictive settings. Fößmeier and Kaufmann proposed an optimal algorithm for the KANDINSKY BEND MINIMIZATION problem which exploits the special structure of the `Kandinsky` network to compute an optimal solution. We will show that this algorithm has a flaw and that for some special input instances the algorithm does not yield an optimal solution. On instances with vertices of high degree it may not even terminate correctly. We show how we can modify the algorithm such that it terminates always with a valid solution. Unfortunately the modified algorithm may return sub-optimal solutions and it is unclear if there is an optimal polynomial time algorithm. However, we present an approximation algorithm for the problem, with approximation factor two and a heuristic based on this approximation algorithm which yields nearly optimal solutions in practice.

As already mentioned we have to consider the upwardness property of directed edges and the hyperedge layout of marked subgraphs if we want to use the `Kandinsky` algorithm for the automatic layout of class diagrams. We will use a two phases approach to solve this problem. In Section 4.5 we present the CONSTRAINED KANDINSKY BEND MINIMIZATION problem which is a generalization of the KANDINSKY BEND MINIMIZATION problem. The CONSTRAINED KANDINSKY BEND MINIMIZATION problem takes besides a planarization a set of low-level constraints as input. These low level constraints can define target values for the number and types of bends at edges and for the angles formed by intervals of neighboring edges around a vertex in the embedding. We will present an extended version of the `Kandinsky` network in which we can specify low level constraints for the resulting shape. Until now it was only possible to specify such constraints using integer linear programming.

In Section 4.6 we will use these low level constraints to define our application specific constraints, the upwardness property of directed edges and the hyperedge layout of marked subgraphs, which leads to an orthogonalization algorithm for UML class diagrams. This two level approach has the advantage that it makes our algorithm reusable in the sense that we can model other constraints as the one specified by class diagrams with low level constraints. We will for example use this approach for dynamic graph drawing in Chapter 6.

## 4.1 Tamassia's Algorithm

In this section we will review Tamassia's algorithm [115], which computes a bend-minimal orthogonal point drawing of a plane 4-graph with respect to an input embedding. The algorithm does not change the input embedding. If we allow that the embedding can be changed the problem turns out to be NP-hard [65]. In the remainder of this section we assume that graphs have maximal degree 4.

Given a bend $b$ on an edge $\{v, w\}$ in an orthogonal drawing $\Gamma$ of a plane graph $G$, and let $e = (v, w)$ be a dart of the edge $\{v, w\}$. The bend $b$ is adjacent to two line segments. We denote with $l_1$ the segment which is first traversed when we follow the path $\Gamma(\{v, w\})$ from $v$ to $w$, and with $l_2$ the second segment. The bend $b$ is called *convex* with respect to $e$ if the angle between $l_1$ and $l_2$ is 90°, and it is called *concave* with respect to $e$ if the angle is 270°.

**Definition 4.1** *Let $G = (V, E, F)$ be a plane 4-graph. An* orthogonal shape *$H$ is a mapping from the set of faces in $F$ to clockwise ordered lists of tuples $(e, b, a)$. The first entry in the tuple corresponds to the dart in the face. The second entry is a bit string denoting the bends of the dart. A 0 in the bit string denotes a convex bend, a 1 a concave bend. The third entry is an integer denoting the angle formed with the preceding dart in the face, stored as multiple of 90°. Note that in a planar orthogonal drawing $1 \leq a \leq 4$ holds.*

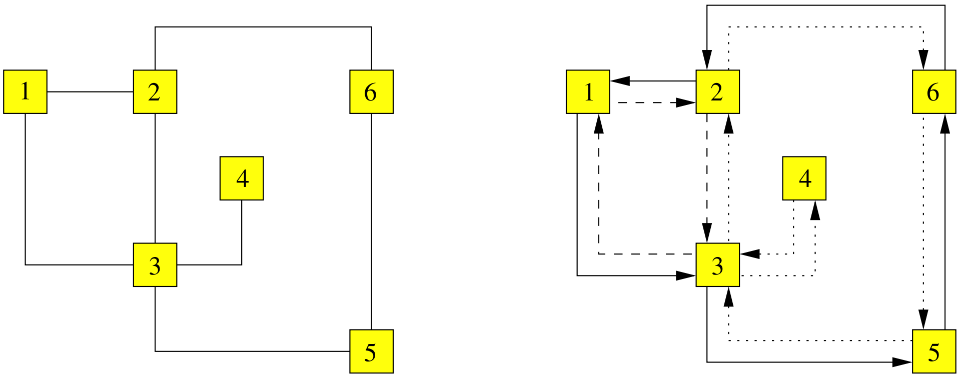


Figure 4.1: Example for an orthogonal shape of a planar graph. The orthogonal drawing of the graph is shown left. On the right the orthogonal shape of this drawing is sketched. Darts with the same line style belong to the same face.

We will illustrate the definition of orthogonal shape on the example shown in Figure 4.1. The plane graph is defined by the planar representation $F = (f_0, f_1, f_2)$ where the face $f_0$ is denoted by the solid darts, the face $f_1$ by the dashed darts and the face $f_2$ by the pointed darts. The orthogonal shape of the drawing in Figure 4.1 is defined as follows:

$$\begin{aligned}
H(f_0) &= \{((1,3),1,2),((3,5),1,0),((5,6),\epsilon,2),((6,2),11,1),((2,1),\epsilon,0)\},\\
H(f_1) &= \{((1,2),\epsilon,0),((2,3),\epsilon,0),((3,1),0,0)\},\\
H(f_2) &= \{((5,3),0,0),((3,4),1,0),((4,3),0,3),((3,2),\epsilon,0),((2,6),00,1),\\
&\qquad ((6,5),\epsilon,1)\}\ .
\end{aligned}$$

A planar orthogonal point drawing $\Gamma$ of a graph $G$ defines a unique orthogonal shape, which we denote with $H(\Gamma)$. However, there does not exist an orthogonal point drawing for every orthogonal shape. An orthogonal shape $H$ of a plane graph $G$ for which a planar orthogonal point drawing $\Gamma(G)$ with $H(\Gamma(G)) = H$ exists, is said to be *valid*. The following theorem characterizes the valid orthogonal shapes for a given plane graph:

**Theorem 4.1 ([115])** *Let $G = (V, E, F)$ be a plane 4-graph with embedding $\mathcal{E}$, and $H$ an orthogonal shape of $G$. We define the rotation $rot(f)$ of a face $f \in F$ as:*

$$rot(f) = \begin{cases} 2|f| - 4 & f \in F, f \neq f_{out} \\ 2|f| + 4 & f \in F, f = f_{out} \end{cases}$$

*$H$ is valid if, and only if, the following conditions hold:*

$$\sum_{e \in \mathcal{E}(v)} H_a(e) = 4 \ \forall v \in V \tag{4.1}$$

$$\sum_{e \in f} (H_a(e) - \#_0 H_b(e) + \#_1 H_b(e)) = rot(f) \ \forall f \in F \tag{4.2}$$

$$H_b(e) = \overleftarrow{\overline{H_b(\bar{e})}} \ \forall e \in \bar{E} \tag{4.3}$$

Computing a drawing from a valid orthogonal shape is called *compaction* and is discussed in Chapter 5.

The number of bends $\#bends(\Gamma)$ in an orthogonal drawing is defined by its orthogonal shape $H$. It holds

$$\#bends(\Gamma) = \#bends(H) := \frac{1}{2} \sum_{f \in F} \sum_{(e,a,b) \in H(f)} |b| \ .$$

**Definition 4.2** *The BEND MINIMIZATION problem for a plane 4-graph $G$ is to find a valid orthogonal shape $H$ of $G$ with a minimal number of bends.*

The BEND MINIMIZATION problem can be solved using minimum cost flow network algorithms. For each plane graph $G$ there is a minimum cost flow network $\mathcal{N}_G$ in which there is a one-to-one correspondence between valid flows in the network and valid orthogonal shapes of $G$. The cost of a network flow corresponds to the number of bends in the induced orthogonal shape, and therefore a bend-minimal orthogonal shape can be computed with a minimum cost flow network solving algorithm.

We first review minimum cost flow networks and then give a description of $\mathcal{N}_G$. For a comprehensive overview over minimum cost flow networks see, for example, [2].

**Definition 4.3 (Minimum Cost Flow Network)** *Let $\mathcal{N} = (N, A)$ be a directed graph with a cost $c_{ij}$ and a capacity $u_{ij}$ associated with every arc*

$(i, j) \in A$. *We associate with each node* $i \in N$ *a number* $b(i)$ *which indicates its supply or demand depending whether* $b(i) > 0$ *or* $b(i) < 0$. *The minimum cost flow problem can be stated as follows:*

$$\textit{Minimize } z(x) = \sum_{(i,j)\in A} c_{ij} x_{ij}$$

*subject to*

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b(i) \quad \forall i \in N,$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i, j) \in A.$$

The vector $x$ is called *flow*. A flow which satisfies all equations and inequalities is called *feasible*. A minimum cost flow network must satisfy $\sum_{i\in V} b(i) = 0$ in order to be feasible. If we require that there is at most one node with positive and at most one node with negative supply, we call the network an $st$-minimum cost flow network. In such a network the node with positive supply is called source, or shorter $s$, and the node with positive demand sink, or shorter $t$.

We can transform every minimum cost flow network to an $st$-minimum cost flow network by a standard transformation: We add two nodes $s$ and $t$ to the network and connect all nodes $i$ with $b(i) > 0$ to the source with an arc of capacity $u_{si} = b(i)$ and all nodes $i$ with $b(i) < 0$ to the target with an arc of capacity $u_{it} = -b(i)$. All newly introduced arcs have zero cost. We set $b(s) = \sum_{(s,i)\in A} u_{si}$ and $b(t) = \sum_{(i,t)\in A} u_{it}$. Clearly the set of feasible solutions of the transformed network is the same as the set of feasible solutions of the original network, and the cost of two corresponding feasible flows is the same.

The BEND MINIMIZATION flow network $\mathcal{N}_G = (N, A)$ for a plane 4-graph $G = (V, E, F)$ is a minimum cost flow network and is defined as follows: The set of nodes $N$ is defined as

$$N = N_V \cup N_F$$

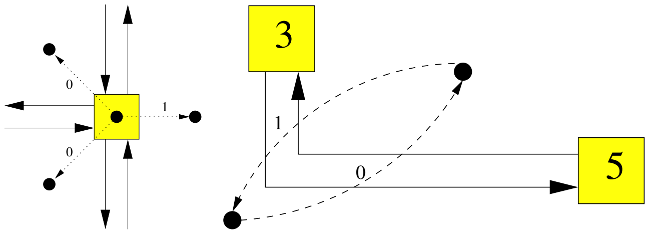
with

1. The set $N_V$ contains a node for each vertex in $V$: $N_V = \{n_v | v \in V\}$. The supply of a vertex-node is defined as $b(n_v) = 4 - \delta(v)$.
2. The set $N_F$ contains a node for each face in $F$: $N_F = \{n_f | f \in F\}$. The supply of a face-node is defined as $b(n_f) = -rot(f)$.

The set of arcs $A$ is defined as:

$$A = A_{VF} \cup A_{FF}$$

with



(a) (b)

Figure 4.2: In (a) the network arcs around vertex 2 of the plane graph from Figure 4.1 is shown and in (b) the network arcs between the faces of edge $(3, 5)$. Arc labels indicate flow values according to the depicted shape

1. The set $A_{VF}$ connects every vertex $v$ with its adjacent faces:
   $A_{VF} = \{a^V_e = (n_v, n_f) | v \in V, e \in \mathcal{E}(v), f = \mathit{face}(e)\}$.
   Arcs in $A_{VF}$ have cost zero and capacity three.
2. The set $A_{FF}$ connects two faces which share an edge:
   $A_{FF} = \{a^F_e = (n_f, n_g) | e \in \bar{E}, f = \mathit{face}(e), g = \mathit{face}(\bar{e}), f \neq g\}$.
   Arcs in $A_{FF}$ have cost one and capacity $\infty$.

The angle defined inside a face $f$ between two consecutive edges $(u, v)$, $(v, w)$ is determined by the flow on the network-edge $a^V_{(v,w)} = (n_v, n_f)$ in $A_{VF}$ that connects the vertex-node $n_v$ of $v$ with the face-node $n_f$ of $f$. A flow of 0 defines an angle of 90°, a flow of 1 an angle of 180° and so forth. A flow unit on an edge $a^F_e = (n_f, n_g) \in A_{FF}$ between two face-vertices denotes a bend on edge $e$, which is convex in $f$ and concave in $g$. See Figure 4.2 for an illustration. We can summarize the above discussion as follows: Given a plane graph $G$ and a feasible flow $x$ in the bend-minimization network $\mathcal{N}_G$ then we can define the orthogonal shape $H^x$ by:

$$
\begin{aligned}
H^x_a(e) &= x(a^V_e) + 1 \;\; \forall e \in \bar{E} \\
H^x_b(e) &= 0^{x(a^F_e)} 1^{x(a^F_{\bar{e}})} \;\; \forall e \in \bar{E}
\end{aligned}
$$

Since the arcs in $A_{FF}$ are the only ones with non-zero cost, and every unit flow on such an arc corresponds to a bend, the value of a feasible flow corresponds to the number of bends in the orthogonal shape. Therefore an optimal flow of $\mathcal{N}_G$ corresponds to a bend-minimal orthogonal shape of $G$. Figure 4.3 illustrates the network $\mathcal{N}_G$ for an example.

Garg and Tamassia [66] proposed an algorithm for solving minimum cost network flows which is optimized for the BEND MINIMIZATION problem. The algorithm has running time $O(\chi^{\frac{3}{4}} |A| \cdot \sqrt{\log |N|})$ where $\chi$ is the cost of the flow. Since for every plane 4-graph there is always an orthogonal

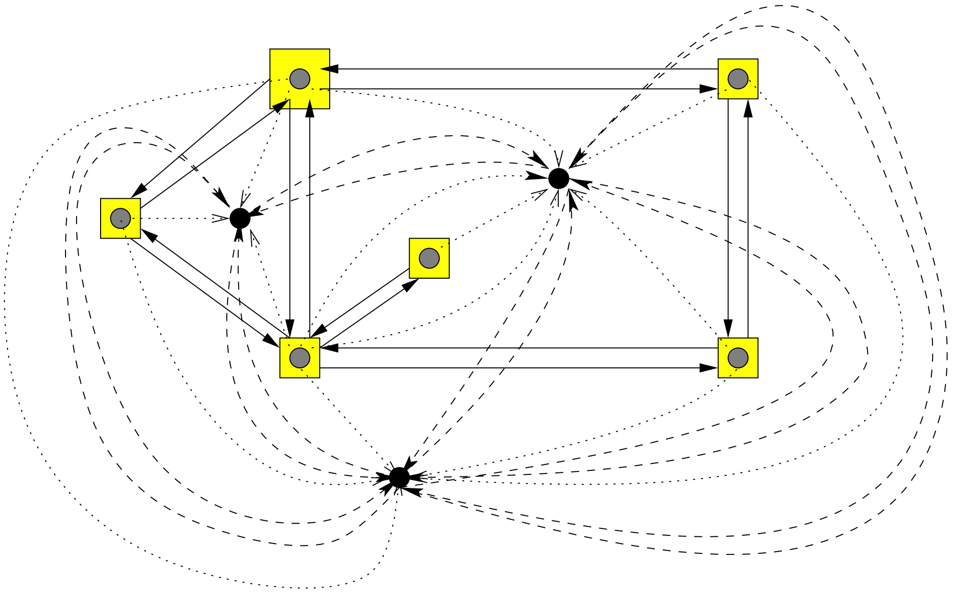


Figure 4.3: The network $\mathcal{N}_G$ for the plane graph from Figure 4.1. The nodes in $N_V$ have gray color in the drawing while the nodes in $N_F$ have black color. Arcs in $A_{VF}$ are pointed while arcs in $A_{FF}$ are dashed.

drawing with linear number of bends, it holds $\chi = O(n)$. Furthermore it follows from Euler's theorem and the definition of $\mathcal{N}_G$ that $|A| = O(|N|)$ and $|N| = O(n)$. We summarize the above discussion in the following theorem:

**Theorem 4.2 ([115, 66])** *Given a plane 4-graph $G$ and a flow $x$ in the BEND MINIMIZATION network $\mathcal{N}_G$. The orthogonal shape $H^x$ is valid if, and only if $x$ is feasible. For a feasible flow $x$ holds $z(x) = \#bends(H^x)$. The BEND MINIMIZATION problem can be solved in running time $O(n^{\frac{7}{4}} \cdot \sqrt{\log n})$.*

## 4.2 Generalizations of Tamassia's Algorithm Using Reduction

One possibility to generalize Tamassia's algorithm to orthogonal rectangle drawings of graphs with arbitrarily degree is to use the *reduction approach.* The idea behind this approach is not to change the algorithm, rather change the input. The reduction approach is based on the observation, that we can often overcome the limitations of an algorithm which requires a special type of input by transforming the input, so that it fulfills these requirements, and then perform the algorithm on the transformed input. We can then obtain a solution for the original input by interpreting the algorithms result. In an object-oriented sense this is captured by the *Algorithmic Reduction* design pattern proposed by Gelfand and Tamassia [67]. In our setting the algorithm

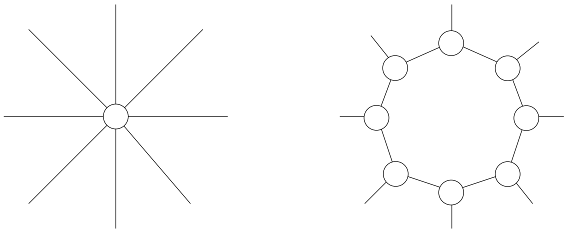


Figure 4.4: Transformation of a high-degree vertex into a face.

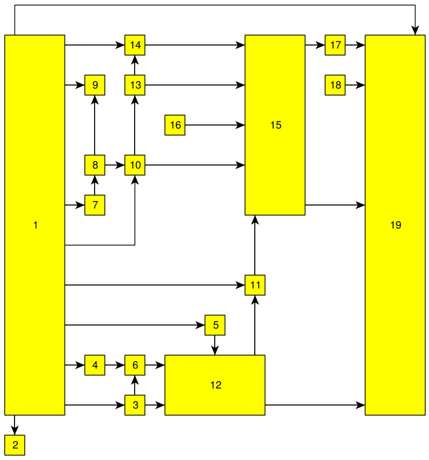


Figure 4.5: A drawing generated with Klau's approach.

is Tamassia's bend minimizing algorithm and the special requirement is that the input graph must have maximum degree 4.

The first algorithm using reduction to generalize Tamassia's algorithm is *GIOTTO* [5]. It creates a rectangle-face for every vertex. Each edge is assigned to a side of the rectangle by a heuristic. Then the bend minimization algorithm is applied to this graph, with a modification which assures that the rectangle-faces have rectangular shape.

The algorithm of Klau described in [83] creates also rectangle vertices, but does not, unlike GIOTTO, assign the edges to vertex sides before the orthogonalization step. The rectangle-faces created by the algorithm are rings, and by modifying the bend-minimization algorithm it is assured that these rings have rectangular shape.

In both cases the reduction is applied before the orthogonalization phase and is reversed not until the compaction phase has finished. Both approaches have the huge disadvantage that vertices may be arbitrary big in the final drawing. Therefore these algorithms are not suitable for diagrams which require prescribed vertex sizes like UML class diagrams.

## 4.3 `Kandinsky`

The `Kandinsky` algorithm was introduced by Fößmeier and Kaufmann [54, 56]. It is an extension of Tamassia's algorithm and removes two severe limitations of Tamassia's algorithm: the restriction to point drawings and the restriction to graphs with degree less than five.

The `Kandinsky` algorithm is motivated by the geometric properties of the *original* *`Kandinsky`* *model* for orthogonal drawings of graphs. From the original `Kandinsky` model a series of other `Kandinsky` models for orthogonal drawings of graphs is derived. We will present these `Kandinsky` models and the KANDINSKY BEND MINIMIZATION problem in Section 4.3.1. We discuss a network flow formulation of the KANDINSKY BEND MINIMIZATION problem in Section 4.3.1.

### 4.3.1 The `Kandinsky` Model

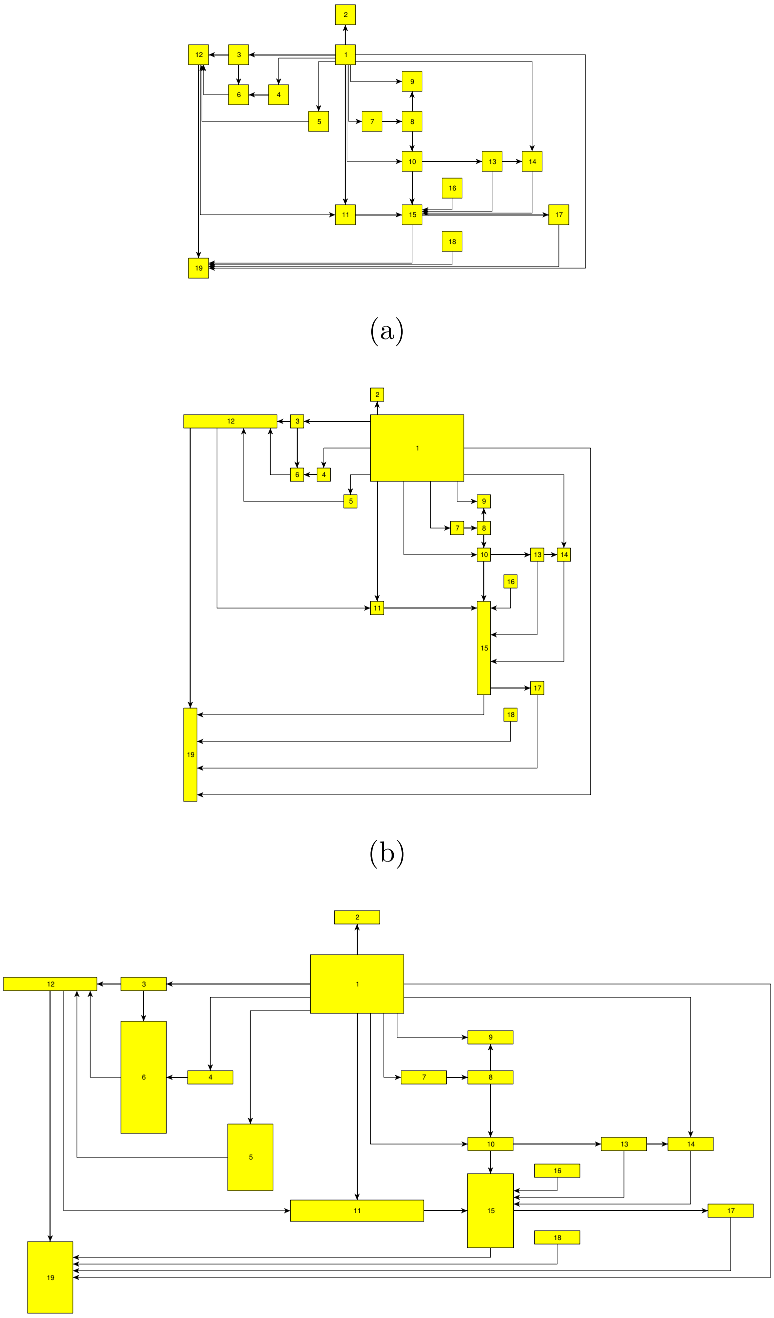
In the original `Kandinsky` model all vertices are represented by squares of equal size, arranged on a coarse vertex grid [56]. Edges are routed on a finer edge grid. Later, the *big-node model* [57] was proposed, in which the size of a vertex is determined by the number of edges attached to the different sides of the vertex. In the big-node model there is only one grid for vertices and edges. In [32] the *podavsnef*-model is introduced in which vertices have prescribed size. This model has, like the big-node model, only one grid. We refer to this model as the *prescribed-size* `Kandinsky` model in this work. When we use the term `Kandinsky` model, we refer to the original `Kandinsky` model.

All `Kandinsky` models define special types of orthogonal rectangle drawings. Although the vertex sizes in the drawings differ, all these models share common properties, which are motivated by the geometric properties of the original `Kandinsky` model.

Before we investigate these properties we first generalize the concept of orthogonal shape to describe orthogonal rectangle drawings. Since in rectangle drawings the angle between two consecutive edges at a vertex is no longer well defined in a traditional sense, we have to introduce a new angle definition which fits our needs.

**Definition 4.4** *Let $r$ be a rectangle and $l_1$ and $l_2$ be line segments which are either horizontal or vertical. Both, $l_1$ and $l_2$, have one endpoint on the boundary of $r$, which must not be a corner of $r$. The rectangle-angle between $l_1$ and $l_2$ is defined as the number of corners between $l_2$ and $l_1$ in counter-clockwise order.*

This definition is a natural extension of the angle definition in point drawings in the sense that if we take a point drawing and replace the points by rectangles of infinitesimal size $\epsilon$ and clip the edges adjacent to the point



(a)

(b)

(c)

Figure 4.6: Example for a drawing in the original `Kandinsky` model (a), in the `Kandinsky` big-node model (b), and in the prescribed-size `Kandinsky` model (c).

against this rectangle, then the rectangle-angle corresponds to the original angle in the point drawing. With this new angle definition we can now define the generalization of orthogonal shapes to orthogonal rectangle drawings:

**Definition 4.5** *Let $G = (V, E, F)$ be a plane graph. A* quasi-orthogonal shape *$Q$ is a mapping from the set of faces $F$ to clockwise ordered lists of tuples $(e, b, a)$. The first entry in the tuple corresponds to the dart in the face. The second entry is a bit string denoting the bends of the dart. A 0 in the bit string denotes a convex bend, a 1 a concave bend. The third entry is an integer and denotes the rectangle-angle formed with the preceding dart in the face, stored as multiple of 90°. Note that in a planar orthogonal rectangle drawing $0 \leq a \leq 4$ holds.*

Analogously to orthogonal shapes, we define a valid quasi-orthogonal shape as a quasi-orthogonal shape for which an orthogonal rectangle drawing exists. Surprisingly theorem 4.1 holds also for quasi-orthogonal shapes [54]. However, the bend minimization problem turns out to be fairly simple: for every plane graph there is a quasi-orthogonal shape with no bends. This follows immediately from the fact, that every plane graph has a one-dimensional visibility representation [108, 116]. However, like in the previous section, the vertex size in the drawing can grow arbitrarily.

This is where the `Kandinsky` model enters the stage. In the original `Kandinsky` model, vertices are represented by squares of equal size. Although for every valid quasi-orthogonal shape exists an orthogonal rectangle drawing, not for every valid quasi-orthogonal shape exists an orthogonal rectangle drawing in the `Kandinsky` model !

But we can define two sufficient properties for a quasi-orthogonal shape which guarantee that a drawing in the `Kandinsky` model exist. These two properties are the *bend-or-end property* and the *non-empty face property.*

**Definition 4.6 (Bend-Or-End Property)** *Let $G{=}(V, E, F)$ be a plane graph. A quasi-orthogonal shape $Q$ of $G$ has the bend-or-end property if for every two darts $(u, v)$ and $(v, w)$ following each other in a face $f \in F$ either the last bend of the dart $(u, v)$ or the first bend of the dart $(v, w)$ is concave.*

Bends which exists because of the bend-or-end property are called *vertex-bends*, all other bends are called *face-bends.*

**Definition 4.7 (Non-Empty Face Property)** *Let $G{=}(V, E, F)$ be a plane graph and $Q$ a quasi-orthogonal shape of $G$. Let $f$ be a face with three edges $f = ((u, v), (v, w), (w, u))$. The face $f$ is called:*

**$L$-triangle** *if* $Q(f) = \{((u, v), 1, 0), ((v, w), \epsilon, 0), ((w, u), \epsilon, 1)\}$.

**$T$-triangle** *if* $Q(f) = \{((u, v), 1, 0), ((v, w), 1, 0), ((w, u), \epsilon, 0)\}$,

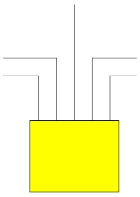
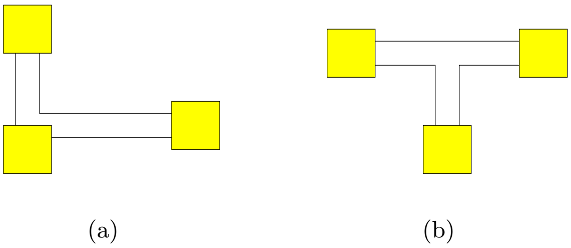


Figure 4.7: A vertex with the bend or end property.

*A quasi-orthogonal shape $Q$ has the non-empty face property if it contains no $T$- and $L$-triangles.*



(a) (b)

Figure 4.8: An example for an $L$-triangle (a) and a $T$-triangle (b).

We are now able to define `Kandinsky` shapes:

**Definition 4.8** *A valid quasi-orthogonal shape is a* `Kandinsky` *shape if it has the bend-or-end property and the non-empty face property.*

While the bend-or-end property is not only sufficient, but also necessary, for a quasi-orthogonal shape to have a drawing in the `Kandinsky` model the non-empty face property is kind of artificial. That the $L$ triangle is excluded can be motivated from the fact that it cannot be represented in the `Kandinsky` model without overlap (which is still possible in the `Kandinsky` big-node model). But the exclusion of the $T$-triangle is of pure technical nature, and is not necessary for the existence of a drawing in the `Kandinsky` model. A motivation to exclude them is that they are the only type of faces which cannot be drawn in the `Kandinsky` model with positive area (this explains the name of the property). In [54] it is argued that faces which cover no area are not desirable in a drawing because, for example, they cannot be labeled properly. A more technical motivation to postulate this property is that it yields an alternative characterization of `Kandinsky` shapes:

**Theorem 4.3 ([56])** *A quasi-orthogonal shape $Q$ has the non-empty face property if, and only if, every $0°$ angle has a unique corresponding vertex-*

*bend. For a* `Kandinsky` *shape always exists a drawing in the original* `Kandinsky` *model.*

**Definition 4.9** *Let $G = (V, E, F)$ be a plane graph. The KANDINSKY BEND MINIMIZATION problem is to find a* `Kandinsky` *shape $Q$ of $G$ which minimizes $\#bends(Q)$.*

### 4.3.2 The Network Flow Formulation

In this section we will present a network flow formulation for the KANDINSKY BEND MINIMIZATION problem. An alternative way to solve the KANDINSKY BEND MINIMIZATION problem is to use integer linear programming as described in [46].

We will base our network flow formulation on Tamassia's algorithm. Because zero is allowed as angle value in quasi-orthogonal shapes we have to change the minimum cost flow network construction. Since zero flow represents an angle of 90°, an angle of 0° corresponds to negative flow which is not possible in the original network. In the `Kandinsky` network, this problem is solved by introducing additional nodes and arcs which allow flow coming from a face-node to enter a vertex-node. Before a unit of flow can enter a vertex-node, the flow must cross an edge and thus create a bend. The arc which represents this bend is called the *vertex-bend arc*. To avoid negative angles only one vertex-bend arc adjacent to an angle is allowed to carry flow. This is achieved by introducing a helper node to which all vertex-bend arcs of an angle connect and an arc of capacity one which connects this helper node with the vertex-node. It follows that each 0° angle has the associated bend required by Theorem 4.3. Note that vertex-nodes may have negative supply in this network since for a vertex $v$ with degree greater than four $b(n_v) = \delta(v) - 4$ is negative. The above changes to the network are illustrated in Figure 4.9.

However, the network defined this way contains feasible flows which do not correspond to a valid `Kandinsky` shape. For an edge $(v, w)$ there are two vertex bend edges entering $v$, one for a 90° vertex bend and one for a 270° vertex bend. A flow saturating both vertex bend edges defines two vertex bends at an endpoint and does therefore not induce a valid `Kandinsky` shape since an edge can only have *one* vertex bend at an end node. We cannot model this requirement directly in the minimum cost flow network, we have to use a generalization of minimum cost flow networks.

**Definition 4.10 (Arc Partition Minimum Cost Flow Network)** *Let $\mathcal{N} = (N, A)$ be a directed graph with arc cots $c_{ij}$ and with a partition $D = \{d_i | 0 \leq i \leq k, d_i \subseteq A\}$ of $A$. The elements of $D$ are called* devices. *A capacity $u_d$ is associated with every device $d \in D$. We associate with each node $i \in N$ a number $b(i)$ which indicates its supply or demand depending*

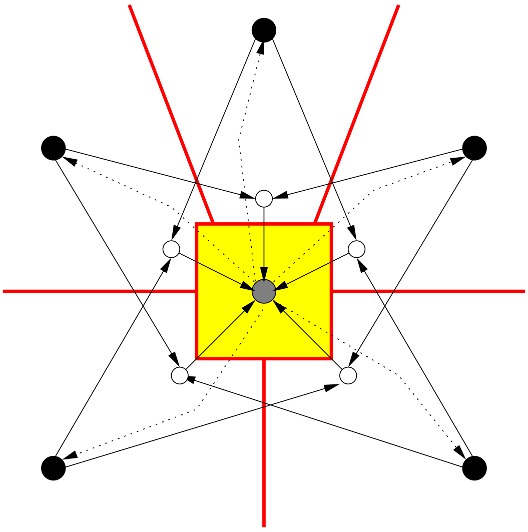


Figure 4.9: `Kandinsky` flow network for a vertex adjacent to five edges. The red lines depict edges of the input graph, black lines depict network arcs. All arcs have capacities one except pointed arcs which have capacity three. Arcs adjacent to face nodes have cost 1, all others cost zero. The white nodes are the introduced helper nodes.

*whether $b(i) > 0$ or $b(i) < 0$. The arc partition minimum cost flow problem can be stated as follows:*

$$\textit{Minimize } z(x) = \sum_{(i,j)\in A} c_{ij}x_{ij}$$

*subject to*

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b(i) \quad \forall i \in N,$$

$$0 \leq x_{ij} \quad \forall (i,j) \in A.$$

$$\sum_{(i,j)\in d} x_{ij} \leq u_d \quad \forall d \in D.$$

A device $d \in D$ with $|d| = 1$ is called a *trivial device*. Arc partition minimum cost flow networks are a generalization of minimum cost flow networks, since an arc partition minimum cost flow network with only trivial devices is a minimum cost flow network.

In `Kandinsky` minimum cost flow networks we use the devices to avoid that both vertex-bend arcs of an edge at a vertex carry flow. We put them together in one device and set the device capacity to one. We call these devices ***`Kandinsky`*** *devices*. Note that they are arranged in a cyclic way around each vertex $v \in V$. In Figure 4.9 the arcs represented by solid lines crossing each other are a `Kandinsky` device.

We will give now a formal description of the KANDINSKY BEND MINIMIZATION network $\mathcal{N}_G^K$ of a plane graph $G$, whose feasible solutions correspond to valid `Kandinsky` shapes. First we need the following definition to shorten the notation: Let $e = (v, w) \in \mathcal{E}(v)$ for a vertex $v \in V$, then

$$h(e, 0) = e \ \text{ and } h(e, 1) = succ_{\mathcal{E}(v)}(e)$$

Let $G = (V, E, F)$ be a plane graph. The set of nodes $N$ of the network $\mathcal{N}_G^K$ is defined as

$$N = N_V \cup N_F \cup N_H$$

with

1. The set $N_V$ contains a node for each vertex in $V$: $N_V = \{n_v | v \in V\}$. The supply of a vertex-node $n_v$ is defined as $b(n_v) = 4 - \delta(v)$.
2. The set $N_F$ contains a node for each face in $F$: $N_F = \{n_f | f \in F\}$. The supply of a face-node $n_f$ is defined as $b(n_f) = -rot(f)$.
3. The set $N_H$ contains a node for each dart in $\bar{E}$: $N_H = \{n_e | e \in \bar{E}\}$. The supply of a helper-node $n_e$ is defined as $b(n_e) = 0$.

The set of arcs $A$ of $N_G^K$ is defined as

$$A = A_{VF} \cup A_{FF} \cup A_{FE} \cup A_{EH} \cup A_{HV} \ .$$

If not stated otherwise, each arc of the network forms a trivial device. The sets are defined as follows:

1. The set $A_{VF}$ connects every vertex $v$ with its adjacent faces: $A_{VF} = \{a_e^V = (n_v, n_f) | v \in V, e \in \mathcal{E}(v), f = face(e)\}$. Arcs in $A_{VF}$ have cost zero and capacity three.
2. The set $A_{FF}$ connects two faces which share an edge: $A_{FF} = \{a_e^F = (n_f, n_g) | e \in \bar{E}, f = face(e), g = face(\bar{e})\}$. Arcs in $A_{FF}$ have cost one and capacity $\infty$.
3. Arcs in $A_{FH}$ are vertex-bend edges connecting the face to the helper node and form the only non-trivial devices: $A_{FH} = \bigcup_{e \in \bar{E}} d_e$ where $d_e = \{a_e^L = (n_f, n_{e'}) | f = face(e), e' = h(e, 0)\} \cup \{a_e^R = (n_f, n_{e'}) | f = face(\bar{e}), e' = h(e, 1)\}$. Arcs in $A_{FF}$ have cost one and each device has capacity one.
4. The set $A_{HV}$ connects the helper nodes with vertex-nodes: $A_{HV} = \{a_e^0 = (n_e, n_v) | v \in V, e \in \mathcal{E}(v)\}$. Arcs in $A_{HV}$ have cost zero and capacity one.

For a valid flow $x$ of the `Kandinsky` network $\mathcal{N}_G^K$ of a plane graph $G = (V, E, F)$ we define the quasi-orthogonal shape $Q^x$ as follows:

$$\begin{aligned} Q_a^x(e) &= x(a_e^V) - x(a_e^0) + 1 \;\; \forall e \in \bar{E} \\ Q_b^x(e) &= vb(e,x) fb(e,x) \overline{vb(\bar{e},x)} \end{aligned}$$

with

$$\begin{aligned} vb(e,x) &= 0^{x(a_e^L)} 1^{x(a_e^R)} \\ fb(e,x) &= 0^{x(a_e^F)} 1^{x(a_{\bar{e}}^F)} \end{aligned}$$

for $e \in \bar{E}$.

**Theorem 4.4 ([56])** *Given a plane graph $G$ and a flow $x$ in the KANDINSKY BEND MINIMIZATION network $\mathcal{N}_G^K$. The quasi-orthogonal shape $Q^x$ is valid if, and only if $x$ is feasible. For a feasible flow $x$ holds $z(x) = \#bends(Q^x)$.*

## 4.4 Solving the `Kandinsky` Network Flow Problem

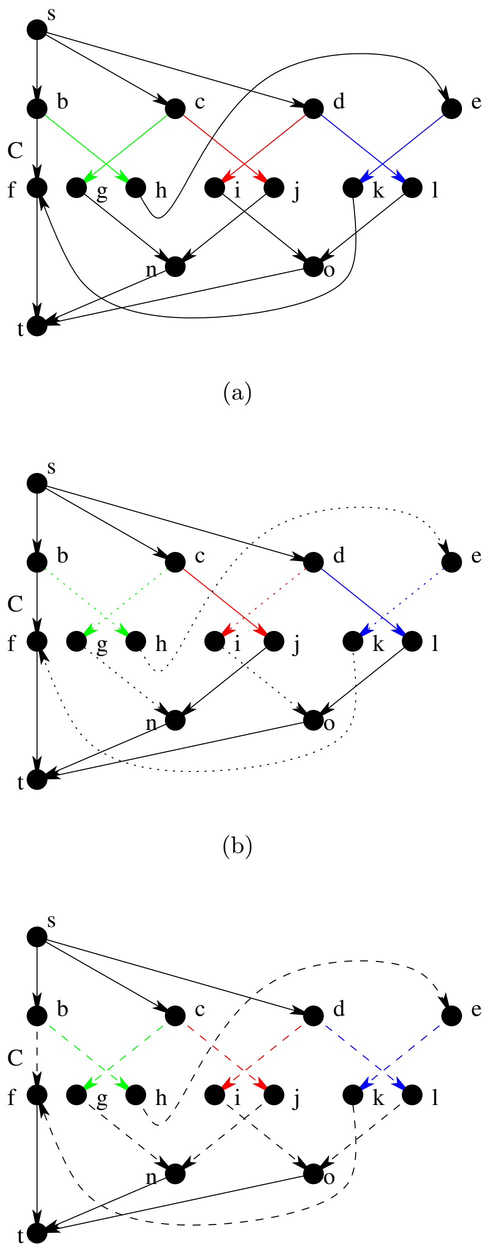
In the last section we modeled the KANDINSKY BEND MINIMIZATION problem as an arc partition minimum cost flow problem. In this section we will discuss how we can solve this problem.

We will first show that solving arc partition minimum cost flow problems is hard in general. Then we discuss the algorithm of Fößmeier to solve `Kandinsky` networks to optimality and show that this algorithm is not correct, it may even fail to return a feasible flow. We present a modified algorithm that at least returns a feasible solution. This solution is not optimal in general which has as consequence that the complexity of the KANDINSKY BEND MINIMIZATION problem is unknown. Finally we present a new approximation algorithm for the minimum cost flow problem in `Kandinsky` networks which computes a 2-approximation of the optimal solution.

### 4.4.1 Complexity of Solving Arc Partition Minimum Cost Flow Networks

While minimum cost flow networks have always an optimal integral solution if an optimal solution exists, this is not true for arc partition minimum cost flow networks. We call the difference between the optimal integral solution and the optimal fractional solution the *integrality gap.* See Figure 4.10 for a network with positive integrality-gap.

While an optimal fractional solution of an arc partition minimum cost flow network can be found in polynomial time using linear programming, the problem of finding an optimal integral solution is NP-hard, even in the case of small devices and low device capacities.



(a)

(b)

(c)

Figure 4.10: Arc partition minimum cost flow network which has only non-integral optimal solution. The network is shown in (a), all network arcs have cost zero, except arc $(b, f)$ which has cost $C > 0$. The black arcs are each a trivial device. Each set of green, red and blue arcs define a device. All devices have capacity one. In (b) an optimal integral flow is shown, with cost $C$. The solid arcs are saturated while the pointed arcs are empty. In (c) an optimal non-integral flow is shown. The solid arcs are saturated, and the dashed arcs carry flow 0.5. The cost of the flow is $C/2 < C$.

**Theorem 4.5** *Finding an optimal solution of an arc partition minimum cost flow network is NP-hard, even in the case that the maximal number of arcs in a device is bounded by two and the maximal capacity of devices is one.*

**Proof:** We will reduce 3-SAT to the integral arc partition minimum cost flow problem to show that this problem is NP-hard. Let $\{y_i|0 \leq i < n\}$ be a set of literals and $C = \{C_j|0 \leq j < m\}$ a set of clauses with $C_j = \{c_{j_1} \vee c_{j_2} \vee c_{j_3}|0 \leq j_i < n, c_{j_i} \in \{y_{j_i}, \bar{y}_{j_i}\}, 0 \leq i < 3\}$. We describe an arc partition network $\mathcal{N} = (V, A)$ which has an integral minimum cost flow of zero, if, and only if, the clause $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ is satisfiable.

We create for each literal $y_i$, $0 \leq i < n$ a node $n_{y_i}$ which is connected with capacity one to the network source. Additionally we create two vertices $n^0_{y_i}$ and $n^1_{y_i}$, both connected to $y_i$ with an arc of capacity one. We call these nodes the *input layer*. In the input layer the assignment of the values to the literals is done. A flow entering a node $n^0_{y_i}$ denotes that literal $y_i$ has value 0 and a flow entering a node $n^1_{y_i}$ denotes that literal $y_i$ has value 1. From the construction of the input layer follows that a flow of value one enters either $n^0_{y_i}$ or $n^1_{y_i}$.

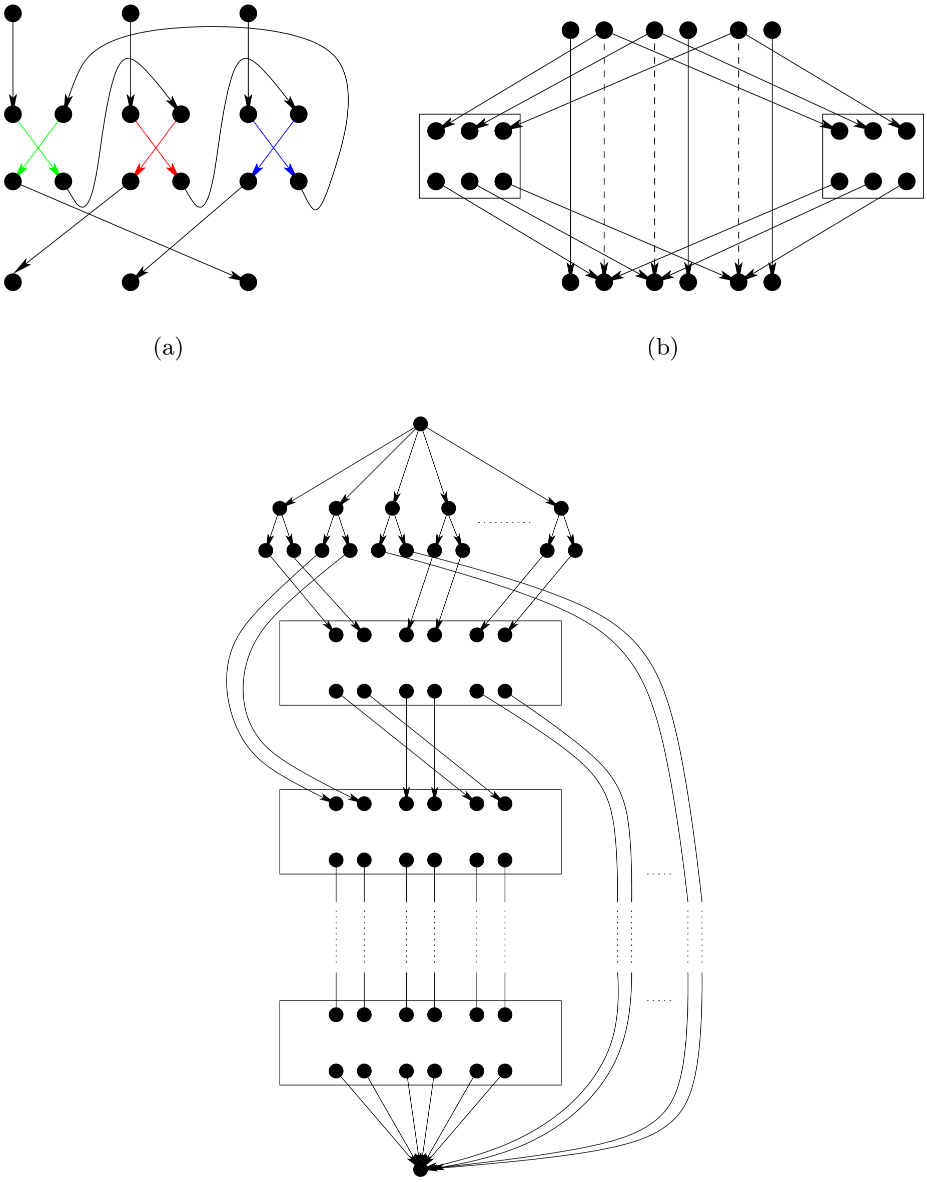
For each clause $C_j$ we create a clause-network with six input nodes $\{a^{lk}_{C_j}|0 \leq l \leq 2, 0 \leq k \leq 1\}$ and six output nodes $\{b^{lk}_{C_j}|0 \leq l \leq 2, 0 \leq k \leq 1\}$. A clause-network has the property that there is outgoing flow on an output node $b^{lk}_{C_j}$ if, and only if there is incoming flow on an input node $a^{lk}_{C_j}$.

We connect the clause-networks the following way: For each $a^{lk}_{C_j}$ we create an arc from the output node $b^{lk}_{C_i}$ of a subgraph which represents the last clause $C_i$, $i < j$ which contains literal $y_{j_l}$, to $a^{lk}_{C_j}$. If no such output node exists, because it is the first occurrence of literal $y_{j_l}$ in a clause, we create an arc from $n^k_{y_{j_l}}$ to $a^{lk}_{C_j}$. Additionally we connect each output node which has no outgoing arc to the network sink. See Figure 4.11 (c) for an illustration.

At the clause-network of clause $C_j$ there is flow on the incoming arc of exactly one node of each pair $a^{l0}_{C_j}$, $a^{l1}_{C_j}$ for $0 \leq l \leq 2$. Therefore for each literal there is a flow from the source to the sink passing through the clause-networks of clauses in which the literal occurs.

Assigning the correct value to one literal in the clause causes the whole clause to evaluate to true. Therefore a clause has one combination of input values for which it evaluates to false, while all other combinations of input values evaluate to true. For the one combination of input values, for which the clause evaluates to false, the only flow through the clause-network has cost $C$, while for all other combinations of input values, there is a zero cost flow through the clause-network.

We achieve this the following way: Input nodes corresponding to literal values satisfying the clause are connected directly to the corresponding



(a) (b)

(c)

Figure 4.11: A 3:1 filter which allows only one of the three input/output pairs to send flow is shown in (a). Black arcs are trivial devices, the other colors define the remaining devices. In (b) a clause-network is depicted. The dashed lines have positive costs. The boxes on the left and on the right represent a sub-network like shown in (a). The global structure of the network of the reduction is shown in (c). On the top we see the input layer followed by the clause networks and then the network sink.

output node with an arc of cost zero. Input nodes not-corresponding to literal values satisfying the clause are connected directly to the corresponding output node with an arc of cost $C$. These input nodes are additionally connected to a small subnetwork which lets a flow of two pass with zero cost. This subnetwork is composed of two *3:1-filter* which have three input/output node pairs. A 3:1 filter has a maximum flow of one and lets only pass flow from one input node to the corresponding output node. See Figure 4.11 (a) for an illustration. The third must take an arc with positive cost.

Assume that clause $C_j$ has the form $y_{j_0} \vee \bar{y_{j_1}}\ y_{j_2}$. Then the nodes $a^{01}_{C_j}$,$a^{10}_{C_j}$ and $a^{21}_{C_j}$ are connected directly by an arc of cost zero to $b^{01}_{C_j}$,$b^{10}_{C_j}$ and $b^{21}_{C_j}$. Then the nodes $a^{00}_{C_j}$,$a^{11}_{C_j}$ and $a^{20}_{C_j}$ are connected to $b^{01}_{C_j}$,$b^{10}_{C_j}$ and $b^{21}_{C_j}$ through and arc of cost $C$. Additionally the nodes $a^{00}_{C_j}$,$a^{11}_{C_j}$ and $a^{20}_{C_j}$ are connected to $b^{01}_{C_j}$,$b^{10}_{C_j}$ and $b^{21}_{C_j}$ through two 1:3 filters.

The network described above has maximum-flow $n$, its size is linear with respect to the input clause and it can be computed in polynomial time.

It follows from the construction of the network that it has a minimum flow of cost 0 if, and only if, the corresponding clause $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ is satisfiable. □

### 4.4.2 The Negative-Cycle Approach to Solve the KANDINSKY BEND MINIMIZATION Problem

Although solving arc partition minimum cost flow networks is NP-hard, the complexity of solving the KANDINSKY BEND MINIMIZATION problem is unknown. In [56] the authors propose to replace each non-trivial device in the `Kandinsky` network by a subnetwork containing a negative cost cycle, and then to solve the transformed network. The transformation is shown in Figure 4.12. A solution of the network is only valid if for every saturated negative cycle one of the two arcs with cost $2C + 1$ is also saturated. The cost $C$ is chosen, such that it is an upper bound of the optimal solution of the network. Therefore an optimal solution of the network can use only one of the $2C+1$ arcs, which is exactly the behavior which we want. The authors propose to use the successive shortest path (SSP) network flow algorithm to solve the transformed network.

We will review the SSP algorithm now, for the rest of the section we assume that the arcs in a minimum cost flow network have non-negative costs. The *residual network* $\mathcal{N}(x)$ of minimum cost flow network $\mathcal{N} = (N, A)$ for a given flow $x$ is defined as $\mathcal{N}$ minus the saturated arcs in $A$ plus for every arc $(i, j) \in A$ with $x_{ij} > 0$ an arc $(j, i)$ with $c_{ji} = -c_{ij}$. The *residual capacity* $r_{ij}$ of an arc $(i, j)$ in $\mathcal{N}(x)$ is $u_{ij} - x_{ij}$ if $(i, j) \in A$ and $x_{ji}$ otherwise. The SSP algorithm starts with a zero flow and then computes in every iteration a shortest path from the source to the sink in the residual network. Along this shortest path it augments the flow until all flow is sent or one arc in the

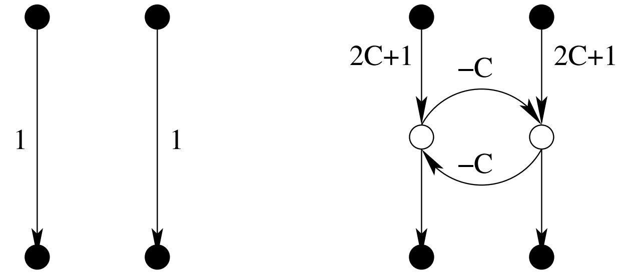


Figure 4.12: Transformation of a device into two paths connected by a negative cycle.

residual network is saturated. Then the new residual network is computed and the next iteration takes place. The algorithm stops when a feasible flow is found.

For a flow $x$ the residual arcs of all arcs in $A$ with positive flow have non-positive costs in the residual network. Therefore computing a shortest path in the residual network with respect to the cost function would require to solve a shortest path problem with negative arcs, a problem for which the best known algorithm has running time $O(|N||A|)$. The SSP algorithm uses *reduced costs* to avoid negative cost arcs which enables us to use Dijkstra's algorithm which has complexity $O((|N| + |A|) + |N| \log |N|)$. We define for each node in $i \in N$ a *potential* $\pi(i)$. The reduced cost $c^{\pi}_{ij}$ of an arc $(i, j)$ is defined as $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$. The SSP algorithm starts with zero potential for each node and updates the potential after each augmentation step. The update for a node $v$ consists of subtracting the shortest path distance of $v$ in the current step from $\pi(v)$.

Since finding a shortest path in a network with negative cycles is NP-complete, there is no generic algorithm to solve the transformed `Kandinsky` network. The authors propose to use the SSP algorithm with a modified version of the Dijkstra's shortest path algorithm: When in the Dijkstra's algorithm the distance to a node is updated and this node is adjacent to a negative cycle, then we walk one time around this negative cycle, collect the negative costs and add them to the distance. In the case of `Kandinsky` networks this method computes indeed the shortest paths and the authors claim that therefore the modified SSP algorithm finds a minimum cost flow for `Kandinsky` networks. This would yield an $O(n^2 \log n)$ algorithm for the KANDINSKY BEND MINIMIZATION problem. However, this is not true since the proposed modification works only on the original network but not on the residual network. The reason for this is that we cannot assign potentials to nodes adjacent to saturated negative cycles such that all reduced costs are non-negative. As a consequence for arcs which are adjacent to such nodes we can not define residual arcs in the residual network and therefore it is not possible to cancel flow which is sent along such an arc. Translating

Algorithm 6: Successive Shortest Path Algorithm

```
Input: Network $\mathcal{N} = (N, A)$, $s, t \in N$, costs $c$, residual capacities $r$
Output: Minimum cost flow $x$
$x = 0$ ;
$\pi = 0$ ;
$flow = 0$ ;
while $flow < b(s)$ do
    determine shortest path distances $d(i)$ from node $s$ to all other
    nodes in $\mathcal{N}(x)$ with respect to the reduced costs $c^{\pi}_{ij}$;
    Let $P$ denote a shortest path from $s$ to $t$ ;
    update $\pi = \pi - d$ ;
    $\delta = \min[flow, \min\{r_{ij} : (i, j) \in P\}]$;
    augment $\delta$ units of flow along path $P$;
    $flow = flow + \delta$ ;
    update $x$, $\mathcal{N}(x)$ and reduced costs $c^{\pi}$;
```

this algorithm to the original arc partition minimum cost flow network formulation, it can be seen as the SSP algorithm with the modification that the residual network does not contain residual arcs of devices with more than one arc. In Figure 4.13 we show an example in which this algorithm leads to sub-optimal solution.

Even worse the algorithm may not even yield a feasible flow. For a vertex $v$ with degree $\delta(v) > 4$ in the input graph the node $n_v$ in the `Kandinsky` network is connected to the sink with an arc of capacity $\delta(v) - 4$. In a feasible flow this arc must be saturated. The only possibility for a flow to enter this arc is to pass through a `Kandinsky` device at the node $v$. Since there are $\delta(n)$ `Kandinsky` devices, at most 4 `Kandinsky` devices may be not saturated.

The SSP algorithm may execute such that more than 4 `Kandinsky` devices are not saturated. Assume that after some iterations of the SSP algorithm, for a vertex $v$ with degree $\delta(v) \geq 15$ there may be five configurations as shown in Figure 4.14 in the list of `Kandinsky` devices of $v$. Each configuration forces one `Kandinsky` device to be non-saturated. Therefore the arc from $n_v$ to the network sink $t$ cannot be saturated and the algorithm fails to send the total amount of supply through the network.

We can repair the modified SSP algorithm the following way: After each augmentation of flow along a path we count at each vertex $v$ of the input graph the number of changes in the bend direction in the cyclic list of the `Kandinsky` devices. A change in the bend direction is counted if between two edges $e_1$ and $e_2$ in the embedding on $v$ all edges have no vertex-bend and $a^L_{e_1} = 1$ and $a^R_{e_2} = 1$ or $a^L_{e_2} = 1$ and $a^R_{e_1} = 1$. A flow in a `Kandinsky` network cannot be completed to a feasible flow if the number of bend changes is

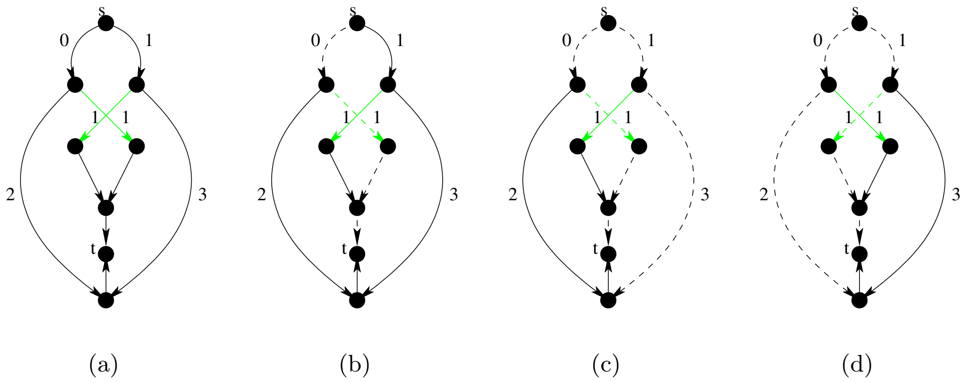


Figure 4.13: Example for a non-optimal solution computed by the SSP algorithm. Figure (a) shows a part of a `Kandinsky` network. Straight lines depict edges, arcs depict paths in the network. All arcs and paths have capacity one. The cost is placed near the arcs, if no cost is placed the cost is zero. All arcs are trivial devices except the two green one which are in one device. In Figure (b) we show the first shortest path calculated by the SSP algorithm and in (c) the second which then yields a solution with value 5. Figure (d) shows the optimal solution which has value 4.

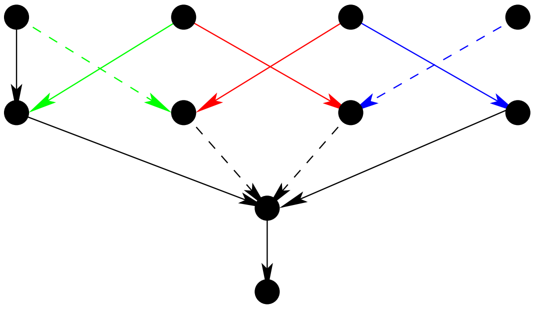


Figure 4.14: Example for a flow which blocks a device (red arcs) without sending flow through one of its arcs. The arcs which are represented by dashed lines have flow one, the arcs represented by solid lines flow zero.

greater than eight. If we count eight bend changes at a vertex, we remove all arcs from the network which could generate a new bend change. These are arcs in a device which define a left, resp. right, bend and which are in the embedding between two devices for which the current flow defines a left , resp. right bend. We can implement the counting and updating in linear time, such that the asymptotic running time of the algorithm does not change.

### 4.4.3 A 2-Approximation Algorithm

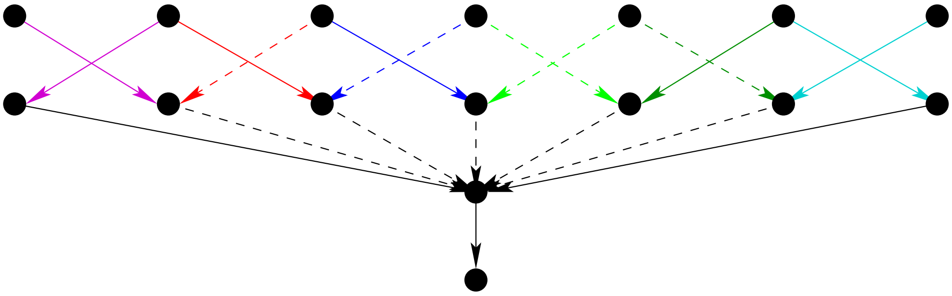
The SSP algorithm we presented in the previous section has the disadvantage that we cannot give any performance guarantees for it. In this section we present an algorithm which computes a feasible flow in a `Kandinsky` network which has at most twice the cost of an optimal solution.

We define the *minimum cost flow relaxation* of an arc partition minimum cost flow network as the corresponding minimum cost flow network in which to each arc the capacity of its device is assigned. Clearly an optimal solution of the minimum cost flow relaxation is a lower bound for the arc partition minimum cost flow problem.

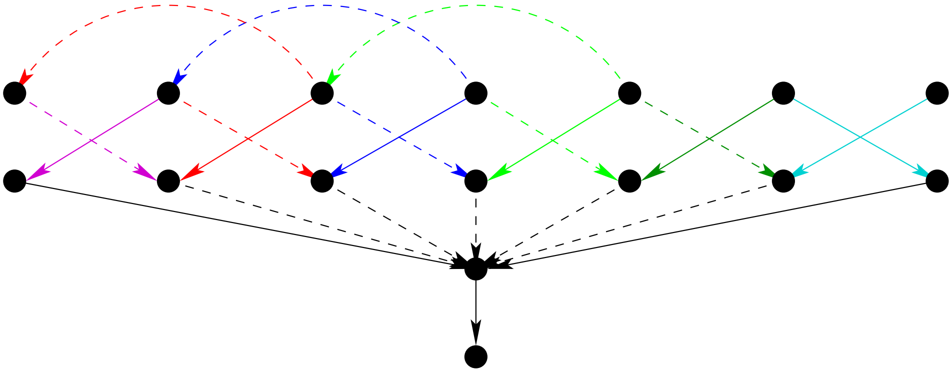
Let $x$ be a feasible flow of the minimum cost flow relaxation, and $d$ be a non-trivial device. We call $d$ *over-saturated* with respect to $x$ if $\sum_{e \in d} x(e) > u_d$. A solution $x$ of the relaxation is a feasible flow of the `Kandinsky` minimum cost flow network if, and only if, it contains no over-saturated device. We denote with $D(v)$ the cyclic list of the `Kandinsky` devices around $v$ for each vertex $v$ of the input graph. Each sub-list in $D(v)$, which consists of non-empty devices contains at most one over-saturated device. Let $I(v)$ denote the set of maximal sub-lists of $D(v)$ which consist of non-empty devices and one over-saturated device. For $I \in I(v)$, we denote with $l(I)$, resp. $r(I)$, the number of devices before, resp. after, the over-saturated device in $I$. We can transform $x$ to a flow $\hat{x}$ such that $I$ does not longer contain this over-saturated device at cost either $2 \cdot (l(I) + 1)$ with a *left-transformation* or cost $2 \cdot (r(I) + 1)$ with a *right-transformation*. See Figure 4.15 for an illustration.

Our approximation algorithm, called the *transformation algorithm*, for solving a `Kandinsky` network first computes an optimal solution $x$ of the relaxation. Then it determines the sets $I(v)$ for all vertices $v$ in the input graph. We partition set of vertices in the input graph into two sets $V_l$ and $V_r$. For the vertices in $V_l$ holds that $\sum_{I \in I(v)} l(I) \leq \sum_{I \in I(v)} r(I)$, $V_r$ contains the remaining vertices. Then we transform $x$ into a flow $\hat{x}$ by applying left transformations to each over-saturated device of $v \in V_l$ and right-transformation to each over-saturated device in $v \in V_r$.

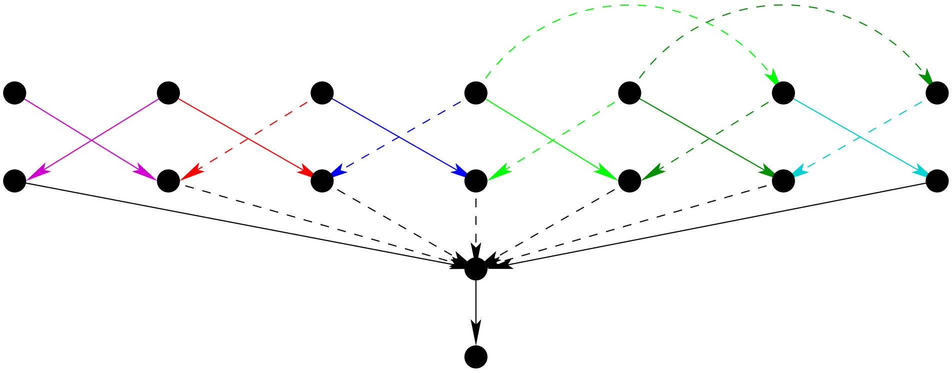
**Theorem 4.6** *The transformation algorithm computes a* 2*-approximation for the* KANDINSKY BEND MINIMIZATION *problem in running-time* $O(n^{\frac{7}{4}} \cdot$



(a) An example for an maximal sub-list which consist of non-empty devices and one over-saturated device. The arcs in the network represented by dashed lines have flow one, the arcs in solid lines flow zero.



(b) Example for a left-transformation.



(c) Example for a right-transformation.

Figure 4.15: Illustration of left- and right-transformations.

$\sqrt{\log n})$.

**Proof:** We define

$$V_l = \{v \in V | \sum_{I \in I(v)} l(I) \le \sum_{I \in I(v)} r(I)\},$$
$$V_r = \{v \in V | \sum_{I \in I(v)} l(I) > \sum_{I \in I(v)} r(I)\} \ .$$

Let $\hat{x}$ be the flow resulting from the transformation. The objective value of $\hat{x}$ is the objective value of $x$ plus the cost of the transformation. It holds $z(x) \ge \sum_{v \in V} \sum_{I \in I(v)} (|I| + 1)$ since every device in a sub-list $I$ contributes with one to the objective function and one device is over-saturated which counts for two. Since $|I| = l(I) + r(I) + 1$ it holds $2 \cdot (l(I) + 1) \le |I| + 1$ for $l(I) \le r(I)$, and $2 \cdot (r(I) + 1) \le |I| + 1$ for $r(I) \le l(I)$. For the cost of the transformations follows then

$$\begin{aligned} \sum_{v \in V_l} \sum_{I \in I(v)} 2 \cdot (l(I) + 1) + \sum_{v \in V_r} \sum_{I \in I(v)} 2 \cdot (r(I) + 1) \le \\ \sum_{v \in V_l} \sum_{I \in I(v)} (|I| + 1) + \sum_{v \in V_r} \sum_{I \in I(v)} (|I| + 1) = \\ \sum_{v \in V} \sum_{I \in I(v)} (|I| + 1) \le z(x). \end{aligned}$$

Let $\bar{x}$ be an optimal solution of the `Kandinsky` minimum cost flow network, then for the cost of transformed flow holds

$$z(\hat{x}) \le 2 \cdot z(x) \le 2 \cdot z(\bar{x}) \ .$$

Because the transformation can be done in linear time, the running time is dominated by the running time of the algorithm to solve the minimum cost flow relaxation, which is $O(n^{\frac{7}{4}} \cdot \sqrt{\log n})$ with the same arguments as in Theorem 4.2. □

### 4.4.4 An Improved Heuristic

The approximation algorithm of the previous section transformed the optimal solution of the minimum cost flow relaxation of a `Kandinsky` network into a valid solution of the `Kandinsky` network. It changes explicitly the flow values in intervals which contain over-saturated `Kandinsky` devices.

We can improve the performance of the algorithm by changing the flow implicitly. We again compute an optimal solution of the minimum cost flow relaxation. Instead of transforming the flow, we change the capacities in the network and solve the relaxation again. We set capacities of all the arcs which have positive flow in the solution of the relaxation and zero flow in the transformed solution to zero. We repeat until we obtain a solution from the relaxation which contains no over-saturated devices.

Since in each iteration we remove at least one arc from a `Kandinsky` device, the algorithm terminates after $n$ iterations with a solution without over-saturated devices. The total running time of the algorithm is therefore $O(n^{\frac{11}{4}} \cdot \sqrt{\log n})$.

In practice the number of iterations is very low and more iterations than 3 are hardly encountered.

## 4.5 Constraints in `Kandinsky`

In this section we present the CONSTRAINED KANDINSKY BEND MINIMIZATION problem which is a generalization of the KANDINSKY BEND MINIMIZATION problem. The CONSTRAINED KANDINSKY BEND MINIMIZATION problem takes besides a planarization a set of low-level constraints as input. These low level constraints can define target values for the number and types of bends at edges and for the angles formed by intervals of neighboring edges around a vertex in the embedding. We will present an extended version of the `Kandinsky` network in which we can specify low level constraints for the resulting shape. Until now it was only possible to specify such constraints using integer linear programming [46].

We will now define the two type of low-level constraints: *angle-constraints* and *bend-constraints*:

**Definition 4.11** *Let $G = (V, E, F)$ be a plane graph. An* angle-constraint *is a tuple $(I, v, a, c)$. The first entry $I$ is a list of darts, which is an interval inside $\mathcal{E}(v)$ for $v \in V$. The entry $a$ is an integer with $0 \leq a \leq 4$ denoting the target-value for the sum of the angles defined by the darts in the interval, and the last entry $c$ is a non-negative integer, denoting a cost-factor for derivations of the target-value of the angle.*

Let $AC$ be a set of angle-constraints for a plane graph $G$, then $AC$ is called *complete* if for each dart in $e \in \bar{E}$ there is exactly one angle-constraint $(I, v, a, c) \in AC$ with $e \in I$. In a complete set of angle-constraints each angle in a quasi-orthogonal shape $Q$ of $G$ is part of *exactly* one constraint. When we have a set of angle-constraints $AC$ in which each angle is contained in *at most* one constraint we can easily construct an equivalent complete angle-constraint set $AC'$ by adding an angle-constraint with cost zero and arbitrary target-angle for each angle not covered by $AC$. We assume therefore for the remainder of this chapter that sets of angle-constraints are complete. The set of all angle constraints for a vertex $v$ is denoted by $ac(v)$.

**Definition 4.12** *Let $G = (V, E, F)$ be a plane graph. A* bend-constraint *is a tuple $(e, b, c_i, c_d)$. The first entry denotes a dart in $\bar{E}$, therefore $e \in \bar{E}$.*

*The second entry $b$, a bit-string, defines the bends of the dart. The third entry $c_i$ is the cost of inserting a new bend into the dart, and $c_d$ defines the cost for a bend in $b$ which is not considered.*

Let $BC$ be a set of bend-constraints for a plane graph $G$, then $BC$ is called *complete* if for each edge $e \in E$ there is exactly one bend-constraint $(e, b, c_i, c_d) \in BC$, for one of the two darts of $e$. When we have a set of bend-constraints $BC$ in which one dart of each edge is contained in *at most* one constraint we can easily construct an equivalent complete bend-constraint set $BC'$ by adding a bend-constraint with bit-string $\epsilon$, arbitrary deletion cost and insertion cost one for a dart of each edge not covered by $BC$. We assume therefore for the remainder of this chapter that sets of bend-constraints are complete. For a bend constraint $(e, b, c_i, c_d) \in BC$ we denote with $bc(e) = (e, b, c_i, c_d)$ and $bc(\bar{e}) = (\bar{e}, \overleftarrow{b}, c_i, c_d)$.

**Definition 4.13** *Let $G = (V, E, F)$ be a plane graph, $AC$ be a complete set of angle-constraints and $BC$ a complete set of bend-constraints. The* CONSTRAINED KANDINSKY BEND MINIMIZATION *problem consist of finding a* `Kandinsky` *shape $Q$ which minimizes $z(Q, AC, BC)$ where*

$$z(Q, AC, BC) = z(Q, AC) + z(Q, BC)$$

*with*

$$z(Q, AC) = \sum_{(I,v,a,c)\in AC} c \cdot |a - \sum_{e\in I} Q_a(e)|$$

$$z(Q, BC) = \sum_{(e,b,c_i,c_d)\in BC} edit_{(c_i,c_d)}(b, Q_b(e)).$$

To solve the CONSTRAINED KANDINSKY BEND MINIMIZATION problem we modify the original `Kandinsky` formulation.

We define for each angle-constraint $ac = (I, v, a, c)$ on a vertex $v$ an *angle-node* $n_{ac}$. We create two arcs, one from $n_{ac}$ to the vertex-node $n_v$ of $v$ and one in the opposite direction. Both arcs have unconstrained capacity and cost $c$. We denote with $N_A$ the set of angle-constraint nodes. Angle-constraint nodes are the only nodes connected to vertex-nodes for constrained angles. All arcs which were connected to vertex-nodes in the original formulation are connected to the corresponding angle-nodes. The supply of an angle node is its target value $a$ minus the number of edges in the interval, since zero degree angles correspond to a flow of -1 in the node. We remove $a - |I|$ from the supply of the vertex-node to keep the maximum flow in the network constant. We define the supply of a vertex-node as

$$b(n_v) = \delta(v) - 4 - \sum_{(I,w,a,c)\in ac(v)} (a - |I|) \ .$$

This construction is illustrated in Figure 4.16.

The second modification concerns the modeling of bends. We treat each bend in a bend constraint $(e, b, c_i, c_d)$ as a node of degree two. We distinguish between two types of bends: bends in the middle of the edge, in other words bends which have a predecessor and successor on the edge, and bends at the end of an edge. A bend of the first type is a face-bend in all cases, while a bend of the second type may be a vertex-bend or a face-bend. For each bend, that is for each $0 \leq i < |b|$ we create two nodes $n_{(e,i,0)}$ and $n_{(e,i,1)}$ in the network. The node $n_{(e,i,0)}$ has supply two and has an edge of cost zero and capacity one to $n_{(e,i,1)}$. A demand of one is added to both faces incident to the edge. Let $f_1$ be the face for which the bend is a concave bend, and $f_2$ the face for which the bend is a convex bend. Then we add an edge of cost zero and capacity two between $n_{(e,i,0)}$ and the face-node $f_1$, and an edge with cost $c_d$ and capacity one between $n_{(e,i,1)}$ and $f_2$. This construction is shown left in Figure 4.17. The effect of the construction is that either the resulting quasi-orthogonal shape contains the bend at zero cost or the bend is removed at cost $c_d$. Bends of the second type are treated like bends of the first type, except that we introduce an additional arc which connects $n_{(e,i,1)}$ to the helper node on the side of $f_2$ with zero cost. This edge ensures that vertex-bends can be confirmed at zero cost. This construction is shown right in Figure 4.17. Additionally we assign arcs between faces node defined by $e$ the cost $c_i$. Since we add for each bend in a bend constraint supply to the face nodes we have to redefine the supply of a face-node as

$$b(n_f) = rot(f) + \sum_{e \in f, bc(e)=(e,b,c_i,c_d)} |b| \ .$$

We will give now a formal description of the network $\mathcal{N}^{CK}_{G,AC,BC}$ whose feasible solutions correspond to valid `Kandinsky` shapes and whose optimal solution solves the CONSTRAINED KANDINSKY BEND MINIMIZATION problem.

Let $G = (V, E, F)$ be a plane graph, $AC$ a complete set of angle constraints and $BC$ a complete set of bend constraints. The set of nodes $N$ of the network $\mathcal{N}^{CK}_{G,AC,BC}$ is defined as

$$N = N_V \cup N_F \cup N_H \cup N_A \cup N_B$$

with

1. The set $N_V$ contains a node for each vertex in $V$: $N_V = \{n_v | v \in V\}$. The supply of a vertex-node is defined as $b(n_v) = \delta(v) - 4 - \sum_{(I,v,a,c) \in ac(v)} (a - |I|)$.
2. The set $N_F$ contains a node for each face in $F$: $N_F = \{n_f | f \in F\}$. The supply of a face-node is defined as $b(n_f) = -rot(f) - \sum_{e \in f, bc(e)=(e,b,c_i,c_d)} |b|$.

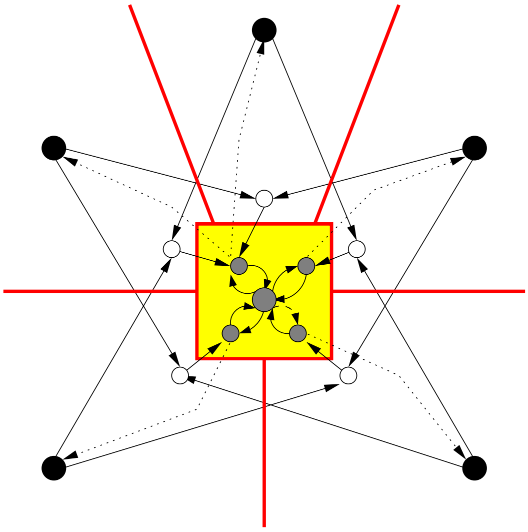


Figure 4.16: Modifications in the `Kandinsky` network modeling an angle constraint.

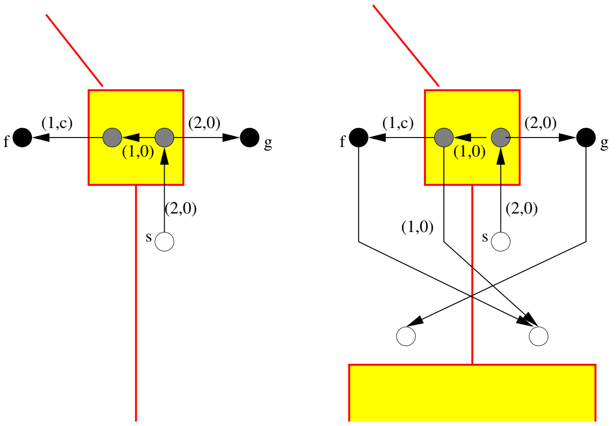


Figure 4.17: Modified Kandinsky network for modeling a bend constraint. The left figure shows a face-bend, the right figure a vertex-bend. Newly introduced arcs are labeled with capacity and cost. Newly introduced nodes are gray. The cost $c$ is the deletion cost specified in the bend constraint, and node $s$ denotes the source of the network.

3. The set $N_H$ contains a node for each dart in $\bar{E}$: $N_H = \{n_e | e \in \bar{E}\}$.
   The supply of a helper-node is defined as $b(n_e) = 0$.

4. The set $N_A$ contains a node for each angle constraint in $AC$:
   $N_A = \{n_{ac} | ac \in AC\}$. The supply of a node of an angle constraint $ac = (I, v, c, a) \in AC$ is defined as $b(n_{ac}) = a - |I|$.

5. The set $N_B$ contains two nodes for each bend in a bend-constraint:
   $N_B = \{n_{(e,i,0)}, n_{(e,i,1)} | (e, b, c_i, c_d) \in BC, 0 \le i < |b|\}$.
   The supply of each bend-constraint-node is defined as $n_{(e,i,j)} = 2$ for $0 \le i < |b|$, $0 \le j < 2$, $bc(e) = (e, b, c_i, c_d)$.

The set of arcs $A$ of $N^{CK}_{G,AC,BC}$ is defined as

$$A = A_{FF} \cup A_{FH} \cup A_{AF} \cup A_{HA} \cup A_{AV} \cup A_{VA} \cup A_{BF} \cup A_{BB} \cup A_{BH}$$

with

1. The set $A_{HA}$ connects the helper nodes with the corresponding angle-node:
   $A_{HA} = \{a^0_e = (n_e, n_{ac}) | ac = (I, v, a, c), e \in I\}$.
   Arcs in $A_{HA}$ have cost zero and capacity 1.

2. The set $A_{AF}$ connects every angle node with its adjacent faces:
   $A_{AF} = \{a^V_e = (n_{ac}, n_f) | ac = (I, v, a, c), e \in I, f = \mathit{face}(e)\}$.
   Arcs in $A_{AF}$ have cost zero and capacity 3.

3. The set $A_{AV}$ connects angle-constraint nodes with the vertex-node:
   $A_{AV} = \{(n_v, n_{ac}) | ac = (I, v, a, c) \in AC\}$.
   Arcs $(n_v, n_{ac})$ in $A_{AV}$ have cost $c$ and capacity $\infty$ for an angle-constraint $ac = (I, v, a, c)$.

4. The set $A_{VA}$ connects vertex-nodes with angle-constraint nodes:
   $A_{VA} = \{(n_{ac}, n_v) | ac = (I, v, a, c) \in AC\}$.
   Arcs $(n_{ac}, n_v)$ in $A_{VA}$ have cost $c$ and capacity $\infty$ for an angle-constraint $ac = (I, v, a, c)$.

5. The set $A_{FF}$ connects two faces which share an edge:
   $A_{FF} = \{a^F_e = (n_f, n_g) | e \in \bar{E}, f = \mathit{face}(e), g = \mathit{face}(\bar{e}), f \ne g\}$.
   Arcs $a^F_e$ in $A_{FF}$ have cost $c_i$ where $bc(e) = (e, b, c_i, c_d)$ and capacity $\infty$.

6. Arcs in $A_{FH}$ are bend-vertex edges connecting the face to the dummy node:
   $A_{FH} = \{a^L_e = (n_f, n_{e'}) | e \in \bar{E}, f = \mathit{face}(e), e' = h(e, 0)\} \cup \{a^R_e = (n_f, n_{e'}) | e \in \bar{E}, f = \mathit{face}(\bar{e}), e' = h(e, 1)\}$.
   Arcs $a^L_e, a^R_e$ in $A_{FH}$ have cost $c_i$ where $bc(e) = (e, b, c_i, c_d)$ and capacity $\infty$.

7. The set $A_{BF}$ connects bend-nodes to face-nodes of adjacent faces: $A_{BF} = \{a^B_{(e,i,b_i)} = (n_{(e,i,b_i)}, n_f), a^B_{(e,i,\bar{b}_i)} = (n_{(e,i,\bar{b}_i)}, n_g) | (e,b,c_i,c_d) \in BC, f = \mathit{face}(e), g = \mathit{face}(\bar{e}), 0 \leq i < |b|\}$.
   Arcs in $A_{FB}$ connecting to a $n_{(,,0)}$ node have cost zero and capacity two, arcs connecting to a $n_{(,,1)}$ node have cost $c_d$.

8. The set $A_{BB}$ connects the two vertex-bends: $A_{BB} = \{(n_{(e,i,b_0)}, n_{(e,i,b_1)}) | (e,b,c_i,c_d) \in BC, 0 \leq i < |b|\}$.
   Arcs in $A_{BB}$ have cost zero and capacity one.

9. The set $A_{BH}$ connects bend-nodes to helper nodes: $A_{BH} = \{a^C_e = (n_{(e,0,1)}, n_{e'}) | (e,b,c_i,c_d) \in BC, |b| > 0, e' = h(e,b_0)\} \cup \{a^C_{\bar{e}} = (n_{(e,|b|-1,1)}, n_{e'}) | (e,b,c_i,c_d) \in BC, |b| > 0, e' = h(\bar{e}, \overline{b_{|b|-1}}\}$.
   Arcs in $A_{BH}$ have cost zero and capacity one.

The only non-trivial devices in the network are

$$d_e = \begin{cases} \{a^R_e, a^L_e, a^C_e\} & |b| > 0 \\ \{a^R_e, a^L_e\} & |b| = 0 \end{cases}$$

for $e \in \bar{E}$ and $bc(e) = (e,b,c_i,c_d)$,

For a valid flow $x$ of the `Kandinsky` network $\mathcal{N}^{CK}_{G,AC,BC}$ of a plane graph $G = (V,E,F)$ with a complete set of angle constraints $AC$ and a complete set of bend-constraints $BC$ we define the quasi-orthogonal shape $Q^x$ as follows:

$$\begin{array}{rcl} Q^x_a(e) & = & x(a^V_e) - x(a^0_e) + 1 \;\; \forall e \in \bar{E} \\ Q^x_b(e) & = & vb(e,x) fb(e,x) \overline{vb(\bar{e},x)} \end{array}$$

with

$$\begin{array}{rcl} vb(e,x) & = & \begin{cases} 0^{x(a^L_e)} 1^{x(a^R_e)} b_0^{x(a^C_e)} & |b| > 0 \\ 0^{x(a^L_e)} 1^{x(a^R_e)} & |b| = 0 \end{cases} \\ fb(e,x) & = & 0^{x(a^F_e)} 1^{x(a^F_{\bar{e}})} \Pi_{0 \leq i < |b|} b_i^{x(a^B_{(e,i,0)})-1} \end{array}$$

for $e \in \bar{E}$, $bc(e) = (e,b,c_i,c_d)$. The number of nodes in the `Kandinsky` network $\mathcal{N}^K_{G,AC,BC}$ is

$$\begin{array}{rcl} |N| & = & |N_V| + |N_F| + |N_H| + |N_A| + |N_B| \\ & = & |V| + |F| + 2|E| + |AC| + \displaystyle\sum_{(e,b.c_i,c_d) \in BC} |b| \\ & \leq & 10|V| + |AC| + \displaystyle\sum_{(e,b.c_i,c_d) \in BC} |b| \;. \end{array}$$

For the number of edges holds:

$$\begin{aligned} |A| &= |A_{FF}| + |A_{FH}| + |A_{AF}| + |A_{HA}| + |A_{AV}| + |A_{VA}| \\ &\quad + |A_{BF}| + |A_{BB}| + |A_{BH}| \\ &= 2|F| + 4|E| + 2|E| + 2|AC| + 3 \sum_{(e,b.c_i,c_d)\in BC} |b| + 2|E| \\ &\leq 30|V| + 2|AC| + 3 \sum_{(e,b.c_i,c_d)\in BC} |b| \; . \end{aligned}$$

Thus the size of the network $\mathcal{N}^K_{G,AC,BC}$ is still linear with respect to the size of the planarization, the number of angle constraints and the size of the bend-constraints.

**Theorem 4.7** *Given a plane graph $G$ with a complete set of angle constraints $AC$ and a complete set of bend-constraints $BC$ and a flow $x$ in the CONSTRAINED KANDINSKY BEND MINIMIZATION network $\mathcal{N}^K_{G,AC,BC}$. The quasi-orthogonal shape $Q^x$ is valid if, and only if, $x$ is feasible, and for a feasible flow $x$ holds $z(x) = z(Q^x, AC, BC)$.*

**Proof:** Observe that the original `Kandinsky` network is modified in two parts. The first change is in the neighborhood of the vertex-nodes. The total amount of demand/supply in the neighborhood of a vertex-node $n_v$ for $v \in V$ remains unchanged since in the original network the supply of $n_v$ is $\delta(v) - 4$ and $b(n_v) + \sum_{(I,w,a,c)\in ac(v)}(a - |I|) = \delta(v) - 4$. Because for a vertex $v$ every node in $ac(v)$ can reach every other node in $ac(v)$ by a path of infinitive capacity, the changes in the network do not affect the set of feasible solutions, only the costs of it. The second changes are nodes and arcs introduced to model bend constraints. For each bend in a bend constraint the changes in the network do not affect the sum of supplies and demands since a node with supply two is added and to the face-nodes of the faces adjacent to the edge of the bend constraint demand one is added, and there is a path of capacity at least one from the new node to both face nodes. Therefore the set of quasi-orthogonal shapes corresponding to a feasible flow in the network is the same as in the original formulation, namely the set of all valid quasi-orthogonal shapes.

It remains to show that $z(x) = z(Q^x, AC, BC)$. There are three types of arcs in the network with positive cost: arcs of cost $c$ in the sets $AV$ and $VA$ between angles nodes and vertex-nodes for an angle constraint $(I, v, a, c)$, arcs of cost $c_i$ between face-nodes in the set $FF$ and $FH$, and arcs of cost $c_d$ between bend-nodes and face-nodes in the set $BF$. We will relate the flow on these arcs to $z(Q^x, AC, BC)$.

Consider an angle constraint $ac = (I, v, a, c) \in AC$. Because of flow conversation it holds at the angle-node $n_{ac}$ representing an angle constraint

$ac$:

$$\sum_{e \in I} (x(a_e^V) - x(a_e^0)) + x(n_{ac}, t) + x(n_{ac}, n_v) - x(n_v, n_{ac}) = 0 \ .$$

Since $x(n_{ac}, t)$ must be saturated it follows with the definition of $Q_a^x(e)$ that the above formula is equivalent to:

$$x(n_{ac}, n_v) - x(n_v, n_{ac}) = a - |I| - \sum_{e \in I} (Q_a^x(e) - 1) = a - \sum_{e \in I} Q_a^x(e) \ .$$

We assume that either $x(n_{ac}, n_v) = 0$ or $x(n_v, n_{ac}) = 0$, otherwise the flow would not be optimal.
If $x(n_{ac}, n_v) > 0$ then

$$x(n_{ac}, n_v) = a - \sum_{e \in I} Q_a^x(e) = |a - \sum_{e \in I} Q_a^x(e)|$$

If $x(n_v, n_{ac}) > 0$ then

$$x(n_v, n_{ac}) = \sum_{e \in I} Q_a^x(e) - a = |a - \sum_{e \in I} Q_a^x(e)|$$

Therefore $z(x_{|AV \cup VA}) = c \cdot |a - \sum_{e \in I} Q_a^x(e)| = z(Q^x, AC)$ .
Let $(e, b, c_i, c_d) \in BC$. We define an alignment of $b$ with $Q_b^x(e)$ the following way: If $x(a^B_{(e,i,1)}) = 1$ for $0 \leq i < 0$ we insert a dash in $Q_b^x(e)$ after position $i$, we insert $x(a_e^L) + x(a_e^R)$ dashes at the beginning of $b$, we insert $x(a_{\bar{e}}^L) + x(a_{\bar{e}}^R)$ dashes at the end of $b$, and we insert $x(a_e^F) + x(a_{\bar{e}}^F)$ dashes at position $a$ in b. This alignment is optimal, and therefore the weighted sum of dashes in the alignment is the edit distance between $b$ and $Q_b^x(e)$.

$$\begin{aligned} &\mathrm{edit}_{(c_i,c_d)}(b, Q_b^x(e)) = \\ &c_d(\textstyle\sum_{0 \leq i < |b|} x(a^B_{(e,i,1)})) + c_i(x(a_e^L) + x(a_e^R) + x(a_e^F) + x(a_{\bar{e}}^F) + x(a_{\bar{e}}^L) + x(a_{\bar{e}}^R)) \end{aligned}$$

Therefore $z(x_{|FF \cup FH \cup BF}) = \sum_{(e,b,c_i,c_d) \in BC} \mathrm{edit}_{(c_i,c_d)}(b, Q_b(e)) = z(Q^x, BC)$ .
It follows $z(x) = z(x_{|AV \cup VA}) + z(x_{|FF \cup FH \cup BF}) = z(Q^x, AC) + z(Q^x, BC) = z(Q^x, AC, BC)$. □

Since it is unknown if there is a polynomial time algorithm to solve the KANDINSKY BEND MINIMIZATION problem, it is also unknown if there is a polynomial time algorithm to solve the CONSTRAINED KANDINSKY BEND MINIMIZATION problem. Unfortunately the approximation algorithm from Section 4.4 does not yield an approximation for the CONSTRAINED KANDINSKY BEND MINIMIZATION problem. It seems to be hard to devise an approximation algorithm for this problem in general, but this is not as bad as one might think, since we will use only special classes of CONSTRAINED KANDINSKY BEND MINIMIZATION problems in this work. For these special classes we will provide approximation algorithms as they arise in the work.

## 4.6 Orthogonalization of UML Class Diagrams

In this section we present the orthogonalization algorithm for UML class diagrams. The input for the algorithm is:

- a mixed upward planarization $G = (V, E_d \cup E_u, F)$, and
- a map `type`: $V \rightarrow \{vertex, crossing, hypervertex\}$ .

We assume that the subgraph induced by the directed edges is connected. Furthermore we assume that the hypervertices have exactly one directed outgoing edge, a positive number of directed incoming edges, and no adjacent undirected edges. The output of the algorithm is a `Kandinsky` shape for which a drawing exists in which all directed edges point upward and the hypervertices are centered. Centered means in this case that the same number of predecessors of a hypervertex are on the left side and on the right side of the hypervertex.

The algorithm uses an orthogonalization algorithm for upward planarizations as subroutine. In a first phase it calculates the orthogonalization for the directed edges using this algorithm. Then the orthogonalization of the undirected edges is calculated while not changing the orthogonalization of the directed edges. This is done by creating constraints from the shape of the directed subgraph and then solving the corresponding CONSTRAINED KANDINSKY BEND MINIMIZATION problem.

This section is organized a follows: We investigate first the properties which a quasi-orthogonal shape of a mixed upward planarization must fulfill to have a mixed upward drawing. Then we present an algorithm for the orthogonalization of upward planarizations. Finally we present the orthogonalization algorithm for UML class diagrams.

### 4.6.1 Mixed Upward Orthogonal Drawings

In this section we will characterize the quasi-orthogonal shapes of a mixed upward planar graph for which a mixed upward drawing exists. The question can be stated formally as: Given a mixed upward planarization $G = (V, E_d \cup E_u, F)$ and a quasi-orthogonal shape $Q$ of $G$: Is there an upward orthogonal rectangle drawing $\Gamma$ of $G$ with quasi-orthogonal shape $Q$ ?

One property of a quasi-orthogonal shape $Q$ of a drawing $\Gamma$ is, that it is invariant under rotations. This stems from the fact, that the quasi-orthogonal shape contains only relative but no absolute angles. When dealing with undirected graphs, this is not important since the orientation of the edges in the drawing is not important. This does not hold for mixed graphs. For mixed graphs the orientation of the directed edges in the drawing is very important. To remove this ambiguity we define an *absolute quasi-orthogonal* shape which contains absolute values rather than relative values.

**Definition 4.14** *Let $G = (V, E, F)$ be a plane graph. An* absolute quasi-orthogonal shape *$S$ is a mapping from the set of faces $F$ to clockwise ordered lists of tuples $(e, b, a)$. The first and last entries are defined as in the definition of quasi-orthogonal shape. The second entry is a string over the alphabet $\{\leftarrow, \uparrow, \rightarrow, \downarrow\}$ denoting the segments of the edge. A $\leftarrow$ denotes a horizontal segment with decreasing x-coordinates, a $\rightarrow$ denotes a horizontal segment with increasing x-coordinate, A $\downarrow$ denotes a vertical segment with decreasing y-coordinates, a $\uparrow$ denotes a vertical segment with increasing y-coordinate. The quasi-orthogonal shape induced by an absolute quasi-orthogonal shape is denoted by $Q(S)$.*

From a quasi-orthogonal shape we can derive four absolute quasi-orthogonal shapes, each representing one rotation of the quasi-orthogonal shape. The above definition will help us to characterize the quasi-orthogonal shapes of a mixed upward planar graph for which a mixed upward drawing exists.

**Lemma 4.1** *Given a mixed upward planarization $G = (V, E_d \cup E_u, F)$ and a valid quasi-orthogonal shape $Q$ for $G$. There is a mixed upward orthogonal rectangle drawing $\Gamma$ of $G$ with quasi-orthogonal shape $Q$ if, and only if, there is an absolute quasi-orthogonal shape $S$ with $Q(S) = Q$ and for every edge $e$ in $E_d$ holds $\#_\downarrow S_b(e) = 0$ and $\#_\uparrow S_b(e) \geq 1$*

**Proof:** Let $\Gamma$ be a mixed upward orthogonal rectangle drawing with $Q(\Gamma) = Q$. Assume that $\#_\downarrow S_b(e) > 0$ for an edge $e \in E_d$. Then, according to the definition of $\downarrow$, $\Gamma$ contains a vertical segment with decreasing y-coordinates for a directed edge, which is a contradiction to the fact, that is a mixed upward drawing. Assume that $\#_\uparrow S_b(e) = 0$ for an edge $e \in E_d$. Then the edge $e$ has no upward pointing segment in $\Gamma$ which is a contradiction to the fact, that $\Gamma$ is a mixed upward drawing. □

### 4.6.2 Orthogonalization of the Upward Subgraph

In this section we describe an orthogonalization algorithm for upward planar graphs which centers hyperedges. We assume that the graph is connected and that the hypervertices have exactly one directed outgoing edge, a positive number of directed incoming edges, and no adjacent undirected edges. The output of the algorithm is an absolute quasi-orthogonal shape in which all incoming edges of a vertex connect to the same side of the vertex and all outgoing edges of a vertex connect to the same side. The sides for the incoming edges and outgoing edges are opposite to each other.

The algorithm works as follows: For each edge a tail- and a head-shape is calculated. The two shapes are concatenated yielding the shape for the edge. The algorithm picks one vertex at a time and assigns the head-shape to the incoming edges of the vertex and the tail-shape to the outgoing edges. According to the bend-or-end property of the `Kandinsky` model only one incoming edge and one outgoing edge may not bend. We choose the non-bending edge according to the following rules: If there are edges which connect to a hypervertex, choose the median edge according to the embedding of these edges. If there are no edges which connect to a hypervertex, choose the median edge according to the embedding if the number of edges is even, choose no non-bending edge if the number is odd. The assignment of the shapes is illustrated in Figure 4.18. The tail- and head-shapes can always be concatenated since the assignment of the shapes has the invariant that each tail-shape ends with a ↑ and each head-shape starts with a ↑. For the assignment of shapes at crossings we provide two alternative configurations which are symmetric. In both cases one edge bends while the other remains straight. We choose the configuration such that if there is one edge which connects to a hypervertex, this edge has no bends.

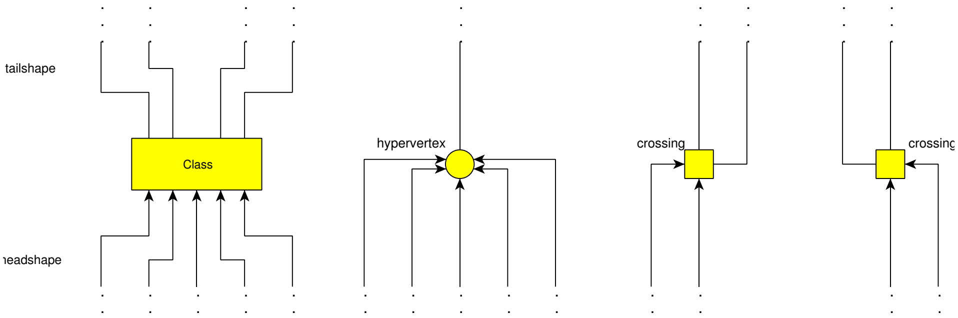


Figure 4.18: Shapes assigned to directed edges and hyperedges.

After this assignment we perform *bend stretching* transformations to reduce the number of bends. A bend stretching transformation removes superfluous bends from an edge by assigning a new shape with less bends to it without changing the first and last direction in the shape. Table 4.1 shows all bend-stretching transformations.

Figure 4.19 shows how the two bend stretching transformations 2 and 3 on the edge $(C1, C7)$ reduce the number of bends in the graph by 4.

**Lemma 4.2** *The above algorithm runs in linear time and the result has the following properties:*

1. *There is an upward orthogonal rectangle drawing of the input graph with quasi-orthogonal shape* $Q(S)$.
2. $Q(S)$ *is a* `Kandinsky` *shape.*
3. *On vertices of type* vertex *the edges leave the source vertex at the top and enter the target vertex on the bottom.*

Algorithm 7: `Upward-Orthogonalization`. We assume that the edges in $in(v)$ and $out(v)$ are ordered from left to right according to the mixed upward embedding.

```
Input: Upward planarization $G = (V, E, F)$,
       Mapping type : $V \to \{vertex, crossing, hypervertex\}$
Output: Absolute quasi-orthogonal shape $S$
// tail-shape
for $v \in V$ do
    $l = out(v)$, $m = median(l)$, $S_s(l_m) = " \uparrow "$
    if type$(v) = crossing$ then
        if $m = 0$ then $S_s(l_1) = " \rightarrow\uparrow "$
        else $S_s(l_0) = " \leftarrow\uparrow "$
    else
        for $0 \leq i < m$ do $S_s(l_i) = " \uparrow\leftarrow\uparrow "$
        for $m < i < |l|$ do $S_s(l_i) = " \uparrow\rightarrow\uparrow "$
// head-shape
for $v \in V$ do
    $l = in(v)$, $m = median(l)$
    if type$(v) = crossing$ then
        if $m = 0$ then $S_s(l_1) += " \leftarrow "$
        else $S_s(l_0) += " \rightarrow "$
    else
        if type$(v) = hypervertex$ then
            for $0 \leq i < m$ do $S_s(l_i) += " \rightarrow "$
            for $m < i < |l|$ do $S_s(l_i) += " \leftarrow "$
        else
            for $0 \leq i < m$ do $S_s(l_i) += " \rightarrow\uparrow "$
            for $m < i < |l|$ do $S_s(l_i) += " \leftarrow\uparrow "$
// Assign shapes to reverse edges
for $e \in E$ do  if $S_s(e) = \epsilon$ then $S_s(e) = S_s(\bar{e})$
// Assign angles
for $v \in V$ do
    for $0 \leq i < |out(v)| - 1$ do $S_a(out(v)_i) = 0$
    if $\delta^-(v) = 0$ then $S_a(out(v)_{|out(v)|-1}) = 4)$
    else $S_a(out(v)_{|out(v)-1|}) = 2$
    for $1 \leq i < |in(v)| - 1$ do $S_a(\overline{in(v)_i}) = 0$
    if $\delta^+(v) = 0$ then $S_a(\overline{in(v)_0}) = 4)$
    else $S_a(\overline{in(v)_0}) = 2$
```

| shape | $v$ is crossing | $w$ is crossing | new shape |
|---|---|---|---|
| ↑→↑→↑ | | | ↑→↑ |
| ↑→↑→ | | X | ↑→ |
| →↑→↑ | X | | →↑ |
| →↑→ | X | X | → |
| ↑←↑←↑ | | | ↑←↑ |
| ↑←↑← | | X | ↑← |
| ←↑←↑ | X | | ←↑ |
| ←↑← | X | X | ← |

Table 4.1: Table of bend-stretching transformations that apply to an edge $e = (v, w)$.

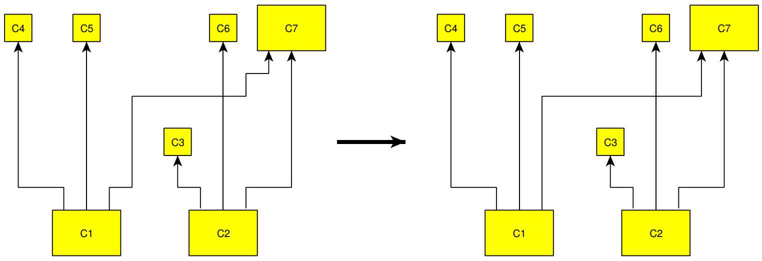


Figure 4.19: Example for a bend stretching transformation.

4. *Vertices of type* hypervertex *are centered.*

5. *The number of bends per edge is at most* $2c + 4$ *where* $c$ *is the number of crossings of the edge.*

**Proof:** First we show that $Q(S)$ is valid. For each upward planar graph there is an upward drawing where vertices are represented as horizontal straight lines and edges as vertical straight lines, called visibility representation, see, e.g., [33]. We denote the quasi-orthogonal shape of this drawing by $Q^V$. The quasi-orthogonal shape $Q(S)$ can be obtained from $Q^V$ by replacing the bend and angles fields in $Q^V$ by the configurations shown in Figure 4.18. The sum of the angles in a face is not changed by these replacements. Since $S$ fulfills the conditions of Lemma 4.1 the first property is proved.

The other properties follow directly from the assignment of the shapes. □

### 4.6.3 The Complete Algorithm

In this section we present the orthogonalization algorithm for UML class diagrams. The input of the algorithm is a mixed upward planarization of a mixed graph and a mapping assigning the vertices of the input graph to a type. We assume that the subgraph induced by the directed edges is connected. The algorithm consist of two phases: In the first phase it calculates the orthogonalization for the directed edges. This is done by removing temporarily all undirected edges from the graph and calculating an upward drawing of the remaining directed subgraph using algorithm `Upward-Orthogonalization`.

In the second phase the orthogonalization of the undirected edges is calculated while not changing the orthogonalization of the directed edges. This is done by creating constraints from the layout of the directed subgraph and then solving the corresponding CONSTRAINED KANDINSKY BEND MINIMIZATION problem. We extract from the shape of each directed edge the bends and create a bend constraint for it. Furthermore we create for each pair of directed edges following each other in the embedding on a vertex an angle constraint with the angle they form in the upward drawing. To all these constraints a cost $C$ is assigned which is an upper bound on the number of bends in orthogonalization of the entire graph in which all constraints are fulfilled.

We use a modified version of the approximation algorithm from Section 4.4 to solve the corresponding CONSTRAINED KANDINSKY BEND MINIMIZATION network. Note that no `Kandinsky` device of a directed edge can be over-saturated in an optimal solution of the minimum cost flow relaxation, otherwise it would have cost greater than $C$. If we apply a transformation to a sublist with an over-saturated `Kandinsky` device, we first check if in one of the two sides of the list are `Kandinsky` devices which are defined by directed edges. In this case we apply a transformation which does not change this side. Note that it is not possible that such devices are on both sides, otherwise the angle-constraints would be violated. Since we may be forced in this modified algorithm to apply a transformation on the longer side of the sub-list we no longer can guarantee an approximation factor of 2, but since each side is still shorter than the entire sub-list we get an approximation factor of 3. The above discussion is summarized in algorithm `Mixed-Upward-Orthogonalization`.

**Theorem 4.8** *Given a mixed-upward planarization $G = (V, E_d \cup E_u, F)$, and a map* `type`*, algorithm* `Mixed-Upward-Orthogonalization` *computes in time $O(n^{\frac{7}{4}} \cdot \sqrt{\log n})$ an absolute* `Kandinsky` *shape $S$ with the following properties:*

1. *There is an upward orthogonal rectangle drawing of $G$ with quasi-orthogonal shape $Q(S)$.*

```
Algorithm 8: Mixed-Upward-Orthogonalization.
Input: Mixed upward planarization $G = (V, E, F)$,
       Mapping type : $V \rightarrow \{vertex, crossing, hypervertex\}$
Output: absolute Kandinsky shape $S$
Remove all directed edges temporarly
$Q^D$ = Upward-Orthogonalization($G$)
Reinsert temporarily removed edges
Construct set of angle constraints $AC$ from $Q^D$
Construct set of bend constraints $BC$ from $Q^D$
$Q$ = Constrained-Kandinsky($G, AC, BC$)
Compute $S$ from $Q$
```

2. *On vertices of type* vertex *the edges in* $E_d$ *leave the source vertex at the top and enter the target vertex on the bottom.*

3. *Vertices of type* hypervertex *are centered.*

4. *The number of bends in a directed edge is at most* $2c + 4$ *where* $c$ *is the number of crossings of the edge.*

5. *The number of bends on undirected edges is not more than 3 times the minimal number of bends in a* `Kandinsky` *shape with the same quasi-orthogonal shape for the directed edges.*

# Chapter 5

# Compaction

In this chapter we consider the problem of computing a prescribed vertex-size drawing from a given absolute `Kandinsky` shape, including the placement of labels, while minimizing the sum of the length of all edges and the area used by the drawing. Since it is NP-hard to optimize one of the above criteria [99] we will present a heuristic for solving his problem. We start with giving a formal definition of the problem:

**Definition 5.1** *Let $G = (V, E, F)$ be a plane graph with absolute* `Kandinsky` *shape $Q$, $L$ a set of labels with reference mapping $r : L \to E$, and $s : V \cup L \to \mathbb{N}^2$ denote the size of vertices and labels. A drawing $\Gamma$ is called* valid compaction *if*

1. $\Gamma$ *is an orthogonal rectangle drawing*
2. $Q(\Gamma) = Q$,
3. $\Gamma(o)$ *is a rectangle of size* $s(o)$ *for* $o \in V \cup L$,
4. $\Gamma(l) \cap \Gamma(r(l)) \neq \emptyset$ *is a part of the boundary of* $\Gamma(l)$ *for* $l \in L$.

*The* MINIMUM EDGE LENGTH KANDINSKY COMPACTION *problem is to find a valid compaction in which the sum of the edge length is minimal.*

We propose a two-step algorithm for the MINIMUM EDGE LENGTH KANDINSKY COMPACTION problem. In the first step a low quality valid compaction, called the *start solution*, is calculated which is then improved in the second step by a post-processing algorithm. Our approach is motivated by empirical tests comparing compaction algorithms for 4-graphs in the orthogonal point-drawing model [82]. In these tests the above approach, with using one-dimensional compaction as post-processing, has yield nearly optimal solutions. Interestingly the quality of the start solution had no significant impact on the final result. Therefore the most important property for the algorithm which computes the start solution is to be fast, the quality of the solution is not of primary concern. However, until yet the fastest algorithm for this problem has at least quadratic runtime. The main result presented in this chapter is a linear time algorithm for providing a start solution. Our algorithm combines two compaction methods for orthogonal shapes and generalizes them to `Kandinsky` shapes. The insight that we gain in combining these two algorithms is a result on its own and may lead as a starting point to develop new compaction algorithms.

This chapter partly follows [49] and is organized as follows: In Section 5.1 we review previous work for compaction and label placement. Section 5.2 contains an overview over our compaction algorithm and describes how we can reduce the MINIMUM EDGE LENGTH KANDINSKY COMPACTION problem to the problem of finding a valid compaction of an absolute `Kandinsky` shape without considering labels. The shape graph approach, which will lead us to a linear time algorithm for finding valid compactions, is presented in Section 5.3, while the algorithm itself is presented in the main Section 5.4.

## 5.1 Previous Work

In this section we review algorithms for compaction and label placement. We first discuss compaction of orthogonal shapes before we turn our attention to the compaction of `Kandinsky` shapes. Finally we discuss label placement and its connection to compaction.

### 5.1.1 Compaction of Orthogonal Shapes

The first approach for the compaction of orthogonal shapes was rectangular decomposition [115]. The starting point for the rectangular decomposition strategy is the observation that if all faces in an orthogonal shape are rectangles, we can easily solve the compaction problem by applying longest path or network flow algorithms to it. Rectangle Decomposition exploits this observation and subdivides those faces which are not rectangular into rectangles. Then it solves the compaction problem on the subdivided orthogonal shape. This induces a valid compaction on the original orthogonal shape. The subdivision of complex faces into rectangular faces can be performed efficiently by searching certain patterns of angles on the face. We denote 90° angles on a face with a 0 and 270° angles with a 1. Every time we find the pattern 100, we cut a rectangle from the face and continue to search the pattern on the remaining face. See Figure 5.1 for an illustration. We terminate if there are no more patterns in any face. Using a list, rectangle decomposition can be done in linear time.

The rectangle decomposition method leads often to drawings which are not satisfactory since the decomposition chosen by the algorithm leads often to suboptimal drawings, even if better solutions are obvious for the user. Therefore two new approaches have been developed which do not rely on

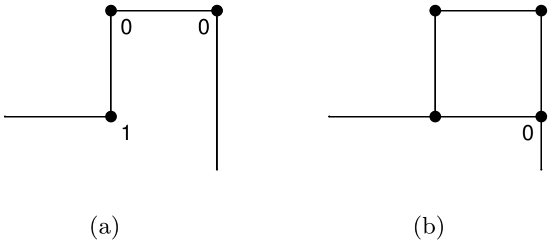


Figure 5.1: The decomposition of a face (a) into a rectangular face and a remaining face (b).

decomposing the faces into rectangles: the turn-regularity approach [20] and the shape graph approach [85]. With the help of both approaches we can describe the set of orthogonal shapes, for which optimal compactions can be calculated in polynomial time, and actually compute them. We will give a detailed description of the shape graph approach in Section 5.3.

### 5.1.2 Compaction in the `Kandinsky` Model

The first compaction algorithm for `Kandinsky` shapes was presented in [56]. It is based on the rectangle decomposition method and produces drawings in which all vertices have uniform size. The algorithm has been extended to drawings in which the size of the vertices is determined by their degree in [57]. A detailed description of these algorithms can be found in [54].

Di Battista et. al. [32] presented the first compaction algorithm which creates drawings with prescribed vertex size. It creates in a first step a drawing of the graph where the vertices are represented as points and edges overlap if they are adjacent to the same vertex at the same side using the original compaction algorithm described above. Then each vertical and each horizontal grid-line is expanded individually, i.e., each grid-line is replaced by a set of grid-lines such that the vertices can be assigned their prescribed sizes and the edges can be routed without overlap. This expansion is done by solving a minimum cost flow problem for each horizontal and each vertical grid-line. The authors do not give any bounds on the time complexity of their algorithm. Since the algorithm is based on minimum cost flow computation it has at least quadratic running time. The authors experienced up to 50 seconds computation time on graphs in the Rome graphs test suite [34].

Klau et al. suggested in [84] that their compaction approach originating from graph labeling can also be used to solve the compaction problem for drawings with prescribed vertex size. The approach relies on branch-and-cut and has, therefore, exponential worst case running time. The authors give neither a detailed description of how the algorithm can be used in the prescribed vertex size setting nor experimental results.

### 5.1.3 Label Placement

The work of Klau and Mutzel [84] was the first one integrating compaction and label placement into one algorithm. Before the two problems have been treated individually. The usual approach was to apply map labeling techniques on the compacted drawing. This approach has the disadvantage that it cannot be guaranteed that all the labels can be placed without overlaps, especially if the labels have non-negligible size, which is often the case in class diagrams. The algorithm described in [84] is designed for vertex labels. It has been extended to edge labels in [10]. Both algorithms calculate optimal solution using integer linear programming techniques, which leads to exponential worst case running time since the problem is NP-hard. Since we require algorithms which can be used in an interactive setting, this makes it prohibitive for us to apply these algorithms.

In [9] various heuristics for integrating edge labeling into compaction have been proposed. They all base on the idea of inserting the edge labels one by one into the compaction. If a label cannot be inserted without overlap, the compaction is changed accordingly so that the label can be included in the drawing. Their approach works on point drawings of 4-graphs and is based on slicing to expand the drawing if a label cannot be placed. Slicing is a technique developed originally in VLSI design, see [89] for details. It is not trivial to generalize their algorithm to rectangle drawings with vertices of prescribed size since slicing is much more difficult in this model. Slicing curves must not intersect rectangles in this model because this would result in changing the size of vertices which is not allowed.

## 5.2 The Compaction Algorithm

In this section we outline our algorithm, for the MINIMUM EDGE LENGTH KANDINSKY COMPACTION problem. The algorithm first removes in a preprocessing step all labels from the input and then computes a low quality valid compaction, called the start solution. This start solution is improved by applying iteratively an one-dimensional compaction algorithm. Finally the labels are placed. Our approach is summarized by algorithm *Compaction.*

We call an orthogonal shape *simple* if it has no bends. Note that every orthogonal shape can be reduced to a simple orthogonal shape by replacing the bends by dummy vertices. We assume for the rest of this chapter that the input is simple. We denote the set of vertices in a simple shape which have been inserted to represent bends with $B$. Note that we distinguish between vertex-bends and face-bends in `Kandinsky` drawings. Each vertex-bend $b$ can be uniquely assigned to a zero degree-angle at a vertex $v$. In a simple `Kandinsky` shape, $b$ and $v$ are connected by two darts $(v, b)$ and $(b, v)$. We define a function *vertexbend* : $E \longrightarrow \{\texttt{true}, \texttt{false}\}$ which returns `true` if a dart connects a vertex-bend to the vertex at the zero-degree angle to which it is assigned, or `false` otherwise.

```
Algorithm 9: Compaction
Input: Valid Kandinsky shape $Q$, size function $s$,
       labels $L$, refer function $r$.
Output: Valid compaction $\Gamma$
Preprocess labels $L$
Calculate start solution $\Gamma$ of $Q$ and $s$
Perform one-dimensional compaction on $\Gamma$
Postprocess labels $L$
return $\Gamma$
```

The computation of the start solution is the subject of the subsequent sections, in the remainder of this section we will first discuss label placement and then one-dimensional compaction.

### 5.2.1 Label Placement

We will use a simple method for label placement which is based on the reduction approach. Our method has no negative effect on the running time and yields sufficient results for our case. We assume that edge labels with preferred placement `source` or `target` have been removed from the input and that they are placed by a map labeling algorithm after compaction. This approach works for class diagrams since these labels are fairly small, i.e., multiplicity labels have the form "$1..n$" or "$*$". Therefore $L$ contains only labels with preferred placement `center`. Before the compaction phase, the edge, to which the label is attached, is a path of segments. We choose a segment in the middle and split it in two parts by introducing a dummy vertex. We take care that the edge which is split is not an edge defining a zero degree angle, therefore *vertexbend* must evaluate to `false` for this edge, otherwise $Q$ would be no longer a `Kandinsky` shape. We assign to the dummy vertex the size of the label. Then after the compaction the labels are placed on the position of the corresponding dummy vertex and the dummy vertex is removed from the graph. See Figure 5.2 for an example.

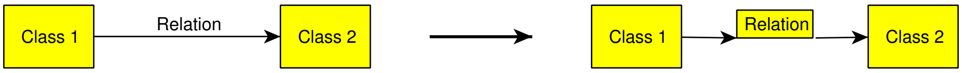


Figure 5.2: Labels with preferred placement `center` are treated like vertices in the compaction.

### 5.2.2 One-Dimensional Compaction

One-dimensional compaction is a technique known from VLSI layout which aims to reduce area and total edge length of a drawing. Either path based or flow based one-dimensional compaction can be used. The path based algorithm has linear running time, whereas the flow based algorithm has quadratic running time. However, the flow based algorithm produces better results than the path based algorithm. We will not cover one-dimensional compaction in this work, for an extensive overview see for example [89]. Empirical tests on 4-graphs [82] have shown that using a heuristic for compaction together with flow based one-dimensional compaction yields nearly optimal solutions.

## 5.3 The Shape Graph Approach

In this section we discuss the shape graph approach for the optimal compaction of orthogonal shapes which was originally proposed by Klau and Mutzel [85]. We will first review the basics of the approach and then derive a linear time algorithm for the compaction of orthogonal shapes from it.

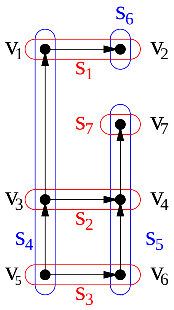
Given a plane graph $G$ with absolute orthogonal shape $H$. We denote with $G_\rightarrow = (V, E_\rightarrow)$ the subgraph of $G$ which contains only darts pointing in positive direction of the x-axis, i.e., $E_\rightarrow = \{e \in E | H_d(e) = \rightarrow\}$ and with $G_\uparrow = (V, E_\uparrow)$ the subgraph of $G$ which contains only darts pointing in positive direction of the y-axis, i. e., $E_\uparrow = \{e \in E | H_d(e) = \uparrow\}$.

The connected components of $G_\rightarrow$, resp. $G_\uparrow$, are directed paths and form a line in a drawing of $G$ with orthogonal shape $H$. We denote with $S_\rightarrow$, resp. $S_\uparrow$, the set of connected components of $G_\rightarrow$, resp. $G_\uparrow$, and call the elements of it *horizontal segments*, resp. *vertical segments*. For a segment $s$ the direction $d(s)$ is $\rightarrow$, resp. $\uparrow$, if the segment is horizontal, resp. vertical. We say that two segments are *adjacent* if they share a point. We denote with $\alpha(s)$ the start vertex of the path which forms a segment $s$ and with $\omega(s)$, the endpoint of this path. We further denote with $E_s$ the edges in a segment and with $V_s$ the vertices. Every edge $e$ is contained in exactly one segment $seg(e)$ and each vertex $v$ is contained in exactly one vertical segment $vert(v)$ and one horizontal segment $hor(v)$. We define $seg(v, d) : V \times \{\rightarrow, \uparrow\}$ as

$$seg(v,d) = \begin{cases} hor(v) & \text{if } d \in \{\rightarrow, \leftarrow\} \\ vert(v) & \text{if } d \in \{\uparrow, \downarrow\}. \end{cases}$$

The ordering of the segments of one type is defined by a *shape graph*. The edge sets of the shape graphs $D_\uparrow = \{S_\rightarrow, A_\uparrow\}$, resp. $D_\rightarrow = \{S_\uparrow, A_\rightarrow\}$, are defined as:

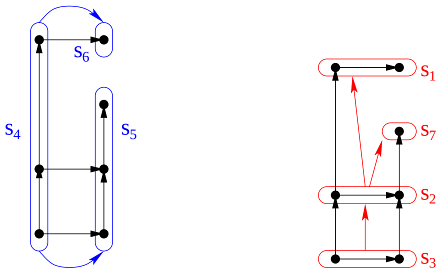
$$\begin{aligned} A_\uparrow &= \{(hor(v), hor(w)) : (v,w) \in E_\uparrow\}, \text{ and} \\ A_\rightarrow &= \{(vert(v), vert(w)) : (v,w) \in E_\rightarrow\} \end{aligned}$$



| Segment | Nodes | Edges |
|---|---|---|
| $s_1$ | $\{v_1, v_2\}$ | $\{(v_1, v_2)\}$ |
| $s_2$ | $\{v_3, v_4\}$ | $\{(v_3, v_4)\}$ |
| $s_3$ | $\{v_5, v_6\}$ | $\{(v_5, v_6)\}$ |
| $s_4$ | $\{v_1, v_3, v_5\}$ | $\{(v_1, v_3), (v_3, v_5)\}$ |
| $s_5$ | $\{v_4, v_6, v_7\}$ | $\{(v_4, v_6), (v_4, v_7)\}$ |
| $s_6$ | $\{v_2\}$ | $\emptyset$ |
| $s_7$ | $\{v_7\}$ | $\emptyset$ |

Figure 5.3: Examples of horizontal and vertical segments.

The *shape description* $\mathcal{S} = (D_{\rightarrow}, D_{\uparrow})$ combines two shape graphs.



(a) (b)

Figure 5.4: Shape graph $D_{\rightarrow}$ (a) and $D_{\uparrow}$ (b) of graph in Figure 5.3.

We define the linear program $(LP)$ as:

$$\begin{array}{rcl} x_b - x_a & \geq & 1 \;\; \forall (a,b) \in A_{\rightarrow} \\ y_b - y_a & \geq & 1 \;\; \forall (a,b) \in A_{\uparrow} \\ x_s & \geq & 0 \;\; \forall s \in S_{\uparrow} \\ y_s & \geq & 0 \;\; \forall s \in S_{\rightarrow} \end{array}$$

We can find a feasible solution $(x, y)$ of $(LP)$ in linear time by solving a longest path problem [89]. From the solution $(x, y)$ we can construct a drawing $\Gamma : V \rightarrow \mathbb{N} \times \mathbb{N}$ from $G$ with $\Gamma(v) = (x_{vert(v)}, y_{hor(v)})$. However, there may be feasible solutions of the $(LP)$ which induce non-valid orthogonal drawings.

**Definition 5.2** *Let $s_{\rightarrow}$ be a vertical segment and $s_{\uparrow}$ be a horizontal segment which are not adjacent. We call $s_{\rightarrow}$ and $s_{\uparrow}$ to be* separated *if one of the following conditions hold:*

$$\begin{array}{ll} \textit{1. } s_{\uparrow} \xrightarrow{*}_{D_{\rightarrow}} vert(\alpha(s_{\rightarrow})) & \textit{2. } vert(\omega(s_{\rightarrow})) \xrightarrow{*}_{D_{\rightarrow}} s_{\uparrow} \\ \textit{3. } s_{\rightarrow} \xrightarrow{*}_{D_{\uparrow}} hor(\alpha(s_{\uparrow})) & \textit{4. } hor(\omega(s_{\uparrow})) \xrightarrow{*}_{D_{\uparrow}} s_{\rightarrow} \end{array}$$

A shape description is called *complete* if every pair of segments with opposite direction is separated.

In Figure 5.4 the segments $s_4$ and $s_7$ are separated, while the segments $s_6$ and $s_5$ are not separated. Adding the edge $(s_7, s_1)$ into $A_{\uparrow}$ makes the shape description complete.

**Lemma 5.1 ([85])** *If $\mathcal{S}$ is complete a feasible solution $(x, y)$ of the linear program (LP) induces an orthogonal drawing $\Gamma : V \to I\!N \times I\!N$ with $\Gamma(v) = (x_{vert(v)}, y_{hor(v)})$ of $G$ with orthogonal shape $H$.*

In [85] it is shown that there always exists a superset of $\mathcal{S}$ which is a complete shape description. We call such a superset $\bar{\mathcal{S}}$ a *complete extension* of $\mathcal{S}$. In the original work the authors propose a branch and cut algorithm which finds the complete extension such that the resulting drawing has minimal edge length [85]. This approach has the disadvantage that it has exponential worst-case running time. We cannot hope to do better when we try to optimize the edge length, since this problem is NP-hard [100].

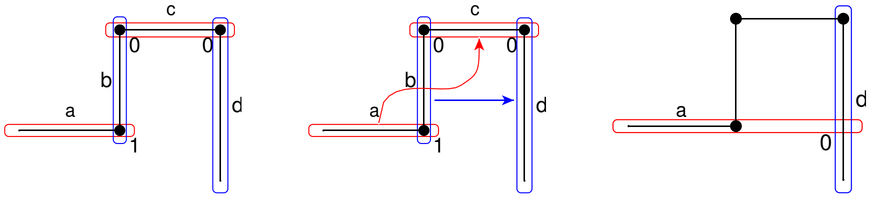
If we drop the goal to achieve optimality, we can get a much faster algorithm by searching a complete shape extension by a heuristic. We use rectangular decomposition [115] to find such a complete shape extension.

For each face we perform the following: First we build a list containing all segments of the face in the order as they occur in it. In addition we store for each segment, as in the original rectangular decomposition algorithm, the angle which the segment forms with its predecessor in the list. Then we put the elements of the list one-by-one on the stack. Every time a '100' pattern of angles is at the top of the stack we perform decomposition as shown in Figure 5.5. Instead of introducing a dummy vertex and a dummy edge in the graph, as in the original rectangle decomposition algorithm, we add edges to the shape graphs. In the example in Figure 5.5, we insert the edge $(a, c)$ into $D_{\uparrow}$ and the edge $(b, d)$ into $D_{\rightarrow}$.

We handle the other three cases symmetrically . They are described in function `define-box`. After the insertion of the edges into the constraint graphs, we remove the second and third top-most elements. We proceed until we can no longer find a 100 pattern.

**Lemma 5.2** *Rectangle decomposition yields a complete shape extension of size $O(n)$ in linear time.*

**Proof:** We first show that the complete shape extension has size $O(n)$. The initial shape description has linear size by Euler's formula. Since the rectangle decomposition introduces $O(n)$ rectangles and we insert two edges into it per rectangle, the complete shape extension has linear size. The



(a) (b) (c)

Figure 5.5: Rectangle decomposition revisited in the shape graph approach. Figure (a) shows a 100 pattern. In Figure (b) the inserted edges in the constraint graph are shown. Figure (c) shows the subdivided face.

---

Algorithm 10: `define-box`

```
Input: Shape description $\mathcal{S} = ((S_\uparrow, A_\rightarrow), (S_\rightarrow, A_\uparrow))$, Direction $d$,
       Segments $a, b, c, d$
if $d = \uparrow$ then $A_\uparrow := A_\uparrow \cup (b, d)$, $A_\rightarrow := A_\rightarrow \cup (c, a)$;
if $d = \downarrow$ then $A_\uparrow := A_\uparrow \cup (d, b)$, $A_\rightarrow := A_\rightarrow \cup (a, c)$;
if $d = \leftarrow$ then $A_\uparrow := A_\uparrow \cup (a, c)$, $A_\rightarrow := A_\rightarrow \cup (b, d)$;
if $d = \rightarrow$ then $A_\uparrow := A_\uparrow \cup (c, a)$, $A_\rightarrow := A_\rightarrow \cup (d, b)$;
```

---

linear running time follows immediately from this fact, too. It remains to be shown that the extension is complete. Take a drawing of $G$ produced by the conventional rectangle decomposition algorithm and take a vertical segment $s_\rightarrow$ and a horizontal segment $s_\uparrow$ which are not adjacent. Because the drawing is planar, $s_\rightarrow$ and $s_\uparrow$ do not cross, so one of the four following cases must hold: $s_\rightarrow$ is above $s_\uparrow$, $s_\rightarrow$ is below $s_\uparrow$, $s_\rightarrow$ is left of $s_\uparrow$ or $s_\rightarrow$ is right of $s_\uparrow$. Assume w.l.o.g. that $s_\uparrow$ is above $s_\rightarrow$ and that $s_\uparrow$ is not to the left of $s_\rightarrow$. The other cases are symmetric. Since $s_\uparrow$ is above $s_\rightarrow$, $s' = hor(\alpha(s_\uparrow))$ is also above $s_\rightarrow$. Assume that one of the following cases holds:

1. There is a $s_v \in S_\uparrow$ such that the intersection of the projection of $s_v$ and $s_\uparrow$ on the y-axis is non-empty and there is a path from $vert(\omega(s_\rightarrow))$ to $s_\uparrow$ in $D_\rightarrow$.

2. There is a segment $s_h \in S_\rightarrow$ such that the intersection of the projection of $s_h$ and $s'$ on the x-axis is non-empty and there is a path from $s_\rightarrow$ to $s_h$ in $D_\uparrow$.

If the assumption is true, we are done. To see this, assume that the second case holds. Then just take a line parallel to the y-axis with x-coordinate in the intersection of the projections. From the rectangles which are intersected by the parallel line, we can now easily construct a path from $s_h$ to $s'$ in $D_\uparrow$.

We give a constructive proof that either $s_v$ or $s_h$ exists. Start at segment $s_\rightarrow$ and go to the lowest rectangle to the right of it, if such a rectangle exists. In this case, go from this rectangle to the leftmost rectangle above. Iterate until a segment is found which induces an intersection of the projections. Because we proceed monotonically increasing in x- and y-coordinates, such a segment must exist for monotonicity reasons. The existence of the path follows from how we traverse the rectangles. □

## 5.4 A Linear Time Compaction Algorithm

To compute a valid compaction of a `Kandinsky` shape $Q$ with vertices of prescribed size we map $Q$ to an orthogonal shape $H_Q$ which represents each vertex with non-zero size in $Q$ by a rectangular face. We call $H_Q$ the *compaction shape* of $Q$. A drawing of $H_Q$ induces a valid drawing of $Q$ if the rectangular shapes have exactly the size of the vertices which they represent. We will characterize the complete shape extensions of $H_Q$ which have such a drawing in the first section and call them *length complete*. To compute a length complete shape extension of $H_Q$ we will use a complete shape extension $Q$. In the second section we will introduce complete shape extensions for `Kandinsky` shapes and show in the third section how we can use them to derive length complete shape extensions of compaction shapes. In the fourth section we present a linear time algorithm to compute from a length complete shape extension of $H_Q$ a valid drawing of $H_Q$. The last section covers extensions of the basic algorithm. Our approach is summarized in the following algorithm:

```
Algorithm 11: Compute Start Solution
Input: Absolute Kandinsky shape $Q$, size function $s$.
Output: Valid Compaction $\Gamma$ of $Q$.
Compute compaction shape $H_Q$ from $Q$
Compute shape description $\mathcal{S}^Q$ of $Q$
Compute complete shape extension of $\bar{\mathcal{S}}^Q$
Compute shape description $\mathcal{S}$ of $H_Q$
Compute length-complete shape extension $\bar{\mathcal{S}}$ of $\mathcal{S}$ from $\bar{\mathcal{S}}^Q$
Assign coordinates to vertices in $H_Q$
Derive drawing $\Gamma$ of $Q$ from $H_Q$
```

### 5.4.1 Compaction Shape

The compaction shape $H_Q$ of a `Kandinsky` shape $Q$ is an orthogonal shape in which each vertex of $Q$ which has non-zero size is represented by a rectangular face. Let $\hat{V} \subset V$ denote the set of vertices which have zero size, in other words $\hat{V} = \{v \in V | s(v) = (0,0)\}$. Let $v$ be a vertex in $V \setminus \hat{V}$. For each edge adjacent to $v$, a new vertex $p(e,v)$ is created which represents the port of $e$ on $v$. Also, four corner vertices $nw(v)$, $ne(v)$, $sw(v)$ and $se(v)$ are created. See Figure 5.6 for an example. Each vertex face has four adjacent *vertex-segments*: the top segment $t(v)$, the bottom segment $b(v)$, the left segment $l(v)$, and the right segment $r(v)$. We denote the underlying graph of a compaction shape $H_Q$ with $G_Q = (V_Q, E_Q)$. The mapping $simple : E \longrightarrow E_Q$ maps every edge in $E$ to the corresponding edge in $E_Q$. $G_Q$ has $4|V \setminus \hat{V}| + 2|E| + |\hat{V}|$ vertices. Since $G$ and $G_Q$ are planar it follows with Euler‘s formula that the above transformation causes a constant blow up.

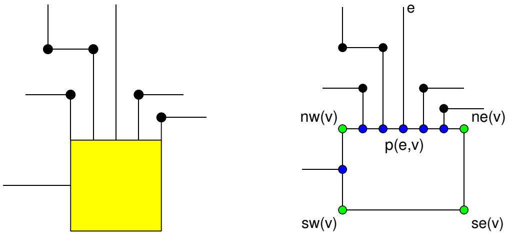


Figure 5.6: Transformation of a vertex in the input graph to a rectangular face in the compaction graph.

A drawing of $H_Q$ which induces a valid drawing on $Q$ is called a *length valid drawing* of $H_Q$. There are two differences between a valid and a length valid drawings of $H_Q$.

1. The edges adjacent to a corner vertex may have zero length in a length valid drawing.

2. The segments of the vertex-faces have prescribed distance in a length valid drawing.

We refine the shape graph definition of the previous section in order to be able to express these requirements. We add a length function to the shape graphs and introduce auxiliary edges which denote the vertex size. These refinings are similar to the ones in [84]. We define the edge-set of the shape graph $D'_\uparrow$ as $A'_\uparrow = A_\uparrow \cup N^+_\uparrow \cup N^-_\uparrow$, with

$$N^+_\uparrow = \{(b(v), t(v)) : v \in V\}, \text{ and } N^-_\uparrow = \{(t(v), b(v)) : v \in V\}$$

The shape graph $D'_\rightarrow$ is defined analogously, and $\mathcal{S}' = (D'_\rightarrow, D'_\uparrow)$ is the corresponding shape description. Additionally, we define the length function

$length : A'_\uparrow \cup A'_\rightarrow \to \mathbb{Z}$ in the following way:

$$length(e) = \begin{cases} 0 & \text{if } e \in A_\uparrow, \text{e is induced by an edge adj. to a corner} \\ s_x(v) & \text{if } e \in N_\uparrow^+ \\ -s_x(v) & \text{if } e \in N_\uparrow^- \\ 1 & \text{otherwise} \end{cases}$$

The values of *length* for $A'_\rightarrow$ are defined analogously. This leads to the `Kandinsky` linear program $(KLP)$:

$$\begin{aligned} x_b - x_a &\geq length(e) \;\; \forall e = (a,b) \in A'_\rightarrow \\ y_b - y_a &\geq length(e) \;\; \forall e = (a,b) \in A'_\uparrow \\ x_s &\geq 0 \;\; \forall s \in S_\uparrow \\ y_s &\geq 0 \;\; \forall s \in S_\rightarrow \end{aligned}$$

To characterize those shape extensions whose solution of $(KLP)$ induce length valid drawings, it is not enough to demand that all segments have to be separated. The reason for this is that the complete shape extension may contain edges which induce a positive length cycle which violates the maximum length condition for vertex faces, see Figure 5.7 for an example. We have to introduce therefore an additional condition.

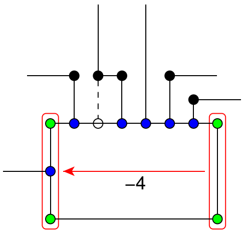


Figure 5.7: Rectangle Decomposition of $H_Q$ which does not lead to a length complete shape extension.

**Definition 5.3** *A shape extension $\bar{\mathcal{S}}$ is* length-complete *if $\bar{\mathcal{S}}$ is complete and every cycle in the shape graphs of $\bar{\mathcal{S}}$ has non-positive length.*

The following lemma shows that length-completeness describes indeed what we wanted.

**Lemma 5.3** *Let $\bar{\mathcal{S}}$ be a complete shape extension. $(KLP)$ has a feasible solution if, and only if, every cycle in the shape graphs of $\bar{\mathcal{S}}$ has non-positive length.*

**Proof:** It is shown in [85] that if a shape graph in $\mathcal{S}$ has a positive length cycle, then $(KLP)$ is not feasible. We can transform $(KLP)$ to a *system of* *difference constraints* by multiplying each constraint by $-1$. This transforms every positive length cycle in a negative length cycle and vice versa. A system of difference constraints is feasible if and only if it has no negative length cycle, see, i.e., [28] for a proof. □

### 5.4.2 Complete Shape Extensions of `Kandinsky` Shapes

In this section we will refine the concept of shape graph from orthogonal shapes to `Kandinsky` shapes and define complete shape extensions for this case. We will present an algorithm which computes a complete shape extension in linear time.

In a `Kandinsky` shape there may be more than one edge adjacent to a side of a vertex. As a consequence the segments in a `Kandinsky` shape are no longer directed paths as this is the case in an orthogonal shape. Segments in `Kandinsky` shapes have a more complex structure, see Figure 5.8 for an example. Therefore the values $\alpha$ and $\omega$ are no longer well defined for a segment of a `Kandinsky` shape. We define the *skeleton* $sk(s)$ of a segment $s$ as $sk(s) = \{e \in E_s | vertexbend(e) = \texttt{false}\}$. It follows directly from the bend-or-end property of `Kandinsky` shapes that $sk(s)$ is a directed path. With the help of $sk(s)$ we can give a new definition of $\alpha$ and $\omega$: we define $\alpha(s)$ of a segment $s$ of a `Kandinsky` shape as the first vertex of $sk(s)$ and $\omega(s)$ as the last vertex of $sk(s)$.

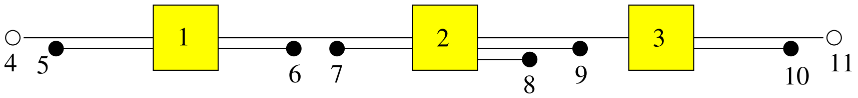


Figure 5.8: A segment in a `Kandinsky` shape. The vertices in $B$ denoting a vertex-bend are represented by black circles, vertices in $B$ denoting a face-bend are represented by white circles and the remaining vertices are represented as yellow rectangles. The $\alpha$ value of the segment is 1 and the $\omega$ value is 11.

Let $s \in S_{\rightarrow}$ be a horizontal segment. We define

$$W_\alpha(s) = \{s' \in S_\uparrow | \alpha(s') \in B \text{ and } hor(\alpha(s')) = s\},$$

$$W_\omega(s) = \{s' \in S_\uparrow | \omega(s') \in B \text{ and } hor(\omega(s')) = s\}.$$

The set $W_\alpha(s)$, resp. $W_\omega(s)$, denotes the set of vertical segments adjacent to $s$ on a bend which are above, resp. below, $s$. We define $W_\alpha(s)$ and $W_\omega(s)$ analogously for vertical segments.

In a `Kandinsky` drawing, the coordinates of the segments in $W_\alpha$ and $W_\omega$ must be in a specific absolute order to avoid overlap of adjacent segments. The segments in Figure 5.8 correspond to this ordering. Figure 5.9 shows what happens if this ordering is violated:

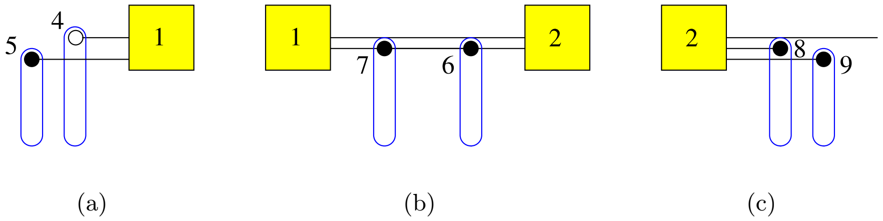


Figure 5.9: Three examples for invalid orderings of the segments in $W_\omega$.

We will give now a definition of this ordering, which we denote with $\lhd$. We denote the skeleton of $s$ as $sk(s) = u_1, e_1, u_2, \ldots, e_{k-1}, u_k$. Let $s'$ be a segment in $W_\alpha(s)$, then there is an edge $e \in E_s$ such that $e$ is incident to $u_i$ and $\alpha(s')$ for $1 \leq i \leq k$. We call $i$ the index of $s'$ or shorter $i(s')$ and denote the edge with $w(s) = e$. The index of a segment in $W_\omega(s)$ is defined similarly. The ordering $\lhd$ must fulfill the following properties:

1. The $\alpha$ vertex of the segment has the smallest coordinate value: $\forall s_1 \in W_\alpha(s) \cup W_\omega(s) : \mathit{seg}(\alpha(s), d) \lhd s_1$
2. The $\omega$ vertex of the segment has the greatest coordinate value: $\forall s_1 \in W_\alpha(s) \cup W_\omega(s) : s_1 \lhd \mathit{seg}(\omega(s), d)$
3. All segments adjacent to bends at a vertex $u_i$ have smaller coordinate value than the segments adjacent to bends with a greater index: $\forall s_1, s_2 \in W_\alpha(s) \cup W_\omega(s) : i(s_1) < i(s_2) \Rightarrow s_1 \lhd s_2$.
4. For two segments $s_1$,$s_2 \in W_\alpha$ with index $i$, $s_1 \lhd s_2$ if there is no skeleton edge between $w(s_1)$ and $w(s_2)$ in $\mathcal{E}(u_i)$.
5. For two segments $s_1$,$s_2 \in W_\omega$ with index $i$, $s_1 \lhd s_2$ if there is no skeleton edge between $w(s_2)$ and $w(s_1)$ in $\mathcal{E}(u_i)$.

We call a shape extension *segment-separated* if the relations $\xrightarrow{*}_{D_\rightarrow}$ and $\xrightarrow{*}_{D_\uparrow}$ induce an ordering $\lhd$.

We employ an idea of the original compaction algorithm of Fößmeier [54] to compute a segment-separated shape extension of $Q$ and use a modified version of rectangle decomposition. As in the usual rectangle decomposition method we consider one face at the time and assign a marker to each angle in the face. We enhance the usual markers with 'K' which denotes a 270° vertex-bend and '-1' which denotes a zero degree-angle. We search for configurations as shown in Figure 5.10. These configurations correspond to '-1K' and 'K-1' patterns on the stack. When we detect one pattern, we remove the appropriate segments from the stack as in the previous section. Contrary to the previous section we add only one edge to the shape graph, since the second edge would be an selfloop. We stop when there are no more -1 markers in the face.

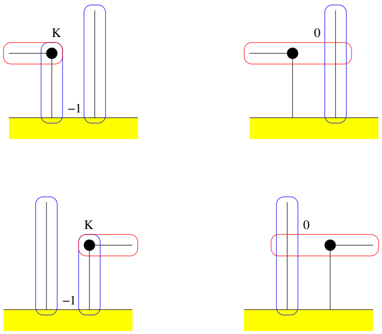


Figure 5.10: Rules to eliminate zero degree angles. The first row shows the application of pattern '-1K' and the second row the application of pattern 'K-1'.

**Definition 5.4** *We call a shape extension of a* `Kandinsky` *shape* complete *if it is segment-separated and each pair of non-adjacent segments with opposite direction is separated.*

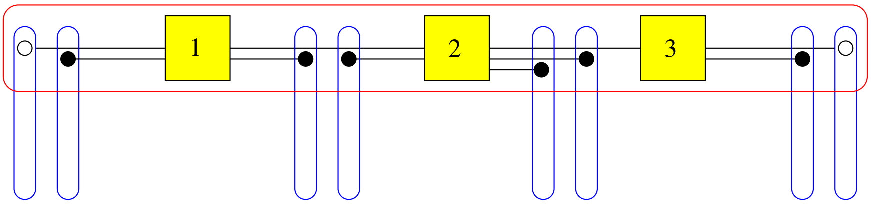


Figure 5.11: The segment from Figure 5.8 after removal of all zero-degree angles.

To calculate a complete shape extension of a `Kandinsky` shape we combine the algorithm to compute a segment-separated shape extension with the algorithm for rectangle decomposition. For each face we first perform segment-separation until each zero-degree angle is removed from the face and then use the traditional rectangle separation algorithm. The completion algorithm is summarized in algorithm *calculate shape extension*. From the fact that the original approach to remove zero-degree angles is correct and because our algorithm for the compaction of 4-graphs is correct, it follows:

**Lemma 5.4** *Algorithm* calculated shape extension *has linear running time and computes a complete shape extension of a* `Kandinsky` *shape $Q$.*

```
Algorithm 12: calculate shape extension
Input: Kandinsky shape $Q$
Output: Complete shape extension $\mathcal{S}$
calculate shape description $\mathcal{S}$;
for each $f \in Q$ do
    List $l \leftarrow \epsilon$;
    for each $(e = (v, w), d, a) \in f$ do
        // Let d' be the direction obtained by rotating d by 90°.;
        if $a = 0$ then append $(-1, seg(e), d)$ to $l$;
        if $a = 1$ then append $(0, seg(e), d)$ to $l$;
        if $a = 3$ then
            if $vertexbend(e) =$ true then
                append $(K, seg(e), d)$ to $l$
            else
                append $(1, seg(e), d)$ to $l$
        if $a = 4$ then append $(1, seg(e), d), (1, seg(w, d'), d')$ to $l$
    count = size(l)+2;
    // denote with $t_i = (a_i, s_i, d_i)$ the i-th tuple in $l$;
    while $count > 0$ do
        if $((a_1 = K)$ and $(a_2 = -1))$ or $((a_2 = -1)$ and $(a_3 = K))$
        then
            replace $t_1$ with $(0, s_1, d_1)$;
            remove $t_2$ from $l$;
        else
            move $t_1$ to the rear of $l$
        count=count-1;
    while $size(l) > 4$ do
        if $(a_1 = 0)$ and $(a_2 = 1)$ and $(a_3 = 1)$ then
            define-box$(\mathcal{S}, d_1, s_1, s_2, s_3, s_4)$;
            remove $t_2$ and $t_3$ from $l$;
        else
            move $t_1$ to the rear of $l$
    define-box$(\mathcal{S}, d_1, s_1, s_2, s_3, s_4)$;
```

### 5.4.3 Computing the Length Complete Shape Extension

In this section we will show how we can derive a length complete shape extension $\bar{\mathcal{S}}$ of the compaction shape $H_Q$.

For the remainder of this section let $\bar{\mathcal{S}}^Q = ((S^Q_{\rightarrow}, \bar{A}^Q_{\uparrow}), (S^Q_{\uparrow}, \bar{A}^Q_{\rightarrow}))$ be a complete shape extension of $Q$ and $\mathcal{S} = ((S_{\rightarrow}, A_{\uparrow}), (S_{\uparrow}, A_{\rightarrow}))$ be the shape description of $H_Q$. For each segment in $\bar{\mathcal{S}}^Q$ we define a *meta-segment* in $H_Q$:

**Definition 5.5** *Let $s$ be a segment in $\bar{\mathcal{S}}^Q$, the meta-segment $meta(s)$ of $s$ is defined as:*

$$\bigcup_{e \in E_s} simple(e) \cup \bigcup_{v \in V_s} seg(v, d(s))$$

*where*

$$seg(v,d) = \begin{cases} \emptyset & if \;\; v \in \hat{V} \\ \{l(v), r(v)\} & if \;\; v \notin \hat{V} \;\; and \; d = \uparrow \\ \{b(v), t(v)\} & if \;\; v \notin \hat{V} \;\; and \; d = \rightarrow . \end{cases}$$

Each segment $s$ of $H_Q$ is contained in exactly one meta-segment. We denote with $qseg(s)$ the segment $s'$ in $\bar{\mathcal{S}}^Q$ with $s \in meta(s')$.

To compute a length complete shape extension $\bar{\mathcal{S}}$ of $\mathcal{S}$ we first calculate a complete shape extension $\bar{\mathcal{S}}^Q$ of the shape description $\mathcal{S}^Q$ of $Q$ and then derive $\bar{\mathcal{S}}$ from $\bar{\mathcal{S}}^Q$. We add for each vertical, resp. horizontal, segment $s$ in $\mathcal{S}^Q$ two vertices $i(s)$ and $o(s)$ into the shape graph $D_{\rightarrow}$, resp. $D_{\uparrow}$, of $\mathcal{S}$ and for each $s' \in meta(s)$ the edges $(i(s), s')$ and $(s, o(s'))$. The vertex $i(s)$ is therefore a predecessor in the shape graph of all segments in $meta(s)$, and the vertex $o(s)$ a successor of all segments in $meta(s)$. For each edge $e = (v, w)$ in a shape graph of $\bar{\mathcal{S}}^Q$ we add an edge $p(e) = (o(v), i(v))$ to the corresponding shape graph of $\mathcal{S}$.

**Definition 5.6** *The shape extension $\bar{\mathcal{S}} = ((\bar{S}_{\rightarrow}, \bar{A}_{\uparrow}), (\bar{S}_{\uparrow}, \bar{A}_{\rightarrow}))$ of the shape description $\mathcal{S}$ is defined as*

$$\begin{array}{lcll} \bar{S}_{\rightarrow} & = & S_{\rightarrow} & \cup \;\; \{i(s), o(s) | s \in S^Q_{\rightarrow}\}, \\ \bar{S}_{\uparrow} & = & S_{\uparrow} & \cup \;\; \{i(s), o(s) | s \in S^Q_{\uparrow}\}, \\ \bar{A}_{\uparrow} & = & A_{\uparrow} & \cup \;\; \{(i(s), s') | s \in S^Q_{\rightarrow}, s' \in meta(s)\} \\ & & & \cup \;\; \{(s', o(s)) | s \in S^Q_{\rightarrow}, s' \in meta(s)\} \\ & & & \cup \;\; \{(o(s_1), i(s_2)) | (s_1, s_2) \in \bar{A}^Q_{\uparrow}\} \\ \bar{A}_{\rightarrow} & = & A_{\rightarrow} & \cup \;\; \{(i(s), s') | s \in S^Q_{\uparrow}, s' \in meta(s)\} \\ & & & \cup \;\; \{(s', o(s)) | s \in S^Q_{\uparrow}, s' \in meta(s)\} \\ & & & \cup \;\; \{(o(s_1), i(s_2)) | (s_1, s_2) \in \bar{A}^Q_{\rightarrow}\}. \end{array}$$

**Lemma 5.5** *The shape extension $\bar{\mathcal{S}}$ is length complete and has linear size according to $\bar{\mathcal{S}}^Q$.*

**Proof:** Let $s_1 \in D_\uparrow$ be a vertical segment and $s_2 \in D_\to$ a horizontal segment in $\mathcal{S}$. Let $ms_1 = qseg(s_1)$ and $ms_2 = qseg(s_2)$. Since $\mathcal{S}^Q$ is a complete shape extension either $ms_1$ and $ms_2$ are separated or adjacent.

We first consider the case that $ms_1$ and $ms_2$ are separated. We assume w.l.o.g. that $ms_1 \xrightarrow{*}_{D^Q_\to} vert(\alpha(ms_2))$, the other three cases are symmetric. We denote the path from $ms_1$ to $vert(\alpha(ms_2))$ with $mp = v_1, e_1, v_2, e_2, v_3, \ldots, e_k, v_{k+1}$. For each segment $s$ in $\mathcal{S}^Q$, the vertices $i(s)$ and $o(s)$ are connected by a collection of paths of length two. We denote with $p(s)$ one of these paths. Therefore the path obtained by concatenating the paths $p(e_1), p(v_2), p(e_2), p(v_3), \ldots, p(e_k)$ defines a path from $o(ms_1)$ to $i(vert(\alpha(ms_2)))$. Since $((s_1), o(ms_1)) \in \bar{A}_\to$ and $vert(\alpha(ms_2)) \xrightarrow{*}_{D^Q_\to} vert(\alpha(s_2))$ it holds $s_1 \xrightarrow{*}_{D_\to} vert(\alpha(s_2))$, which proves that $s_1$ and $s_2$ are separated.

If $ms_1$ and $ms_2$ are adjacent, then the ordering $\lhd$ ensures that $s_1$ and $s_2$ are separated.

Therefore all segments are separated which shows that $\bar{\mathcal{S}}$ is complete. The shape graphs do not contain paths in which both endpoints of the edge belong to the same meta-segment and which contain edges not defined in the shape description. If this were the case, these edges would have been induced by edges in $\mathcal{S}^Q$ which would imply that a shape graph in $\mathcal{S}^Q$ has a selfloop, which is contradictory to the assumption that the shape graphs are acyclic. Therefore the shape extension would contain a negative cycle, already the shape description would contain this negative cycle which is not the case. Therefore the shape extension is also length complete.

It remains to show that the size of $\bar{\mathcal{S}}$ is linear according to $\mathcal{S}^Q$. We introduce at most $2|E|$ segment vertices. Every segment in $H_Q$ is contained in one segment of $Q$ and is therefore connected twice to a segment vertex. The linear size of $\bar{\mathcal{S}}$ follows therefore directly from the construction. $\square$

### 5.4.4 Coordinate Assignment

To assign coordinates to the vertices in $H_Q$ we cannot use the standard longest path algorithm because there are cycles and negative edges in the shape graphs. We will present therefore a special tailored algorithm for this problem.

We will present the algorithm to calculate the x-coordinates from a complete shape extension $\bar{\mathcal{S}} = ((\bar{S}_\to, \bar{A}_\uparrow), (\bar{S}_\uparrow, \bar{A}_\to))$ of the shape description $\mathcal{S} = ((S_\to, A_\uparrow), (S_\uparrow, A_\to))$ of $H_Q$. The algorithm for the assignment of the y-coordinates is similar. The algorithm consists of three phases: In the first phase we calculate the offset of the segments at each vertex, then in the second phase, we compute the offset of the segments inside each meta-segment, and in the third phase we compute the distances between the meta-segments.

In the first phase we first remove the edges with negative length from the $D_\to$. For each vertex $u \in V$ we define $D_\to(u)$ as the subgraph $D_\to$ induced by the vertices:

$$S_{\uparrow}(u) = \{seg(simple(e)) | e \in adj(u) \text{ and } Q_d(e) = \uparrow\} \cup seg(u, \uparrow).$$

If $u$ has positive size, in other words $u \in V \setminus \hat{V}$, we compute the x-value $x_1(u, s)$ of the segments by computing the longest path value from the segment $l(u)$. If $u$ has zero size, in other words $u \in \hat{V}$, then there is only one segment $s$ in $D_{\rightarrow}(u)$ and we set $x_1(u, s) = 0$.

In the second phase, in which we compute the offset of the segments inside each meta-segment, we consider one vertical segment at the time. Let $s'$ be a vertical segment of $Q$ and $s = meta(s')$ the corresponding meta-segment. The x-coordinates $x_2$ can be easily computed by considering the vertices in $s'$ from bottom to top: Let $sk(s') = u_1, e_1, u_2, \dots, e_{k-1}, u_k$ be the skeleton of $s'$. We start with initializing the $x_2$ value of the segments in $S_{\uparrow}(u_1)$ with the value in $x_1$:

$$x_2(s) = x_1(u_1, s) \, \forall s \in S_{\uparrow}(u_1).$$

Then we compute iteratively the values for the remaining segments in the meta-segment with the formula:

$$x_2(s) = x_1(s, u_i) + x_2(seg(simple(e_k))) - x_1(u_i, seg(simple(e_k))) \, \forall s \in S_{\uparrow}(u_i).$$

In the third phase we compute the final coordinates. We compute the reduced graph of the horizontal shape graph of $\bar{\mathcal{S}}$: We remove the vertices of $S_{\uparrow}$ from the graph, and merge for a meta-segment $s$ the vertices $i(s)$ and $o(s)$ to one vertex $ms(s)$. For each edge $e = (s_1, s_2) \in \bar{A}_{\rightarrow}, s_1, s_2 \in S_{\rightarrow}$,with $qseg(s_1) \neq qseg(s_2)$, we define an edge $(ms(qseg(s_1)), ms(qseg(s_2)))$ with length $\max\{length(e) + x_2(s_1) - x_2(s_2), 0\}$. Again we apply the longest path algorithm to this graph and denote the result with $x_3$. We define the x-coordinate of a vertex $v \in V_H$ as:

$$x(v) = x_2(vert(v)) + x_3(qseg(vert(v))).$$

**Lemma 5.6** *The above algorithm calculates a feasible solution of* $(KLP)$ *in time* $O(n)$.

**Proof:**

Each edge $e = (s_1, s_2) \in D_{\rightarrow}$ with negative length has the form $(r(v), l(v))$ for some $v \in V$. Since $r(v)$ and $l(v)$ are in the same meta-segment and in $D_{\rightarrow}(v)$, it must hold for the distance between $r(v)$ and $l(v)$:

$$x(r(v)) - x(l(v)) = x_2(r(v)) - x_2(l(v)) = x_1(r(v)) - x_1(l(v)) \geq length(e)$$

which is ensured by the first phase of the algorithm.

Since for every edge $e = (s_1, s_2) \in D_{\rightarrow}$ with positive length holds

$$x(qseg(s_2)) - x(qseg(s_1)) \geq length(e) + x_2(s_1) - x_2(s_2)$$

it follows:

$$\begin{aligned} x(s_2) - x(s_1) \quad &= \quad x_2(s_2) + x(qseg(s_2)) \\ &\quad\; -x_2(s_1) - x(qseg(s_1)) \geq length(e). \end{aligned}$$

The proof for the y-coordinates is similar. The algorithm has linear running time because longest path calculation can be done in linear running time. □

### 5.4.5 Arbitrary Number of Edges at One Side

Until now we assumed that vertices have sufficient size to connect all edges adjacent to a certain side. However, this assumption may be not satisfied by the orthogonalization algorithm which provides the input for the compaction. In this case, we define in the simplification step the distances between certain edges adjacent to the same side of a vertex as zero such that the size constraint can be fulfilled. In the final drawing the edges which overlap can be separated by a small value. This is illustrated in Figure 5.12.

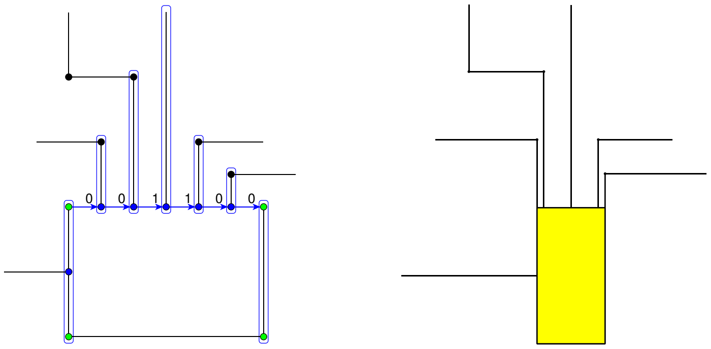


Figure 5.12: Vertex with not enough width to place all edges on the grid.

Another issue is the centering of edges. Since the coordinate assignment algorithm proceeds from left to right, all edges adjacent to a vertex tend to be on the left side and the bottom of the vertex. These drawings have a very unbalanced appearance, see Figure 5.13 for an example.

We can avoid this effect by changing the length value of the edges adjacent to corner vertices in $H_Q$. Consider for example the edges with direction $\uparrow$ adjacent to a vertex $v$. There is one edge $e$ without vertex bend, which we choose to center. The edges on the left of this edge have a left vertex bend, the edges on the right of this edge a right vertex bend. Let $k_1$ denote the

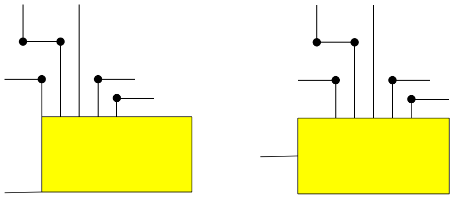


Figure 5.13: Unbalanced (left) and balanced (right) compaction.

number of edges on the left of $e$, $l$ the leftmost edge on the upper side of $v$, $k_2$ denote the number of edges on the right of $e$, and $r$ the rightmost edge on the upper side of $v$. Then we insert an edge $(l(v), seg(l))$ with length $k_1$ and an edge $(seg(r), r(v))$ with length $k_2$ into $D_{\rightarrow}$. An example for such a constraint graph is illustrated in Figure 5.14.

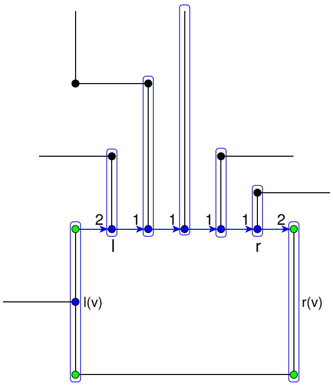


Figure 5.14: Shape graph for balanced compaction for the example in Figure 5.13.

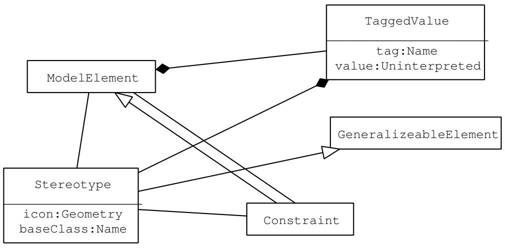
# Chapter 6

# Interactive Layout

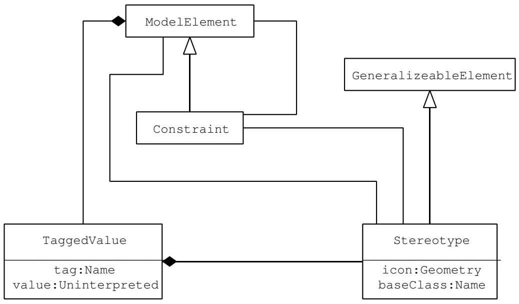
In this chapter we discuss the interactive layout of UML class diagrams. The content of this chapter is joint work with Ulrik Brandes and Dorothea Wagner, the presentation of the algorithms and methods follows partly [16].

The UML-`Kandinsky` algorithm for automatic layout of class diagrams presented in the preceding chapters is a static layout algorithm in the sense that given a diagram as input it computes a drawing only once. In UML modeling quite often users interact with the diagram, continuously adding and removing elements. Unfortunately, our static layout algorithm is not suitable for this scenario. If we compute a new drawing of a diagram after a small update of the diagram the result may be significantly different from the previous drawing. This makes it difficult for the user to read the new drawing because the internal picture of the diagram that the user has in his mind is disturbed. This internal picture is usually referred to as "mental map" [40]. A layout algorithm should make only small changes in the drawing when there are small changes in the diagram structure, which is called *preserving the user's mental map.* Updating a drawing correctly after updates of the diagram is referred to as *dynamic layout.*

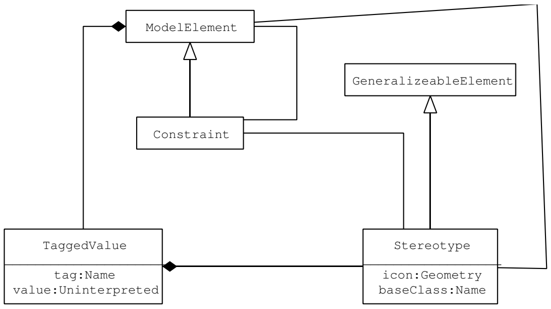
Another aspect of interactive layout is supporting user preferences. Since automatic layout algorithms are in general not perfect users often rearrange parts of the diagram after automatic layout to their personal preference without changing the content of the diagram. Often this aims to reflect their understanding of the system modeled by the diagram. An alternative view is that an existing drawing is to be improved subject to user-supplied constraints or hints for the algorithm analogously to [39]. Usually the user changes will not be taken into account by the dynamic layout algorithm. This has annoying consequences for the user, imagine the following scenario: In step (a) the user performs automatic layout on a diagram and then rearranges in step (b) a small part of the diagram to his personal preference. Then he adds in step (c) some elements to the diagram and quickly discovers that the diagram gets too crowded and the layout of the diagram must



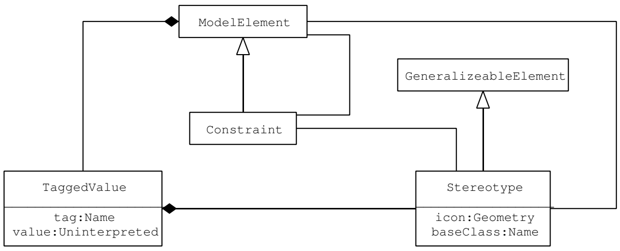
(a) input graph



(b) result of UML-`Kandinsky`

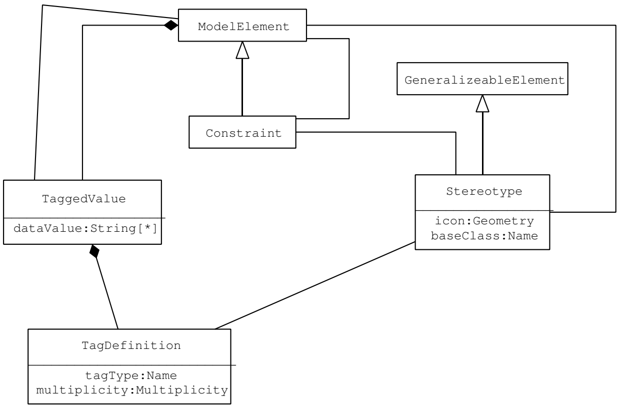


(c) user change

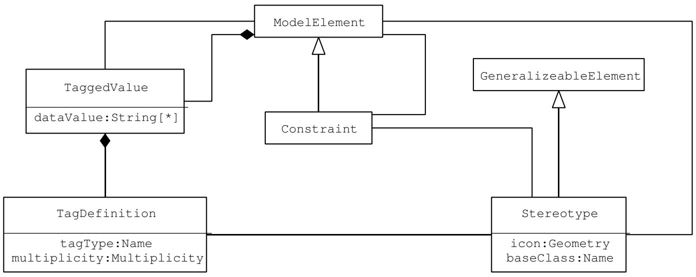


(d) result of applying sketch-driven algorithm

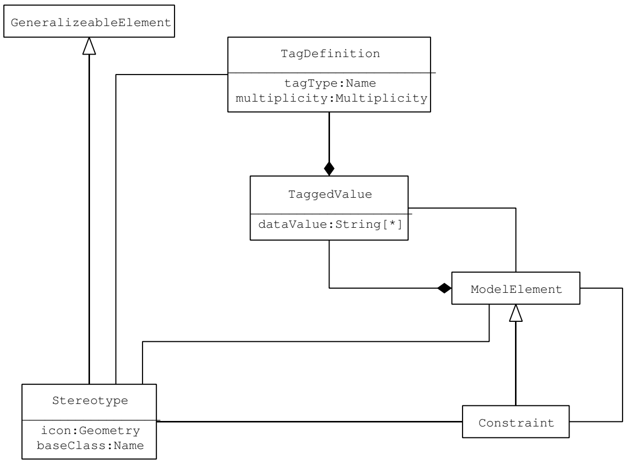
Figure 6.1: Example for interactive improvement of a diagram's layout.



(a) update on the diagram of Figure 6.1



(b) result of applying sketch-driven algorithm



(c) result of applying UML-`Kandinsky` to the updated diagram

Figure 6.2: Example for dynamic layout. While the sketch driven approach keeps the mental map, it is disturbed by applying the static UML-`Kandinsky` algorithm.

be updated. The user can choose now between two alternatives, neither of them is compelling: Either he uses automatic layout with the consequence that he has to perform step (b) again or he has to clean up the diagram by hand, in which case the user might be angry spending money for buying a tool with automatic layout facilities which fails to work correctly.

There is a fair amount of research on dynamic graph layout (see [19] for an overview) and on utilizing user interaction for force-directed [109, 75], quasi-visibility [98] and layered layout [39], there are even attempts to learn parameters of layout objectives from example drawings [91, 93], but there are no truly interactive algorithms for orthogonal drawing which support the scenarios described above.

In dynamic graph drawing, usually it is required that the input drawing is already drawn by the dynamic layout algorithm or is at least in the same representation as the target drawing. Often the algorithm keeps the state of the last run and uses this state information for the next run, see for example [94]. Keeping the state has many drawbacks, especially it makes the implementation of the algorithm costly and error-prone. The algorithm has to be informed about each change in the diagram, and if one change is missed, the algorithm and the diagram are out of synchronization which may have fatal consequences.

In this chapter we present a new interactive orthogonal graph drawing algorithm based on the topology-shape-metrics paradigm that does not keep state information explicitly and supports as well dynamic changes of the diagram as well as user preferences. It takes a diagram as input together with a drawing of a part of the diagram. We call the part of the diagram for which a drawing exists "sketch". Our algorithm computes from the diagram and the sketch a tidy, orthogonal drawing with few bends that preserves the overall appearance of the sketch. Note that our algorithm makes no assumptions about the sketch, it is for example not assumed that the drawing must be orthogonal or planar.

The "sketch" exhibits the main features to be conveyed, but may be unsatisfactory from an aesthetic point of view or incomplete since not every element of the diagram may be present in the sketch. The sketch represents in some sense the state of the previous runs of the algorithm together with the user preferences. We call therefore our algorithm *sketch-driven.*

Our method was inspired by SCHEMAP a system generating schematic maps for communication networks that was demonstrated at the Software Exhibition [88] of Graph Drawing 2001. This system gradually orthogonalizes a given network layout ("ground plan") into a schematic map while preserving the embedding (including crossings). Since it is based on a force-directed algorithm there is no guarantee that the result is indeed orthogonal, and running times are apparently far from interactive.

While our algorithm produces an orthogonal drawing with few bends in the `Kandinsky` model it also preserves the general appearance of the sketch.

Although we can devise an interactive layout algorithm for class diagrams with this approach, there are potential applications for this kind of drawing algorithm beyond class diagrams including the generation of schematic maps from geographic networks.

In Section 6.1 we review dynamic layout for orthogonal drawings and we give an overview over our interactive layout algorithm. The algorithm extends our automatic layout algorithm for class diagrams according to the Bayesian paradigm for dynamic layout of [17]. The two main parts of the algorithm are interactive planarization and interactive orthogonalization which are described in Section 6.2 and Section 6.3.

## 6.1 The Algorithm

Our algorithm for sketch-driven orthogonal graph layout relies on a framework for extending (static) layout algorithms to dynamic graphs proposed by Brandes and Wagner [17].

In dynamic graph drawing, the input is a sequence of graphs which represents the states of a single graph that is changing over time. Dynamic graphs can be visualized, for example, in an animation or in a sequence of drawings, but it is important to keep changes between consecutive frames to a minimum in order not to destroy a viewer's mental map of the graph [40]. Methods for dynamic orthogonal layout are proposed in [97, 21, 18, 26].

The core modeling task of dynamic layout, i.e. the combination of criteria for good (static) layout with the requirement of small change, is therefore very similar to that of sketch-driven layout. For layout algorithms based on the optimization of an objective function, the *Bayesian framework* [17, 14] suggests to incorporate a *difference metric* [22] as a penalty in the objective function. Optimization of the combined objective function thus naturally results in a trade-off between static layout criteria and stability.

Since orthogonal drawing algorithms in the topology-shape-metrics framework heavily depend on the angles and bends computed in the shape step, it seems natural to use the change in angles and bends as a difference metric for orthogonal shapes [18, 14].

Throughout this section, let $\Gamma$ be a drawing (a *sketch*) of a subgraph $G_\Gamma = (V_\Gamma, E_\Gamma)$ of the input graph $G = (V, E)$. Our objective is to determine an orthogonal box drawing of $G$ with the following properties:

- the topology of $G_\Gamma$ is preserved,
- the drawing is in the `Kandinsky` model,
- angles in the drawing deviate little from angles in the sketch (stability), and
- the drawing contains few bends (readability).

Our algorithm follows the topology-shape-metrics approach and proceeds as follows. First, a planarization of $G_\Gamma$ is determined by replacing each crossing in the sketch by a dummy vertex. Since we do not change the embedding in the following steps, this ensures that the topology of $G_\Gamma$ is preserved. The main problem in this phase is that the sketch may contain degeneracies which make it impossible to determine a planarization for the entire graph. Therefore we first determine a subgraph $G_{\text{valid}}$ of $G_\Gamma$ whose drawing contains no degeneracies and compute the planarization $G'$ for this subgraph. Then the edges in $E$ which are not part of $G_{\text{valid}}$ are inserted into $G'$ using algorithms from Chapter 3. This includes the edges which have not been defined in the sketch as well as the edges which have been removed from the sketch to remove degeneracies. This algorithm is described in detail in Section 6.2.

Next, a quasi-orthogonal shape $\hat{Q}$ of the subgraph of $G'$ induced by the edges of $G_{\text{valid}}$ is determined from $\Gamma$ by rounding angles in the sketch to the nearest multiple of 90° and classifying each edge bend as either a 90° or a 270° bend. Note that the resulting quasi-orthogonal shape $\hat{Q}$ needs not be valid. A `Kandinsky` shape $Q$ of $G'$ is then determined so as to satisfy a trade-off between stability and bend-number. Note that an angle in the sketch corresponds in general to a set of angles in the final drawing. Since we preserve the topology of the sketch, these sets of angles are intervals. We can therefore use the Constrained `Kandinsky` approach from Chapter 4 to compute $Q$. The details of the algorithm are described in detail in Section 6.3.

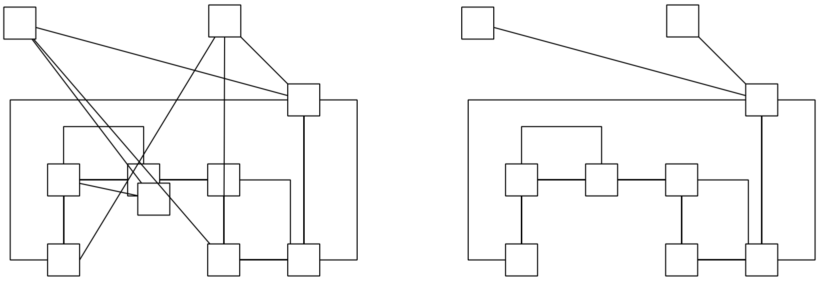
Finally, a standard compaction algorithm is applied to $Q$ to compute a drawing of $G'$, from which the final drawing is obtained by replacing dummy vertices with edge crossings.

Our algorithm ignores the FLOW and HYPEREDGE aesthetic criteria. The reason for this is that the user has already defined a direction in the sketch which may be conflicting with the FLOW and HYPEREDGE criteria and we value the decision of the user more than these abstract aesthetic criteria. If we want to consider these criteria we would have to make sure that the interactive planarization is mixed-upward planar and merge the user supplied constraints with the constraints defined by the FLOW and HYPEREDGE aesthetic criteria, where the last ones have higher priority.

Our approach is a fully interactive orthogonal graph drawing algorithm when it considers the most recent drawing as sketch, which itself does not contain newly added elements.

## 6.2 Interactive Planarization

In Chapter 3 an algorithm was presented, which computes a planarization of a graph from scratch. In our setting the planarization is already given



(a) A graph containing degeneracies

(b) Valid subgraph.

Figure 6.3: Example for the valid subgraph of a degenerated input graph.

as a drawing of the input graph and the problem consists of extracting it from the drawing. This problem can be subdivided into two subproblems: First determine the planarization of the graph by computing the crossings in the drawing and second computing the embedding by determining the circular order of the edges around the vertices in the drawing and the outer face. The problem is made more difficult by the fact that the drawing might contain degeneracies. Suppose, for example, that all vertices of the graph are placed at the origin $(0, 0)$. In this case it is not possible to extract any useful information from the drawing, especially not a planarization of the input. Therefore we first compute a subgraph $G_{\text{valid}} = (V, E_{\text{valid}})$ of $G$ from whose drawing $\Gamma(G_{\text{valid}})$ we can compute a planarization. See Figure 6.3 for an example. We additionally determine the angle between neighbor edges in the embedding and the bends of the edges. This information is needed by the orthogonalization algorithm in the next section. After computing the planarization of $G_{\text{valid}}$ the edges $E \setminus E_{\text{valid}}$ are inserted into the planarization using the algorithm *undirected edge insertion* described in Section 3.3.

We denote the resulting planarization with $G' = (V', E', F')$. The edges in $E_{\text{valid}}$ are in general subdivided to paths in $G'$. We define the set $\hat{E} = \{\hat{p}(e) | e \in E_{\text{valid}}\} \subseteq E'$, as the set of edges in $E'$ which are part of an subdivision of an edge in $E_{\text{valid}}$.

### 6.2.1 The Straight Line Segment Intersection Problem

The main tool for our algorithm for interactive planarization is an algorithm for straight line segment intersection. The straight line segment intersection problem consists of finding all crossings between two segments of a set of straight line segments $S$. The variant of Näher and Mehlhorn [92] of the famous Bentley-Ottmann sweep line algorithm [6] does not only report the crossings, it further supplies a plane graph $G'$ as result. The vertices of $G'$ are all endpoints and all proper intersection points of the segments in $S$. The edges of $G'$ are the maximal relatively open subsegments in $S$ that contain no vertex of $G'$. The algorithm has running time $O((n + s) \log n)$, where $n$ is the number of segments and $s$ is the size of the graph $G'$. It is important to note that the algorithm makes no assumptions about the input, the segments may have arbitrary coordinates. We refer to this algorithm as algorithm *Straight Line Segment Intersection.*

### 6.2.2 Valid Drawings

In this section we present a set of properties for a drawing of a graph that are sufficient to extract a planarization from the drawing. These properties are summarized in Definition 6.1:

**Definition 6.1** *A rectangle-drawing* $\Gamma(G)$ *of a graph* $G = (V, E)$ *is said to be* valid *if it has the following properties:*

**V1** *The path* $\Gamma(e)$ *of an edge* $e = (a, b) \in E$ *starts at the boundary of* $\Gamma(a)$ *and ends at the boundary of* $\Gamma(b)$.

**V2** *The rectangles* $\Gamma(v)$ *and* $\Gamma(w)$ *of two vertices* $v, w \in V$ *do not intersect nor do they contain each other.*

**V3** *The rectangle* $\Gamma(v)$ *and the path* $\Gamma(e)$ *of a vertex* $v \in V$ *and an edge* $e \in E$ *do not intersect nor does* $\Gamma(v)$ *contain* $\Gamma(e)$.

**V4** *Two paths* $\Gamma(e_1)$ *and* $\Gamma(e_2)$ *of two edges* $e_1, e_2 \in E$ *share only a finite number of points.*

**V5** *No point lies in the interior of more than two paths of edges in* $E$.

*A drawing* $\Gamma(G)$ *of a graph* $G$ *is called* degenerated *if it is not valid.*

Let $G = (V, E)$ be a graph with valid rectangle-drawing $\Gamma$. We can derive a planar point-drawing $\Gamma'$ of $G$ the following way: First we define for each rectangle $\Gamma(v)$ of a vertex $v \in V$ a point $\Gamma'(v)$ in the interior of $\Gamma(v)$. For an edge $e = (v, w) \in E$ we define $\Gamma'(e)$ as the composition of a straight line from $\Gamma'(v)$ to the starting point of $\Gamma(e)$, $\Gamma(e)$, and a straight line from the endpoint of $\Gamma(e)$ to $\Gamma'(w)$. Furthermore we replace each edge crossing by a vertex. Property V5 ensures that each crossing vertex has degree four, property V4 ensures that each edge has only finitely many crossings. Properties V1, V2, and V3 ensure that there are no crossings inside the rectangles representing a vertex. We can now use algorithm *Straight Line Segment Intersection* on $\Gamma'$ to compute a planarization of $G$.

### 6.2.3 Determining the Valid Subgraph

We will now present an algorithm for determining a valid subgraph $G_{\text{valid}}$ of a graph $G_\Gamma = (V, E_\Gamma)$ with drawing $\Gamma$. We will first compute the line segment intersections in $\Gamma$ which will help us to remove the graph elements from the input which cause degeneracies in the drawing.

Before we determine the subgraph $G_{\text{valid}}$ for which $\Gamma$ defines a valid drawing, we ensure that the path $\Gamma(e)$ of each edge $e \in E_\Gamma$ fulfills property V1. For each edge $e = (a, b)$ we connect the first, resp. last, point of $\Gamma(e)$ with the center of $\Gamma(a)$, resp. $\Gamma(b)$, with a straight line. Then we clip the resulting path against $\Gamma(a)$ and $\Gamma(b)$.

Then we transform our input drawing, which consists of rectangles for vertices and paths for edges, in a set of line segments $S(\Gamma)$. The set $S$ contains for each edge $e \in E_\Gamma$ the line segments of $\Gamma(e)$ and four line segments for each vertex $v \in V$ which represent the boundary of $\Gamma(v)$.

**Lemma 6.1** *A drawing* $\Gamma$ *of a graph* $G_\Gamma = (V_\Gamma, E_\Gamma)$ *is valid if, and only if the following holds for the result* $G'(V', E')$ *of algorithm* Straight Line Intersection*:*

**L1** *There are no vertices in the interior of* $\{\Gamma(v) | v \in V_\Gamma\}$.

**L2** *Each vertex in the interior of* $\mathbb{R}^2 \setminus \{\Gamma(v) | v \in V_\Gamma\}$ *has degree two or four. If a vertex* $v' \in V'$ *has degree two the adjacent edges of* $v'$ *point to segments that belong to the same edge in the input graph. If* $v'$ *has degree four, the adjacent edges can be partitioned in two pairs of edges each pointing to segments that belong to the same edge in the input graph.*

**L3** *Each vertex* $v' \in V'$ *on the boundary of* $\Gamma(v), v \in V_\Gamma$ *is adjacent to edges pointing to segments representing the boundary or to the first, resp. last, segment of an edge adjacent to* $v$.

**L4** *There are no multiple edges in* $G'$.

**Proof:** That the above criteria are necessary follows directly from definition 6.1 and $G'$. It remains to show that these criteria are also sufficient. That property V1 is fulfilled follows from our preprocessing step. If property V2 had been violated at most one corner of the rectangle would be in the interior of another rectangle (conflicting with L1) or the two rectangles would touch (conflicting with L4). If property V3 had been violated, property L3 would not be true, and if property V4 had been violated, property L4 would not hold. If finally V5 had been violated again property L2 would not hold. □

Our algorithm for computing $G_{\text{valid}}$ works as follows: We first set $G_{\text{valid}}$ to $G_\Gamma$. Then we determine vertices violating condition V2 by a sweepline algorithm and remove them from $G_{\text{valid}}$. Next we run the straight line segment intersection algorithm on $S(\Gamma_{|G_{\text{valid}}})$ which yields $G' = (V', E')$. For each vertex of $v' \in V'$ we check conditions L1, L2 and L3. Edges violating the conditions are removed from $G_{\text{valid}}$. Finally we remove edges violating condition L4.

## 6.3 Interactive Orthogonalization

In this section we describe an orthogonalization algorithm for the sketch driven approach. The presented algorithm optimizes a bi-criteria optimization function where the two objectives are readability (number of bends) and stability (change in shape).

The readability of a shape $Q$ is independent of the given sketch and defined as the total number of bends, namely

$$\#bends(Q) = \frac{1}{2} \sum_{f \in Q} \sum_{(e,a,b) \in Q(f)} |b| \ .$$

To measure the stability of a quasi-orthogonal shape $Q$ we have to compare it with the quasi-orthogonal shape $\hat{Q}$ of the sketch drawing. However, $Q$ is defined on $G'$ while $\hat{Q}$ is defined on $(V', \hat{E})$. Since $(V', \hat{E})$ is a subgraph of $G'$, an angle between two edges $e_1$ and $e_2$ in $\hat{Q}$ corresponds to a set of angles in $Q$, defined by the edges between $e_1$ and $e_2$. We denote with $I(e)$ for an edge $e \in \hat{E}$ the set of edges consisting of $e$ and the edges following $e$ in the embedding of $Q$ which are not in $\hat{E}$.

The stability of a quasi-orthogonal shape $Q$ with respect to the quasi-orthogonal shape $\hat{Q}$ defined by the sketch is then expressed in terms of the difference between angles in $Q$ and corresponding angles in $\hat{Q}$

$$\Delta_A(Q, \hat{Q}) = \sum_{f \in \hat{Q}} \sum_{(e,a,b) \in f} |\hat{Q}_a(e) - \sum_{e' \in I(e)} Q_a(e')|$$

and the difference in edge bends

$$\Delta_B(Q, \hat{Q}) = \frac{1}{2} \sum_{f \in \hat{Q}} \sum_{(e,a,b) \in f} \text{edit}(\hat{Q}_b(e), Q_b(e)) \ .$$

This is similar to the shape difference metrics used in [22]. From these components we form a weighted compromise between the degree of change with respect to the given sketch and the number of bends in the quasi-orthogonal shape. Our objective function thus reads

$$D(Q|\hat{Q}) = \underbrace{\alpha \cdot \Delta_A(Q, \hat{Q}) + \beta \cdot \Delta_B(Q, \hat{Q})}_{\text{stability}} + \underbrace{\gamma \cdot (\#bends(Q) - \#bends(\hat{Q}))}_{\text{readability}}$$

where parameters $\alpha$, $\beta$, and $\gamma$ control the relative importance of angle or bend changes and bend number. We are now ready to state our problem formally.

**Definition 6.2** *Given a quasi-orthogonal shape $\hat{Q}$ of a plane graph $G'$, the SKETCH ORTHOGONALIZATION problem is finding a valid* `Kandinsky` *shape $Q$ of $G'$ such that $D(Q|\hat{Q})$ is minimum.*

If we have a closer look at the SKETCH ORTHOGONALIZATION problem we discover that it is quite similar to the CONSTRAINED KANDINSKY BEND MINIMIZATION problem defined in Section 4.5. In fact we can reduce it to a special instance of the CONSTRAINED KANDINSKY BEND MINIMIZATION problem.

Let $\pi$ be an ordering of the vertices $V$ of a graph $G = (V, E)$. The set $E_\pi = \{(u, v)|\{u, v\} \in E \text{ and } \pi(u) < \pi(v)\}$ contains for each edge $E$ of $G$ exactly one dart.

**Theorem 6.1** *Let $G' = (V', E', F')$ denote a plane graph, $\hat{Q}$ a quasi-orthogonal shape of the subgraph $(V, \hat{E})$ of $G'$, and $\pi$ an ordering of $V'$. We define the set of angle-constraints $AC$ as*

$$AC = \bigcup_{f \in \hat{Q}, (e,a,b) \in f} (I(e), \hat{Q}_a(e), \alpha)$$

*and the set of bend constraints $BC$ as*

$$BC = \bigcup_{e \in \hat{E}_\pi} (e, \hat{Q}_b(e), \beta + \gamma, \beta - \gamma) \cup \bigcup_{e \in E_\pi \setminus \hat{E}_\pi} (e, \epsilon, \gamma) \ .$$

*A solution $Q$ of the CONSTRAINED KANDINSKY BEND MINIMIZATION problem for $G'$, $AC$, and $BC$ is an optimal solution of the SKETCH ORTHOGONALIZATION problem for $G'$ and $\hat{Q}$.*

**Proof:**

For the stability of angles at vertices holds:

$$\begin{aligned} \alpha \cdot \Delta_A(Q, \hat{Q}) \quad &= \quad \alpha \cdot \sum_{f \in \hat{Q}} \sum_{(e,a,b) \in f} |\hat{Q}_a(e) - \sum_{e' \in I(e)} Q_a(e')| = \\ & \quad \alpha \cdot \sum_{(I(e),a,c) \in AC} |a - \sum_{e' \in I(e)} Q_a(e)| = \\ & \quad \sum_{(I,a,c) \in AC} c \cdot |a - \sum_{e \in I} Q_a(e)| \end{aligned}$$

while for stability and cost of bends holds:

$$\begin{aligned}
&\beta \cdot \Delta_B(Q,\hat{Q}) + \gamma \cdot (\#bends(Q) - \#bends(\hat{Q})) = \\
&\quad \frac{\beta}{2} \cdot \sum_{f \in \hat{Q}} \sum_{(e,a,b) \in f} \mathrm{edit}(\hat{Q}_b(e), Q_b(e)) + \\
&\quad \frac{\gamma}{2} \cdot (\sum_{f \in Q} \sum_{(e,a,b) \in f} |Q_b(e)| - \sum_{f \in \hat{Q}} \sum_{(e,a,b) \in f} |\hat{Q}_b(e)|) = \\
&\quad \sum_{e \in \hat{E}_\pi} (\beta \cdot \mathrm{edit}(\hat{Q}_b(e), Q_b(e)) + \gamma \cdot (|Q_b(e)| - |\hat{Q}_b(e)|)) + \\
&\quad \sum_{e \in E_\pi \setminus \hat{E}_\pi} \gamma \cdot |Q_b(e)| \overset{Lemma\ 2.2}{=} \sum_{(e,b,c_i,c_d) \in BC} (\mathrm{edit}_{(c_i,c_d)}(b, Q_b(e)) \ .
\end{aligned}$$

This yields for the objective function:

$$\begin{aligned}
&D(Q|\hat{Q}) = \\
&\quad \alpha \cdot \Delta_A(Q,\hat{Q}) + \beta \cdot \Delta_B(Q,\hat{Q}) + \gamma \cdot (\#bends(Q) - \#bends(\hat{Q})) = \\
&\quad \sum_{(I,a,c) \in AC} c \cdot |a - \sum_{e \in I} Q_a(e)| + \sum_{(e,b,c_i,c_d) \in BC} (\mathrm{edit}_{(c_i,c_d)}(b, Q_b(e)) \ .
\end{aligned}$$

This corresponds exactly to the definition of the CONSTRAINED KANDINSKY BEND MINIMIZATION problem in Definition 4.13 which finishes the proof. □

The algorithms presented in Chapter 4 to solve the CONSTRAINED KANDINSKY BEND MINIMIZATION problem apply therefore also for the SKETCH ORTHOGONALIZATION problem. Like the complexity of the CONSTRAINED KANDINSKY BEND MINIMIZATION, the complexity of the SKETCH ORTHOGONALIZATION problem is unknown.

# Chapter 7

# Experiments

In this chapter we provide experimental evaluations of the algorithms presented in this work. The implementations of UML-`Kandinsky` is presented shortly in Section 7.1 The experimental setting and the test data are described in Section 7.2.

In Section 7.3 we compare our implementation of the UML-`Kandinsky` algorithm to SugiBib, an automatic layout algorithm for class diagrams based on the hierarchical graph drawing approach. We perform the tests on a set of automatically generated class diagrams.

In the following we evaluate the algorithms for each phase of the topology-shape-metrics approach. In Section 7.4 we study the mixed upward planarization in different settings. First we show how the number of iterations in algorithm mixed GT (`MGT`) affects the number of crossings in the planarization. Then we consider the important case of the planarization of directed graphs and compare our algorithm for mixed-upward planarization to the hierarchical approach. In Section 7.5 we measure the performance of the improved heuristic for the KANDINSKY BEND MINIMIZATION problem. We compare it to the optimal solution obtained from a integer linear program solver and to the SSP heuristic. In Section 7.6 we study the performance of our compaction algorithm, especially we measure the impact of visibility post-processing.

Finally we present in Section 7.6 some examples drawings produced by UML-`Kandinsky` .

## 7.1 Implementation of UML-`Kandinsky`

All algorithms presented in this work have been implemented in the programming language Java and are based on or part of yFiles, a powerful Java library for graph visualization. The presentation of yFiles and the implementation of the algorithms follows partly [118, 119].

*yFiles* is a Java-based library for the visualization and automatic layout of graph structures. Conceptually, the *yFiles* library consists of three cooperating components:

The *yFiles Basic component* contains essential classes and data structures. It provides very efficient implementations of advanced data structures like graph, trees and priority queues. It furthermore makes available a wide variety of graph and network algorithms.

The *Viewer/Editor component* is built upon the Basic component. It provides a powerful graph viewer component and other Java-Swing based graphical user interface (GUI) elements.

The *yFiles Layout component* is also build upon the Basic component. It provides a large suite of graph layout algorithms. Diverse layout styles like hierarchical, orthogonal or circular are supported.

The Layout as well as the Viewer/Editor component can be used as independent building blocks. Based on the single components, extension packages in different directions are available and can be added, like support for different data formats or special applications like biochemical networks. Our implementation of the layout algorithm for class diagrams is such an extension package.

Currently the layout component of *yFiles* includes graph layout algorithms for the following styles: Hierarchical, tree, force-directed, circular-radial and orthogonal. These algorithms are mostly tuned variants of published algorithms [106], [114], [56], [60], [113]. Besides layout algorithms which assign coordinates to edge paths and nodes, *yFiles* also supports the automatic assignment of edge and node label coordinates [24].

The orthogonal layout algorithm of yFiles is our implementation of the `Kandinsky` algorithm. The parts of the implementation which are of general interest in graph visualization have been included in the standard yFiles distribution, this packages have prefix `y`. The parts more specialized for the automatic layout of class diagrams have been bundled in an extension package, these packages have prefix `yext`. This is illustrated by package structure of our algorithms:

- The package `y.layout.planar` contains classes related to planarity. This includes implementation of a planar representation, the Goldschmit/Takvorian algorithm and *undirected edge insertion* from Chapter 3.

- The package `y.layout.orthogonal` contains an implementation of the `Kandinsky` and Constrained `Kandinsky` algorithm of Chapter 4 as well as an implementation of the compaction algorithm of Chapter 5.

- The package `yext.layout.upwardplanar` contains classes related to mixed upward planarity. especially an implementation of the mixed upward planarization algorithm of Chapter 3.

- The package `yext.layout.orthogonal.mixed` contains the implementation of the orthogonalization algorithm for UML diagrams described in 4.
- The package `yext.uml.layout.orthogonal` contains the layout algorithm for UML class diagrams.

## 7.2 Data and Experimental Setting

All experiments have been performed on a Pentium IV System with 1.8 GHz and 512 Megabyte main-memory running Redhat Linux 8.0. We used the Sun JDK 1.4.1 for Linux as runtime-environment. We used five test sets for our experiments: Class Diagrams, Rome Graphs, Directed Rome Graphs, Upward Planar Graphs, and Graphs with Limited Height. These tests are now described in detail.

### 7.2.1 Class Diagrams

To test the performance of our algorithm on a larger data set we generated class diagrams automatically from the byte code of Java programs and Java libraries. For each package in the byte code, a diagram is created which contains the classes and interfaces of the package. The diagram contains all generalization relationship between classes/interfaces, and associations which are defined by a field of a class. Each connected component of the diagram with more than two nodes defines one problem instance.

Our test set consists of 1324 class diagrams and is generated from the following data:

- all classes located in the java-archives which ship with JBoss 4.0 Developer Release (835 diagrams). JBoss is an open-source application server which uses a lot of libraries from the open source community.
- the classes of the runtime-environment of the Java Development Kit (JDK) 1.4.1 located in the Java-archive `rt.jar` (407 diagrams)
- the classes defined by `batik.jar`, a library for treating SVG documents in Java (82 diagrams).

The average density of diagrams is 1.06 and the average percentage of generalization edges in the diagram is 0.49.

### 7.2.2 Rome Graphs

The Rome-graphs test suite [34] contains about 11.000 undirected connected graphs. The number of nodes in the test suite ranges from 10 to 100, the average density of the graphs ranging from 1 to 2 with average value of 1.3.

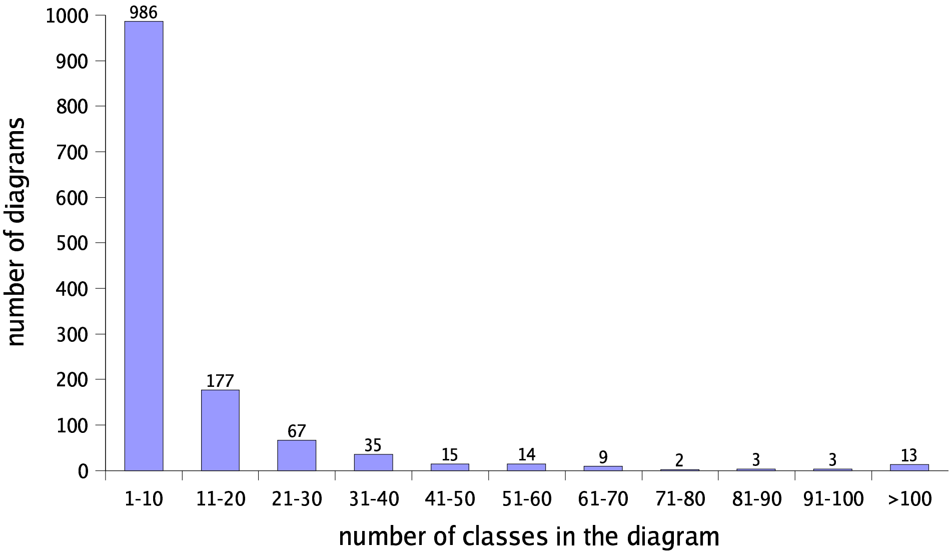


Figure 7.1: Distribution of the number of class diagrams with respect to number of classes in the diagram.

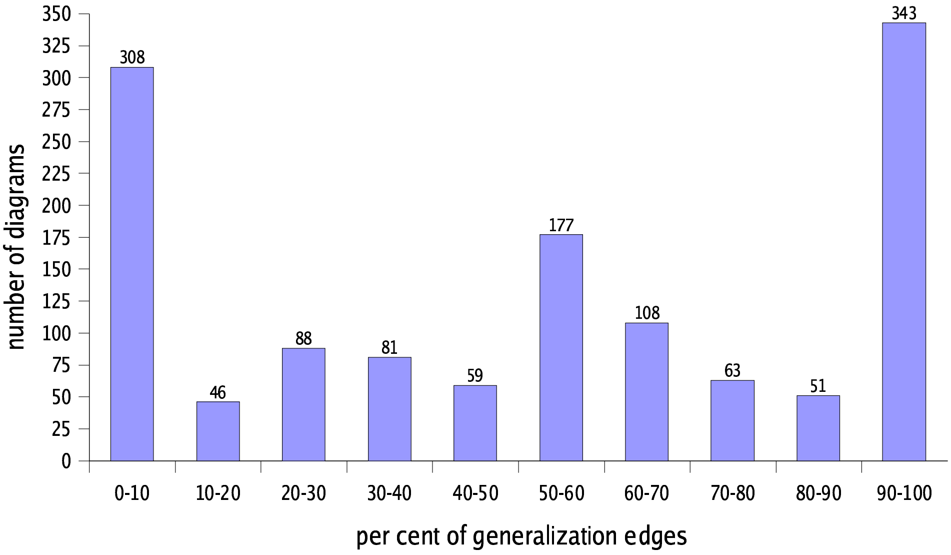


Figure 7.2: Distribution of percentage of fraction of generalization edges in the diagram.

### 7.2.3 Directed Rome Graphs

This data set is the directed version of the Rome Graphs test suite. We transformed the undirected graphs form the Rome Graphs test suite to directed acyclic graphs by directing the edges according to an ordering of the nodes of the graph. As ordering we chose the implicit ordering of the nodes as defined in the file.

### 7.2.4 Upward Planar Graphs

There are two test sets consisting of connected upward planar graphs, the first contains graphs with density 1.3, the second graphs with density 2.6. Both test sets contain 910 upward planar graphs, the number of nodes ranging in each from 10 to 100, containing 10 graphs for each node count. The graphs were generated the following way: First a random set of points in a triangle was generated. For this point set a delauney triangulation was performed which yields a planar triangulated graph. We deleted edges randomly until we reached the desired density. To assure that the generated graphs were connected we computed a spanning tree of the triangulated graph by randomized DFS and ensured that edges in the spanning tree are not deleted in the previous step. Finally we directed the edges according to the coordinates of their endpoints.

### 7.2.5 Graphs With Limited Height

There are two test sets which contain connected directed graphs with maximum height three, the first with density 1.3, the second with density 2.6. Maximum height three means in this setting that they have a layer assignment with at most three layers. Both test sets contain 910 upward planar graphs, the number of nodes ranging in each from 10 to 100, containing 10 graphs for each node count. The graphs were generated the following way: First we distributed randomly the nodes in three layers. To assure that a generated graph was connected we generated a spanning tree for it. Then we inserted edges randomly between nodes in neighbored layers until the desired density was reached.

## 7.3 Comparing UML-`Kandinsky` to SugiBib

In this section we compare our implementation of UML-`Kandinsky` to SugiBib, a sophisticated automatic layout algorithm for UML class diagrams based on the hierarchical layout paradigm. See Section 2.6.2 for a description of SugiBib. We compare the running time of the algorithms, the number of crossings, the number of bends, the width, the height and the size of the diagrams.

We used the following parameters for UML-`Kandinsky` in the tests. We use 50 iterations in the MGT algorithm for determining the mixed-upward-planar subgraph and enabled rerouting as postprocessing. In the compaction phase we enabled visibility postprocessing.

For SugiBib we used the following parameters communicated by the authors of the implementation:

| | |
|---|---|
| OrderingStrategy | ORDER_HIERARCHICAL_LATE |
| CoordinatesStrategy | COORDINATES_DETERMINISTIC |
| ImproveLayout | false |
| NodeSubLayout | false |
| HybridNodeSubLayout | false |
| StretchEntireGraph | false |
| LateOrthogonalization | true |

We performed the tests on the class diagram test set described in Section 7.2. Unfortunately SugiBib failed on 35 of the 1324 test instances. We excluded these instances from the test set for both algorithms.

Both algorithms show a different behavior for small and for large instances in the test set. While both algorithms perform satisfactory for instances of up to 80 classes, the performance of both algorithms degrades for instances with more than 80 classes. We first present the results of the comparison for small instances, then cover large instances.

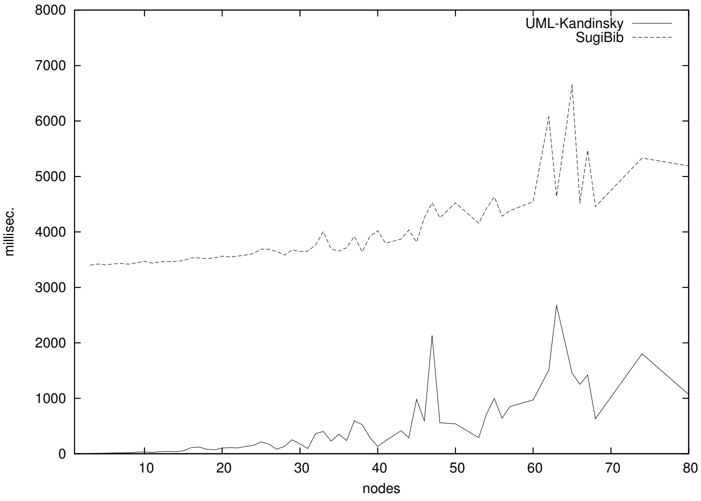
### 7.3.1 Tests on Diagrams with up to 80 Classes

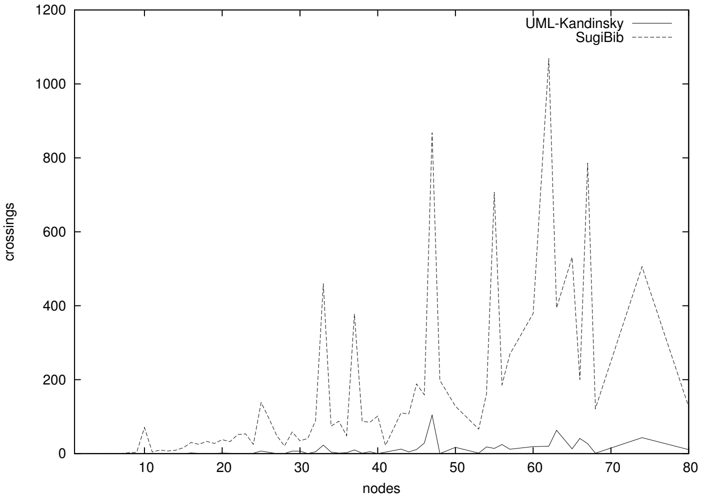
In this section we compare both algorithms on class diagrams with less than 80 classes.

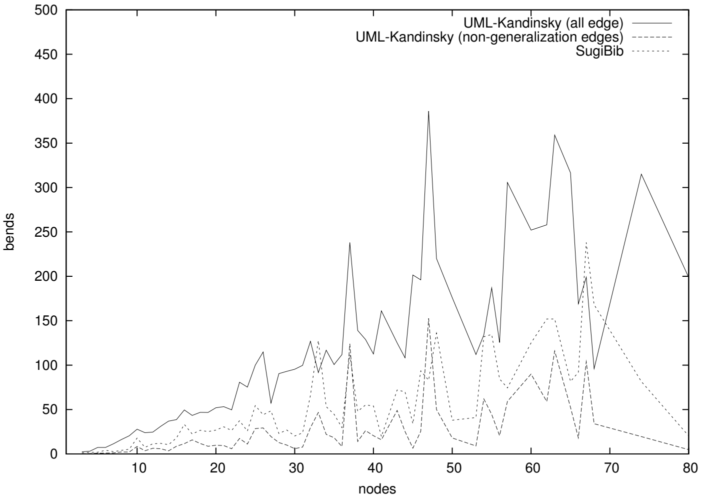
In terms of running time and number of crossings UML-`Kandinsky` is clearly superior to SugiBib. The comparison of the running times of the algorithms has to be interpreted with care since it is not only heavily dependent on the algorithm but also on the implementation of the algorithm. But nevertheless it is an indication that the topology-shape-metrics approach can compete with the hierarchical approach in terms of running time on these types of test instances.

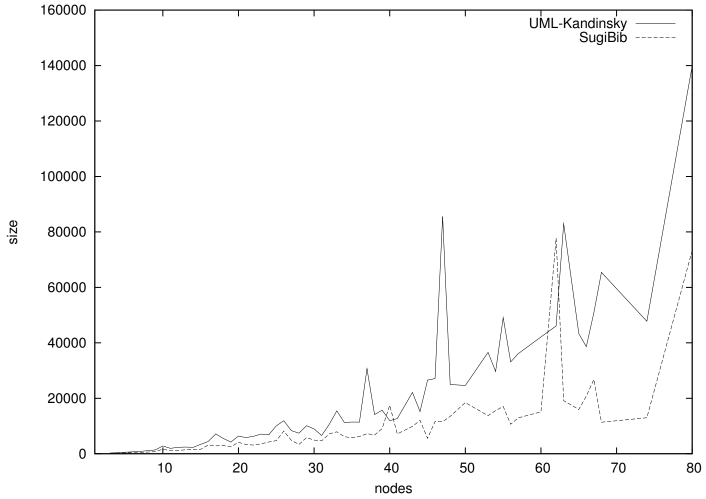
A more significant measure is the number of crossings in the drawing. We have seen in the introduction that the number of crossings in a drawing has a tremendous impact on its readability. Looking at the test-results we see that drawings of SugiBib have a lot more crossings than drawings of UML-`Kandinsky`. In terms of crossings UML-`Kandinsky` outperforms SugiBib very clearly.

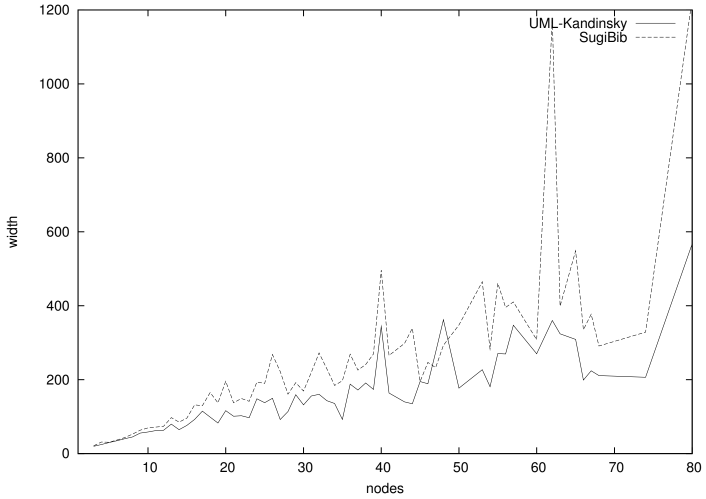
Drawings of SugiBib have a lower number of bends as the drawings of UML-`Kandinsky`. This is due to the fact that generalization edges are drawn straight-line and only association edges are drawn orthogonal and may pro-

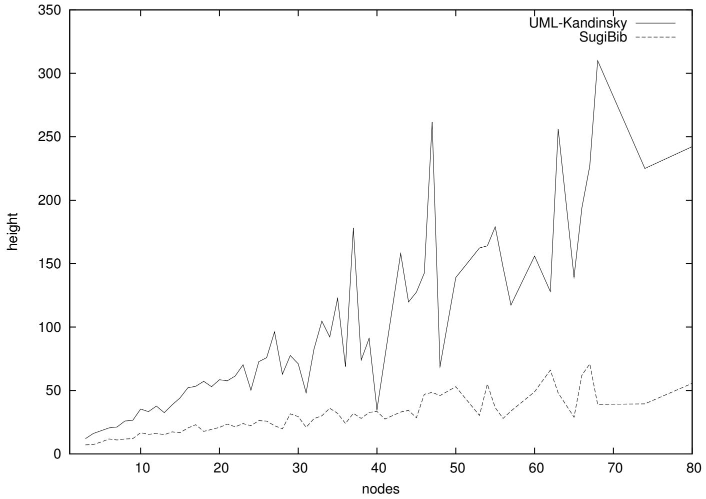

UML-Kandinsky
SugiBib
millisec.
8000
7000
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
nodes


UML-Kandinsky
SugiBib
crossings
1200
1000
800
600
400
200
0
10
20
30
40
50
60
70
80
nodes


UML-Kandinsky (all edge)
UML-Kandinsky (non-generalization edges)
SugiBib
bends
500
450
400
350
300
250
200
150
100
50
0
10
20
30
40
50
60
70
80
nodes


UML-Kandinsky
SugiBib
size
0
20000
40000
60000
80000
100000
120000
140000
160000
nodes
10
20
30
40
50
60
70
80


UML-Kandinsky
SugiBib
width
0
200
400
600
800
1000
1200
nodes
10
20
30
40
50
60
70
80


UML-Kandinsky
SugiBib
height
0
50
100
150
200
250
300
350
nodes
10
20
30
40
50
60
70
80

duce bends while in the UML-`Kandinsky` algorithm all edges are drawn orthogonal. It makes therefore more sense to compare the number of bends only for association edges, in which case UML-`Kandinsky` outperforms SugiBib clearly.

Comparing the area used by the drawings of both algorithms, we see that drawings of SugiBib use significantly less area than drawings of UML-`Kandinsky`. While the width of UML-`Kandinsky` drawings is even a little smaller on the average, the height of the drawings is significantly greater.

### 7.3.2 Tests on Diagrams with more than 80 Classes

In this section we compare both algorithms on class diagrams with more than 80 classes.

In the original test set there are 19 diagrams with more than 80 classes. For one instance of this test with 96 classes the execution of SugiBib failed. We therefore consider only 18 diagrams for our comparison of the algorithms for large class diagrams. The 18 diagrams have the following properties:

| Nr | Classes | Edges | Density | Upwardness |
|---|---|---|---|---|
| 1 | 82 | 81 | 0.99 | 1 |
| 2 | 84 | 93 | 1.11 | 0.35 |
| 3 | 84 | 110 | 1.31 | 0.75 |
| 4 | 91 | 124 | 1.36 | 0.6 |
| 5 | 96 | 126 | 1.31 | 0.83 |
| 6 | 103 | 263 | 2.55 | 1 |
| 7 | 107 | 147 | 1.37 | 0.55 |
| 8 | 111 | 159 | 1.43 | 0.85 |
| 9 | 122 | 200 | 1.64 | 0.65 |
| 10 | 132 | 235 | 1.78 | 0.66 |
| 11 | 138 | 232 | 1.68 | 0.52 |
| 12 | 139 | 288 | 2.07 | 0.53 |
| 13 | 159 | 257 | 1.62 | 0.4 |
| 14 | 199 | 295 | 1.48 | 0.57 |
| 15 | 200 | 298 | 1.49 | 0.43 |
| 16 | 213 | 485 | 2.28 | 0.66 |
| 17 | 297 | 482 | 1.62 | 0.39 |
| 18 | 311 | 413 | 1.33 | 0.54 |

As for the small diagrams, UML-`Kandinsky` outperforms SugiBib clearly in terms of running-time, crossings and bends on association edges. And, as for small diagrams, UML-`Kandinsky` uses more area than SugiBib which is mostly due to the fact that the diagrams have a greater height. However, the running time of UML-`Kandinsky` is not satisfactory on some of these diagrams. This is not mainly caused by the size of the diagrams, it is because these diagrams have a lot of crossings. SugiBib performs even worse for these diagrams.

## 7.4 Planarization

In this section we evaluate the algorithm for mixed-upward planarization from Section 3. We first study the performance of the algorithm for class diagrams and then compare it to the hierarchical approach.

### 7.4.1 Class Diagrams

In this section we study how the algorithm for mixed upward planarization behaves for the class diagrams test set.

In our experiments on class diagrams rerouting did not affect the number of crossings in the drawings significantly. However, the number of iterations of the mixed-upward-planar subgraph algorithm did have an effect on the number of crossings in the drawing as we can see in Figure 7.4. Choosing an even higher number of iterations did not further decrease the crossing number. The number of directed edges removed from the input graph to obtain a mixed upward planar graph is at most 16 for all graphs with up to 80 nodes.

### 7.4.2 Directed Graphs

In this section we present the results of an experimental comparison of our algorithm to the hierarchical approach. For the experiments we used a randomized version of GT which takes the largest subgraph from 150 different node orderings. In the experiments we did not use rerouting.

We compare our algorithm to the implementation of the hierarchical approach in yFiles. This implementation uses a randomized version of the iterated barycenter method for crossing minimization. This method was the clear winner of an experimental comparison of heuristics for the crossing minimization problem of layered graph [78]. We performed our experiments on three test sets Directed Rome Graphs, Upward Planar Graphs and Graphs With Limited Height. Figure 7.5 shows the relation between the average number of crossings and the number of vertices. Figure 7.6 shows the relation between average running time and the number of vertices. For the test sets with density 1.3 our algorithm yields better results than the hierarchical approach. In the case of limited height graphs, the improvements are considerable. For the test sets with density 2.6 the hierarchical approach is the clear winner. In terms of running time, the hierarchical approach clearly outperforms our approach, however the running time of our algorithm is still acceptable for interactive use.

| Nr | Time | Cross. | Bends A | Bends T | Area | Width | Height |
|---:|---:|---:|---:|---:|---:|---:|---:|
| 1 | 822 | 0 | 0 | 160 | 170624 | 688 | 248 |
| 2 | 1118 | 0 | 64 | 153 | 44940 | 321 | 140 |
| 3 | 1732 | 49 | 25 | 142 | 59555 | 215 | 277 |
| 4 | 2136 | 21 | 93 | 409 | 56875 | 455 | 125 |
| 5 | 4676 | 78 | 8 | 457 | 134385 | 465 | 289 |
| 6 | 112220 | 2545 | 0 | 1164 | 384930 | 658 | 585 |
| 7 | 2382 | 22 | 85 | 341 | 108962 | 362 | 301 |
| 8 | 5175 | 126 | 20 | 287 | 119600 | 260 | 460 |
| 9 | 9153 | 141 | 169 | 759 | 135120 | 563 | 240 |
| 10 | 15656 | 546 | 238 | 991 | 208278 | 406 | 513 |
| 11 | 17908 | 347 | 444 | 917 | 402476 | 842 | 478 |
| 12 | 22831 | 610 | 533 | 1190 | 697161 | 827 | 843 |
| 13 | 13874 | 278 | 381 | 852 | 337086 | 549 | 614 |
| 14 | 20023 | 669 | 459 | 1057 | 527646 | 739 | 714 |
| 15 | 18688 | 213 | 295 | 927 | 476064 | 696 | 684 |
| 16 | 138268 | 2415 | 1033 | 3019 | 2035500 | 1380 | 1475 |
| 17 | 148309 | 1588 | 1152 | 2322 | 2050353 | 1489 | 1377 |
| 18 | 42230 | 352 | 167 | 703 | 835315 | 905 | 923 |

(a) Results of UML-`Kandinsky`

| Nr | Time | Cross. | Bends | Area | Width | Height |
|---:|---:|---:|---:|---:|---:|---:|
| 1 | 5143 | 0 | 0 | 93391 | 1531 | 61 |
| 2 | 5178 | 290 | 104 | 21024 | 438 | 48 |
| 3 | 7227 | 359 | 187 | 28080 | 585 | 48 |
| 4 | 8025 | 540 | 148 | 43575 | 1245 | 35 |
| 5 | 6234 | 433 | 84 | 91860 | 1531 | 60 |
| 6 | 129709 | 7032 | 293 | 75384 | 698 | 108 |
| 7 | 7081 | 213 | 141 | 54675 | 729 | 75 |
| 8 | 6127 | 503 | 128 | 30030 | 455 | 66 |
| 9 | 8246 | 938 | 250 | 105750 | 1175 | 90 |
| 10 | 60237 | 3767 | 374 | 37202 | 418 | 89 |
| 11 | 27794 | 2885 | 360 | 139285 | 1565 | 89 |
| 12 | 23095 | 3806 | 518 | 138067 | 1367 | 101 |
| 13 | 19031 | 2972 | 414 | 248430 | 1690 | 147 |
| 14 | 21877 | 5043 | 494 | 200760 | 1673 | 120 |
| 15 | 95813 | 2386 | 522 | 242136 | 2124 | 114 |
| 16 | 1995719 | 36179 | 786 | 515430 | 1242 | 415 |
| 17 | 264264 | 9721 | 791 | 303096 | 2076 | 146 |
| 18 | 109391 | 8066 | 749 | 456741 | 2671 | 171 |

(b) Results of SugiBib

Figure 7.3: Test results for diagrams with more than 80 classes.

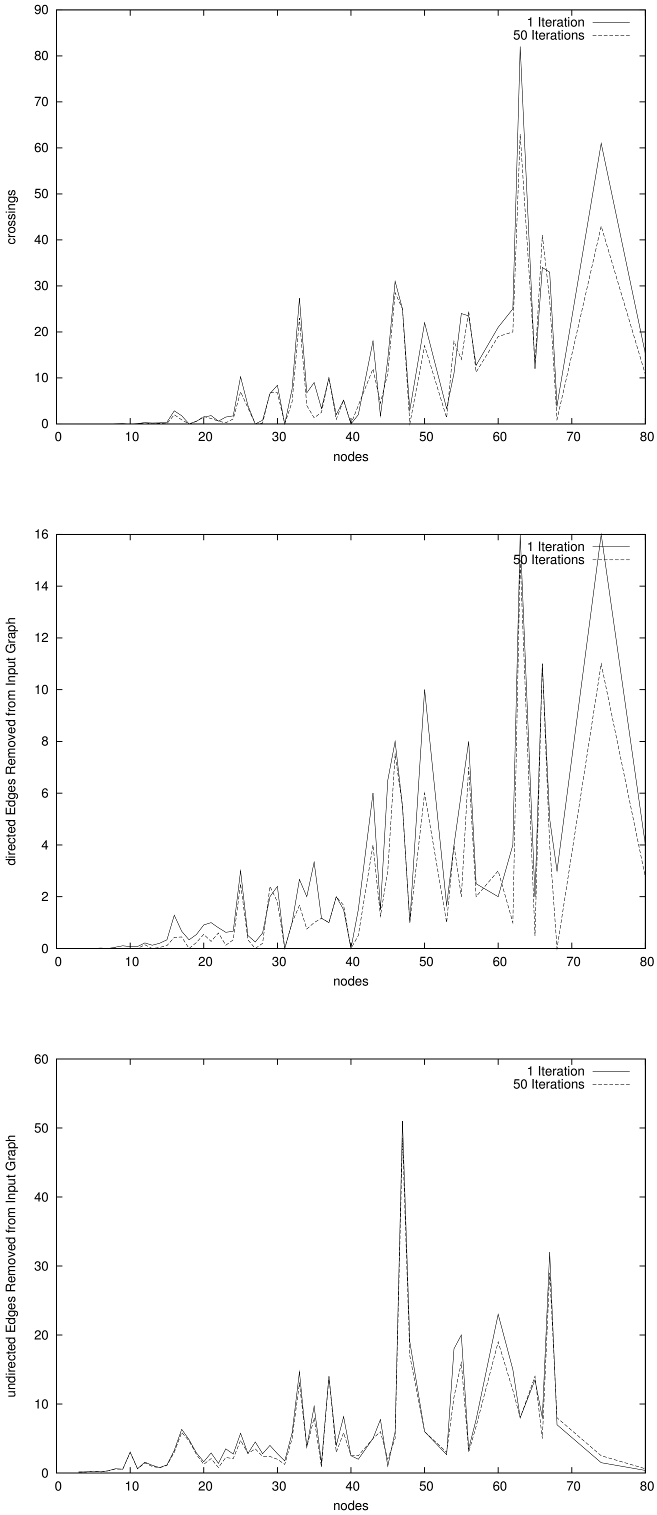
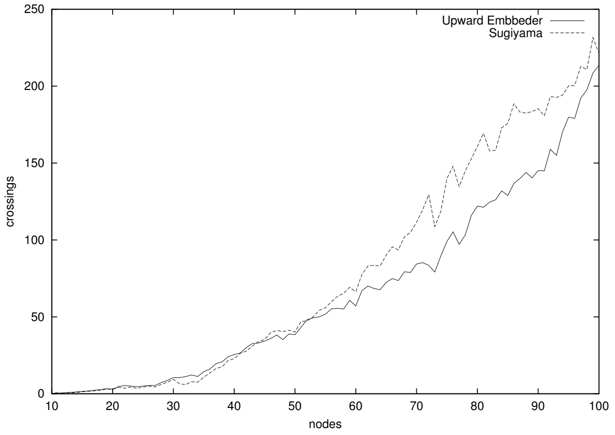
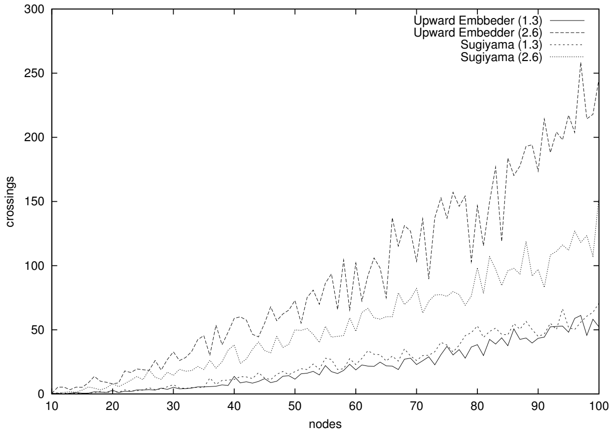


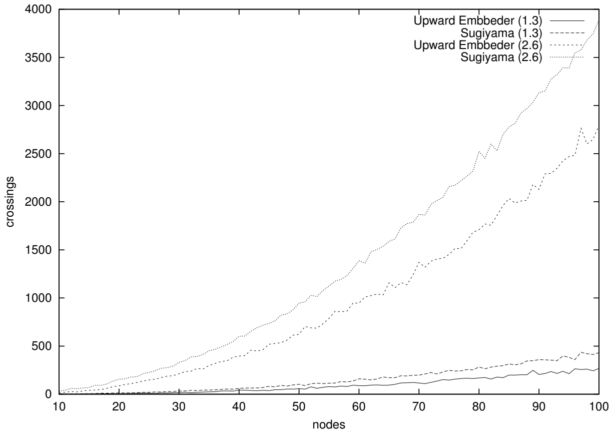
Figure 7.4: Results of experiments on class diagrams with up to 80 classes.



(a) Directed Rome Graphs

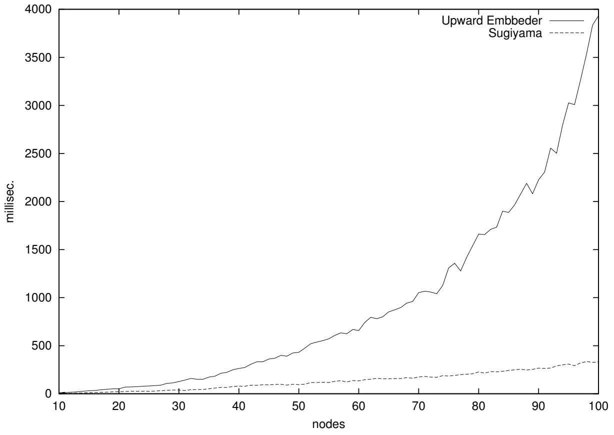


(b) Upward Planar Graphs

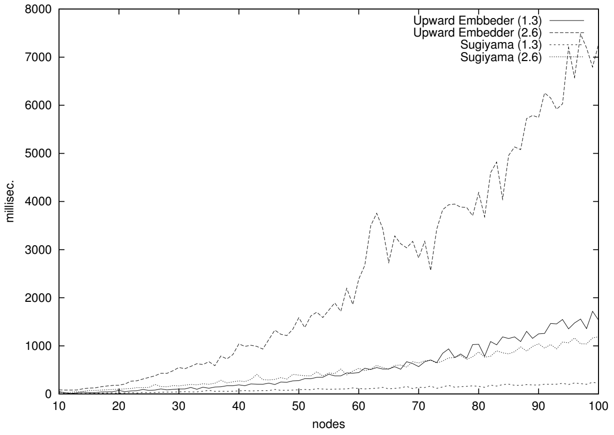


(c) Limited Height Graphs

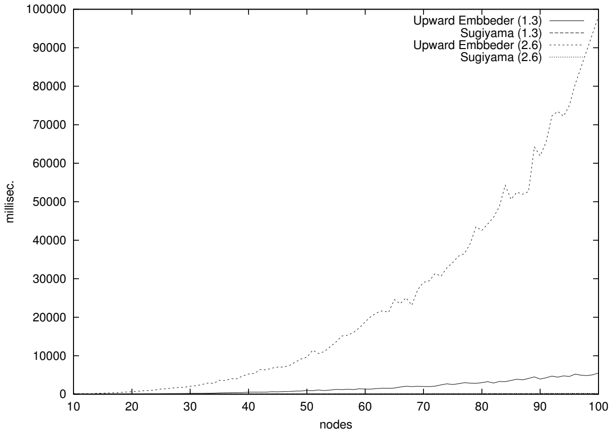
Figure 7.5: Experiments on directed graphs: Number of crossings.



(a) Rome Graphs



(b) Upward Planar Graphs



(c) Limited Height Graphs

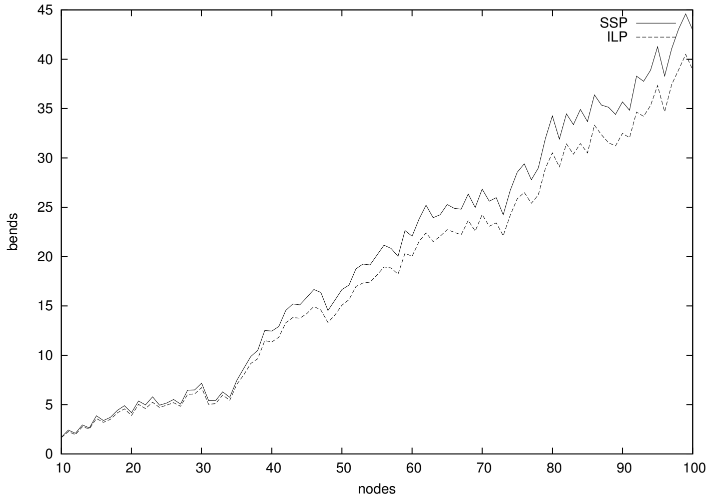
Figure 7.6: Experiments on directed graphs: Running time in milliseconds.

## 7.5 Orthogonalization

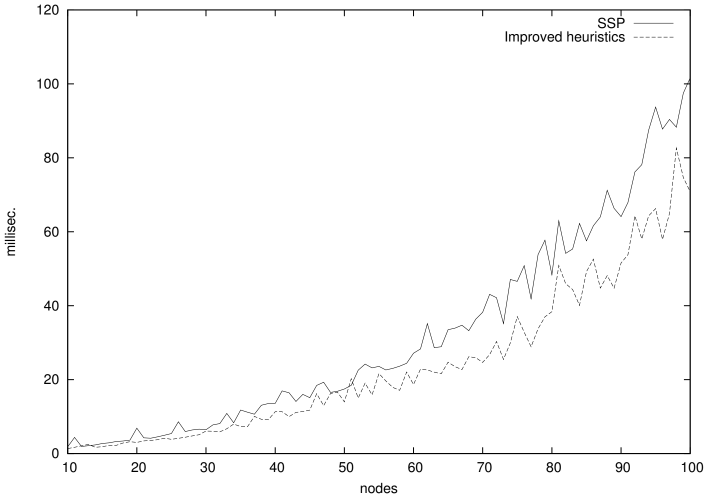
In this section we compare the improved heuristic with the SSP algorithm and the optimal solution for KANDINSKY BEND MINIMIZATION problem. We used the Rome graphs as test set. The optimal solutions have been computed with the help of an integer linear program solver. The running time of the solver is prohibitive for practical use but nevertheless acceptable for testing purposes.

The improved heuristic performed very well on the test set, it never produced solutions with more than 2 additional bends compared to the optimal solution. On 13 instances it produced exactly 2 additional bends, on 392 instances it produced 1 additional bend, the remaining over 10000 instances were solved to optimality.

The SSP algorithm performed worse, its average performance relative to the optimal solution is shown below.

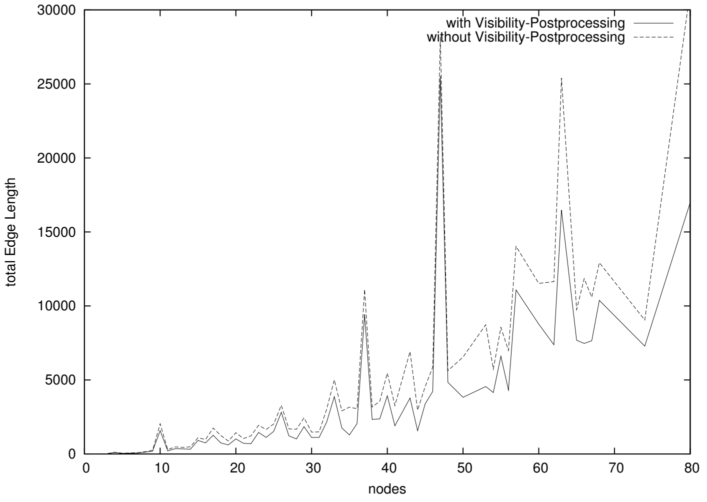


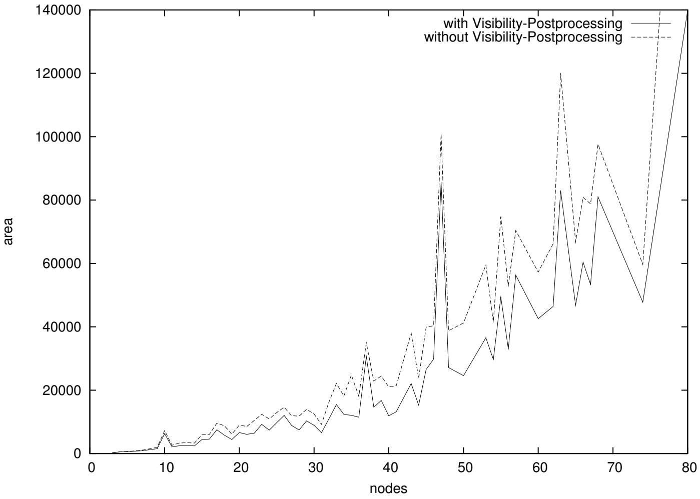
Even the running time of the algorithm is worse than the running time of the improved heuristic. This is partly due to the fact that the implementation of the improved heuristic had been carefully tuned in our case. Nevertheless it indicates that the improved heuristic is also competitive with respect to running time to the SSP algorithm although it has a higher worse case running time.

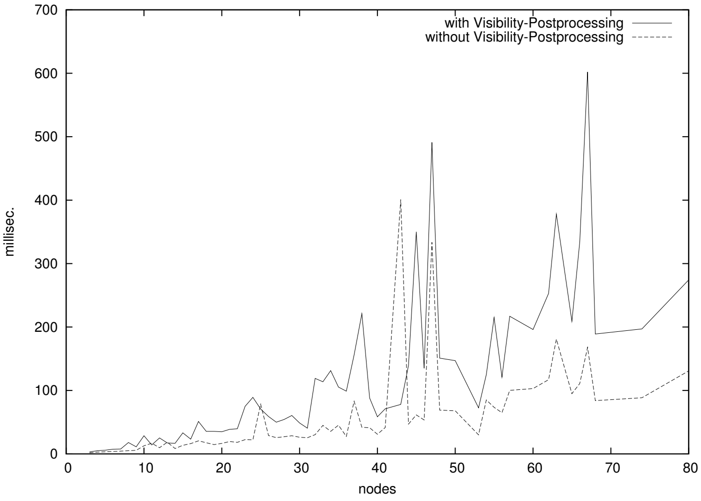


## 7.6 Compaction

In this section we present the results of an experimental evaluation of the compaction algorithm. We executed the compaction algorithm with and without visibility postprocessing on the class diagram test data. The running time of the algorithm was always below one second. While visibility postprocessing increases the running time, it improves significantly the total edge length and the area consumption of the drawings. Since the increase in time is not huge compared to the total running time of the UML-`Kandinsky` algorithm, it is reasonable to use visibility postprocessing.







## 7.7 Examples

In this section we present some example drawings produced by UML-`Kandinsky`. The example diagrams are automatically generated. Each of them visualizes a package of the Batik project as discussed in Section 7.2.

Figure 7.7 shows a diagram with a balanced number of associations vs. generalizations. It contains a lot of connected components which are arranged by a floor planning algorithm. Apparently UML-`Kandinsky` performs very well on this example, it produces no crossings and few bends. Figure 7.8 shows a diagram with a deep inheritance hierarchy and few associations. Again UML-`Kandinsky` performs very well on this example, it produces no crossings and few bends. The diagram in Figure 7.9 contains no associations and the classes have a rich signature which results in large rectangles for representing the classes. The example shows that

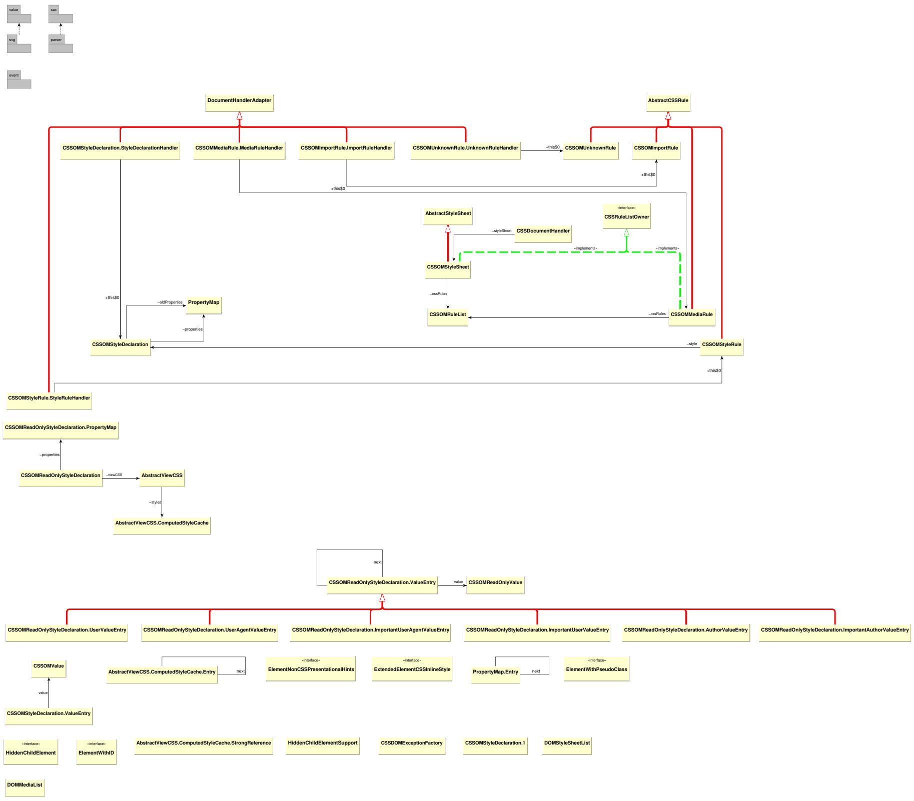

value
sac
svg
parser
event
DocumentHandlerAdapter
AbstractCSSRule
CSSOMStyleDeclaration.StyleDeclarationHandler
CSSOMMediaRule.MediaRuleHandler
CSSOMImportRule.ImportRuleHandler
CSSOMUnknownRule.UnknownRuleHandler
+this$0
CSSOMUnknownRule
CSSOMImportRule
+this$0
+this$0
AbstractStyleSheet
«interface»
CSSRuleListOwner
~styleSheet
CSSDocumentHandler
«implements»
«implements»
CSSOMStyleSheet
~cssRules
+this$0
~oldProperties
PropertyMap
~properties
CSSOMRuleList
~cssRules
CSSOMMediaRule
CSSOMStyleDeclaration
~style
CSSOMStyleRule
+this$0
CSSOMStyleRule.StyleRuleHandler
CSSOMReadOnlyStyleDeclaration.PropertyMap
~properties
CSSOMReadOnlyStyleDeclaration
~viewCSS
AbstractViewCSS
~styles
AbstractViewCSS.ComputedStyleCache
next
CSSOMReadOnlyStyleDeclaration.ValueEntry
value
CSSOMReadOnlyValue
CSSOMReadOnlyStyleDeclaration.UserValueEntry
CSSOMReadOnlyStyleDeclaration.UserAgentValueEntry
CSSOMReadOnlyStyleDeclaration.ImportantUserAgentValueEntry
CSSOMReadOnlyStyleDeclaration.ImportantUserValueEntry
CSSOMReadOnlyStyleDeclaration.AuthorValueEntry
CSSOMReadOnlyStyleDeclaration.ImportantAuthorValueEntry
CSSOMValue
AbstractViewCSS.ComputedStyleCache.Entry
next
«interface»
ElementNonCSSPresentationalHints
«interface»
ExtendedElementCSSInlineStyle
PropertyMap.Entry
next
«interface»
ElementWithPseudoClass
value
CSSOMStyleDeclaration.ValueEntry
«interface»
HiddenChildElement
«interface»
ElementWithID
AbstractViewCSS.ComputedStyleCache.StrongReference
HiddenChildElementSupport
CSSDOMExceptionFactory
CSSOMStyleDeclaration.1
DOMStyleSheetList
DOMMediaList

Figure 7.7: Class diagram of the package `org.apache.batik.css`. A typical example for an UML-`Kandinsky` drawing.

UML-`Kandinsky` handles diagrams with varying class sizes very well. In Figure 7.10 a diagram is depicted which contains some reflective associations and multiple relations between two classes. Again UML-`Kandinsky` can handle this type of input and produces a layout without crossings. Figure 7.11 shows a diagram for which UML-`Kandinsky` does not performs very well. It contains a complicated inheritance structure which is highly non-planar and therefore a lot of crossings are produced. For this type of diagrams the hierarchical approach seems more appropriate.

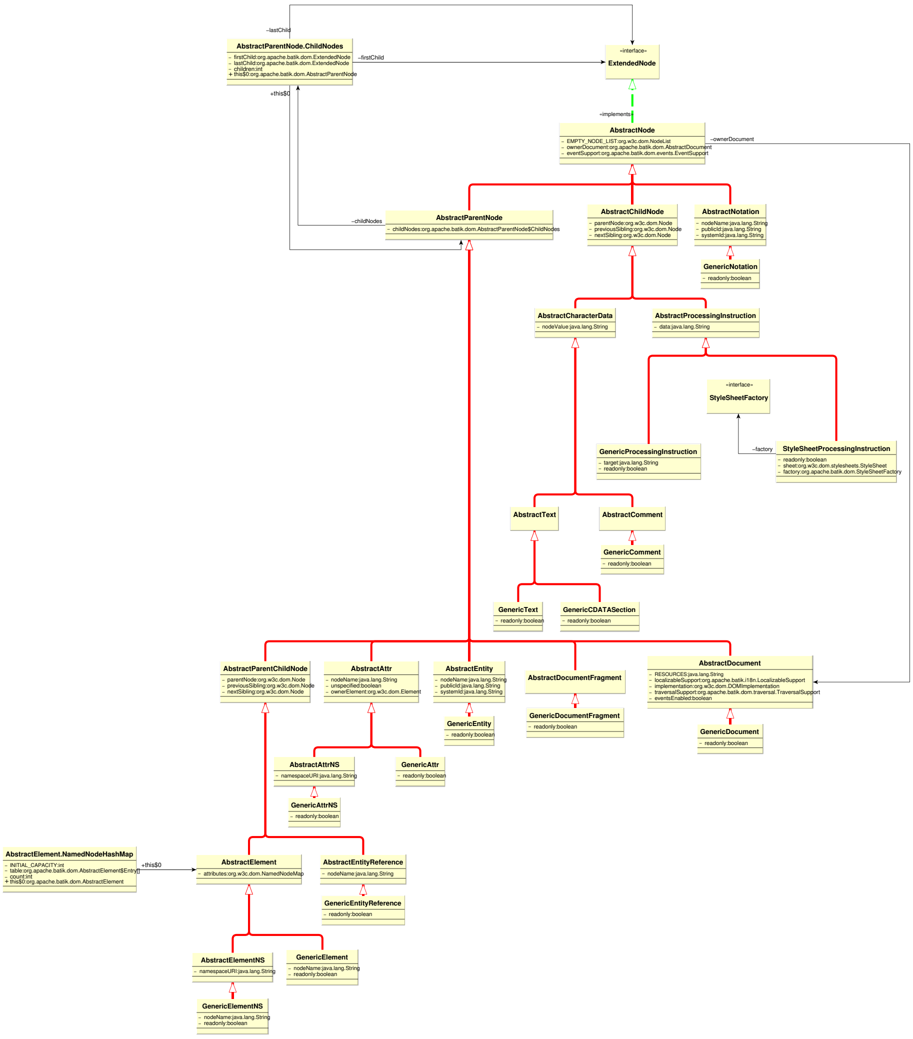

–lastChild
AbstractParentNode.ChildNodes
– firstChild:org.apache.batik.dom.ExtendedNode
– lastChild:org.apache.batik.dom.ExtendedNode
– children:int
+ this$0:org.apache.batik.dom.AbstractParentNode
–firstChild
«interface»
ExtendedNode
+this$0
«implements»
AbstractNode
– EMPTY_NODE_LIST:org.w3c.dom.NodeList
– ownerDocument:org.apache.batik.dom.AbstractDocument
– eventSupport:org.apache.batik.dom.events.EventSupport
–ownerDocument
–childNodes
AbstractParentNode
– childNodes:org.apache.batik.dom.AbstractParentNode$ChildNodes
AbstractChildNode
– parentNode:org.w3c.dom.Node
– previousSibling:org.w3c.dom.Node
– nextSibling:org.w3c.dom.Node
AbstractNotation
– nodeName:java.lang.String
– publicId:java.lang.String
– systemId:java.lang.String
GenericNotation
– readonly:boolean
AbstractCharacterData
– nodeValue:java.lang.String
AbstractProcessingInstruction
– data:java.lang.String
«interface»
StyleSheetFactory
GenericProcessingInstruction
– target:java.lang.String
– readonly:boolean
–factory
StyleSheetProcessingInstruction
– readonly:boolean
– sheet:org.w3c.dom.stylesheets.StyleSheet
– factory:org.apache.batik.dom.StyleSheetFactory
AbstractText
AbstractComment
GenericComment
– readonly:boolean
GenericText
– readonly:boolean
GenericCDATASection
– readonly:boolean
AbstractParentChildNode
– parentNode:org.w3c.dom.Node
– previousSibling:org.w3c.dom.Node
– nextSibling:org.w3c.dom.Node
AbstractAttr
– nodeName:java.lang.String
– unspecified:boolean
– ownerElement:org.w3c.dom.Element
AbstractEntity
– nodeName:java.lang.String
– publicId:java.lang.String
– systemId:java.lang.String
AbstractDocumentFragment
AbstractDocument
– RESOURCES:java.lang.String
– localizableSupport:org.apache.batik.i18n.LocalizableSupport
– implementation:org.w3c.dom.DOMImplementation
– traversalSupport:org.apache.batik.dom.traversal.TraversalSupport
– eventsEnabled:boolean
GenericEntity
– readonly:boolean
GenericDocumentFragment
– readonly:boolean
GenericDocument
– readonly:boolean
AbstractAttrNS
– namespaceURI:java.lang.String
GenericAttr
– readonly:boolean
GenericAttrNS
– readonly:boolean
AbstractElement.NamedNodeHashMap
– INITIAL_CAPACITY:int
– table:org.apache.batik.dom.AbstractElement$Entry[]
– count:int
+ this$0:org.apache.batik.dom.AbstractElement
+this$0
AbstractElement
– attributes:org.w3c.dom.NamedNodeMap
AbstractEntityReference
– nodeName:java.lang.String
GenericEntityReference
– readonly:boolean
AbstractElementNS
– namespaceURI:java.lang.String
GenericElement
– nodeName:java.lang.String
– readonly:boolean
GenericElementNS
– nodeName:java.lang.String
– readonly:boolean

Figure 7.8: Class diagram of the package `org.apache.batik.dom`. This example has a deep inheritance hierarchy and few associations.

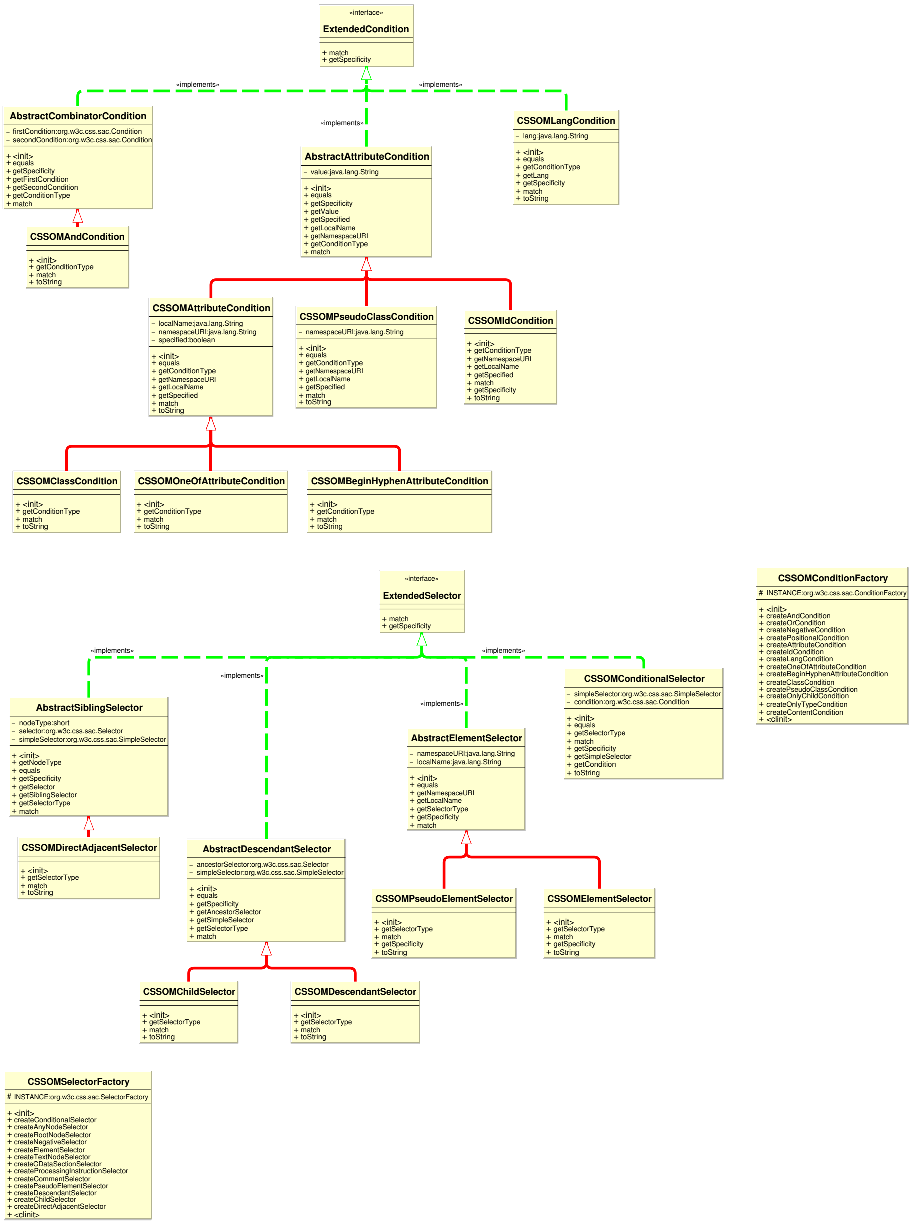


Figure 7.9: Class diagram of package `org.apache.batik.css.sac`. In this diagram classes have varying size.

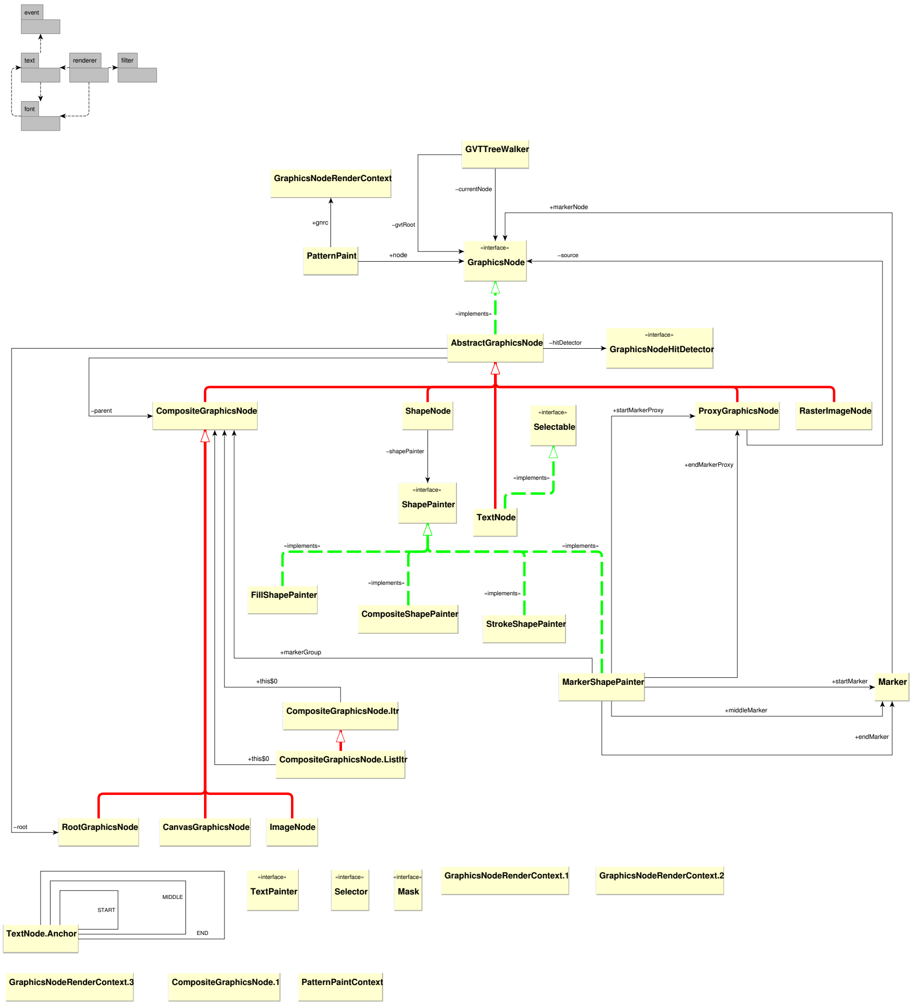


Figure 7.10: Class diagram of the package `org.apache.batik.gvt`. In this diagram the number of associations is dominating the number of generalizations.

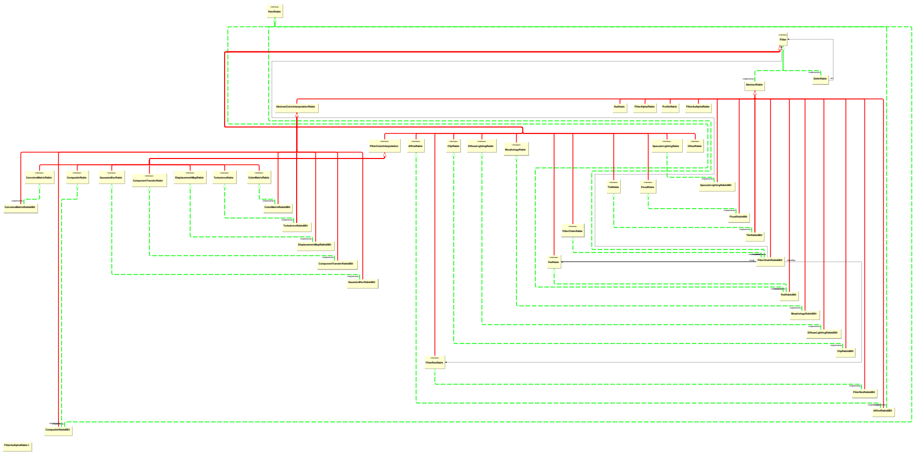


Figure 7.11: Class diagram of the package `org.apache.batik.awt.image.renderable`. The large number of generalizations lead to a large number of crossings from which the layout of this diagram suffers.

# Chapter 8

# Conclusion

In this work we presented UML-`Kandinsky` a new automatic layout algorithm for UML class diagrams which is based on the topology-shape-metrics approach.

An example layout of UML-`Kandinsky` is shown in Figure 8.1. It is the same diagram as in Figure 2.16. The drawing computed by SugiBib, the hierarchical approach of Seemann and Eichelberger, contains over twenty crossings while the drawing computed by UML-`Kandinsky` contains no crossings.

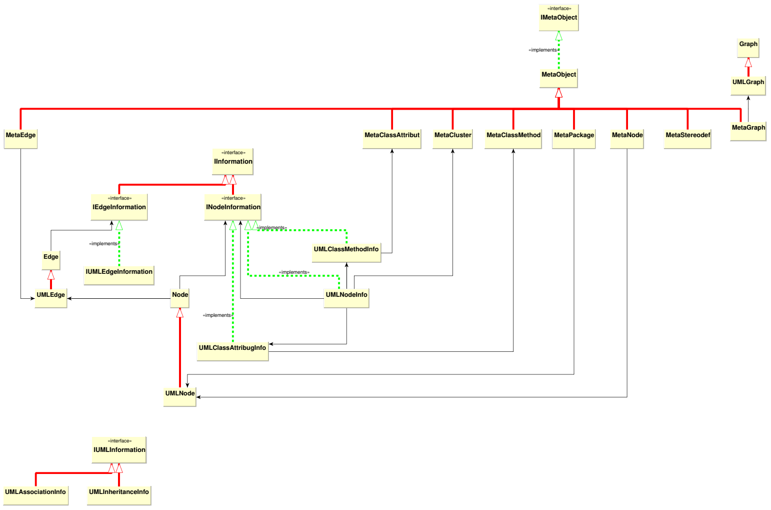


Figure 8.1: Example layout of UML-`Kandinsky` for the example in Figure 2.16.

From our theoretical results and from our empirical evaluation we can draw the conclusion that UML-`Kandinsky` is well suited for the automatic layout of sparse UML class diagrams of moderate size (up to 80 classes). We have shown that for these types of diagrams our algorithm is superior to algorithms based on the hierarchical approach like SugiBib. Diagrams found in documentation of software systems usually fall in this category. In object-oriented modeling it is encouraged to use multiple small diagrams instead of using few big diagram to model large systems [11, 12].

While UML-`Kandinsky` still produces satisfiable results for larger diagrams we think that other paradigms than the topology-shape-metrics approach are better suited for this type instances. The topology-shape-metrics approach tries to minimize crossings in the first place which usually has the effect that drawings need more area than drawings optimizing the AREA aesthetic criterion with higher priority. In our opinion the AREA aesthetic criterion is among the most important aesthetic criterion for large diagrams. However, there are no empirical evaluations of aesthetic criteria for large diagrams yet, until now only fairly small diagrams have been investigated.

Another aspect becoming increasingly important is the interactive layout of UML diagrams. Especially for the use in case tools it is indispensable that an automatic layout algorithm is interactive. We have proposed a new paradigm, the sketch-driven approach, for interactive layout. The advantages of this approach are that it is very robust and flexible since it only depends on a drawing of a subgraph of the input graph and does not keep state like conventional dynamic graph drawing algorithms. To our knowledge this is the first interactive orthogonal layout algorithm which supports dynamic changing graphs and user hints to improve the layout.

We will now give a short overview over the individual results of this work and discuss directions of future research.

## 8.1 Results

In this work we presented new techniques for the automatic layout of graphs using the topology-shape-metrics approach which, summarized in the algorithm UML-`Kandinsky` yield a new automatic layout algorithm for UML class diagrams. UML-`Kandinsky` has proven to be far superior to existing algorithms in a series of experiments.

The new results can be summarized as follows:

- In order to define the problem of automatic layout of UML class diagrams mathematically, we analyzed the structure of UML class diagrams, their notation and the aesthetic criteria which influence the readability of them. We analyzed existing automatic layout algorithms with respect to these criteria. and showed that these algorithms work only satisfactory if there is rich and deep structural information in the diagram, an assumption which is often violated in practise.

- We introduced the concept of mixed upward planarity, a generalization of planarity and upward planarity to mixed graphs. We presented an algorithm for constructing mixed upward planarizations. For the special case when the input graphs are directed, this is the first algorithm for constructing upward planarizations.

- We showed that the successive-shortest-path algorithm for the KANDINSKY BEND MINIMIZATION problem is not correct, which opens the question if there is a polynomial time algorithm for this problem.
- We presented a 2-approximation algorithm with running time $O(n^{\frac{7}{4}} \cdot \sqrt{\log n})$ for the KANDINSKY BEND MINIMIZATION problem, where $n$ is the size of the plane input graph. This is the first approximation algorithm for this problem.
- We extended the `Kandinsky` algorithm to consider constraints on angles and bends. We showed how we can use this extended version of the algorithm to formulate the special requirements on the layout of class diagrams.
- We presented the first linear time compaction algorithm for `Kandinsky` shapes which can handle prescribed vertex sizes. The algorithm unifies two existing algorithms for the compaction of point drawings and gives additional insight into how they work.
- We presented an algorithm for the interactive layout of orthogonal graph drawings. The algorithm is based on a new concept, the sketch-driven graph drawing approach. It produces a drawing which balances readability and change in the drawing. The sketch-driven approach has applications beyond interactive graph drawing for example the creation of schematic maps of ground plans.

All algorithms cited in this list have been implemented and results of experiments have been included proving their effectiveness.

## 8.2 Directions of Future Research

If we have a closer look on the problems and algorithms discussed in this work many questions comes to our mind, three of them seem to be especially important:

- Can we devise theoretical or practical better algorithms for the problems discussed in this work ?
- Can we include more advanced modeling constructs of the UML in this approach ?
- Can we apply the results of this work to other problem domains than UML class diagrams ?

In the following sections we will have a closer look on these questions.

### 8.2.1 Improvement of the Presented Algorithms

From a theoretical point of view there remain several questions open.

For the planarization of mixed graphs we used a layering approach to avoid directed cycles in the planarization. This layering produces sometimes artifacts and has a negative effect on the running time, since there may be a linear number of layers. This leads to an algorithm with running time $|V|^2|E|$ for sparse graphs, while in the undirected case the running time is $|V||E|$. It remains an open question if we can devise a faster algorithm for this problem, probably without using layering.

Regarding the KANDINSKY BEND MINIMIZATION problem, the main problem is to determine if it is solvable in polynomial time or NP-complete. Our conjecture is that there is a polynomial time algorithm, however it is not clear how it looks like. A promising direction is to analyze the polyhedron defined by the integer linear program of the `Kandinsky` network with the help of polyhedral combinatorics. A polynomial time separation algorithm for the faces of this polyhedron induce a polynomial time algorithm for the KANDINSKY BEND MINIMIZATION problem.

In this case it is interesting if the CONSTRAINED KANDINSKY BEND MINIMIZATION problem is also solvable in polynomial time. In the case that the KANDINSKY BEND MINIMIZATION problem is NP-hard, the CONSTRAINED KANDINSKY BEND MINIMIZATION is also NP-hard since it is a generalization of it. In this case it is of interest if we can devise approximation algorithms for these problems with either better approximation factors or better running times.

Regarding the compaction phase it is promising to investigate new post-processing algorithms or adaptions of known post-processing algorithms, like the 4M algorithm [55], to prescribed vertex-size drawings. In some cases the area is used very inefficiently by the algorithm. Often this is an effect of the planarization created in the first phase of the algorithm and using a slightly different planarization would yield a much more compact drawing.

For interactive layout an issue we have not addressed in this work is compaction. In our implementation we use the standard compaction algorithm. However, sometimes it may be helpful to preserve some distances in the sketch drawing, especially in the user supplied constrained scenario. Furthermore it would be interesting to study other difference metrics than the angle and bend difference metrics we have used. For example the relative positions of nodes are not taken into account in this scenario, but is nevertheless important for preserving the mental map.

### 8.2.2 Advanced Modeling Constructs

Although we have tried to model almost all constructs of UML class diagrams, two notations of the UML are not captured by the class diagram

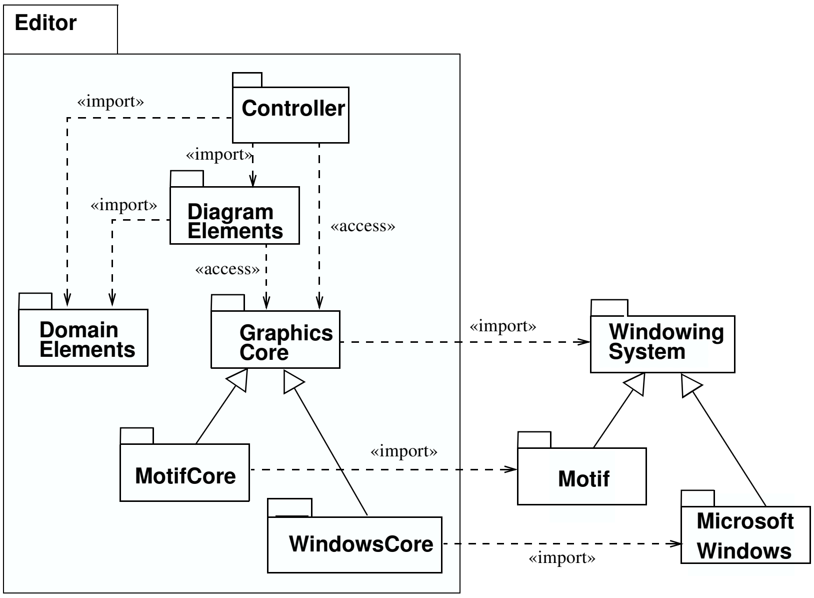


Figure 8.2: Example for clustering from [95].

graph of Chapter 2 and have not been considered in this work: clustering and edges between edges.

In clustering parts of the diagram are inside a single vertex. Clustering is used sometimes to emphasize that some classes or packages belong to a certain package, see Figure 8.2 for an example. Clustering can be modeled by adding a containment hierarchy to the class diagram graph. All three steps of the topology-shape-metrics approach have to be adapted to consider clustering. One possible way to incorporate clustering is to merge the concepts of c-planarity with mixed upward planarity. The concept of *c-planarity* [52] extends planarity to clustered graphs. A c-planarization algorithm together with a sketch for an orthogonalization algorithm is presented in [29]. In [90] details for an orthogonalization algorithm can be found. We have chosen to not consider clustering since it is not supported by most modeling tools and is therefore used rarely in the context of class diagrams. Most tools do only allow to manipulate a flat diagram.

Edges between edges, as illustrated in Figure 8.3, can not be modeled with graphs, therefore a model considering them cannot make use of the enormous amount of results in this field. Since edges between edges are only used rarely in the UML, it seems not to be worth to abandon the graph data model to capture a feature which is seldom used, and besides that, is not supported by most tools. We recommend to handle edges between edges in a postprocessing step in a combination of an edge routing and a label placement algorithm.

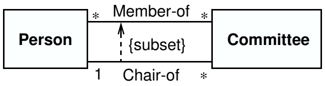


Figure 8.3: Example for an edge between edges from [95].

### 8.2.3 New Application Domains

This work contains various extensions of the topology-shape-metrics approach which enabled us to apply this approach to the automatic layout of UML class diagrams. However, the extensions can be used to devise automatic layout algorithms for other application domains with similar requirements.

One obvious application domain are other types of UML diagrams. The UML specifies several other diagram types than class diagrams: use case diagrams, component diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, activity diagrams, and deployment diagrams. To apply our methods to these diagram types we have first to analyze the aesthetic criteria which apply to these diagrams.

An interesting diagram type are activity diagrams. See Figure 8.4 for an example. As in class diagrams there are directed and undirected edges in activity diagrams and vertices in the diagrams have prescribed size. Therefore our new methods can be applied to these diagrams. However, activity diagrams are often layouted in *swim lanes*, see Figure 8.5 for an example. We have to extend our framework to consider swim lanes if we want to apply it to activity diagrams.

Another interesting application domain is the visualization of metabolic pathways. Diagrams of metabolic pathways contain also hyperedges and have direction information embedded, which are issues addressed by our new algorithms. The vertices in diagrams of metabolic pathways have usually prescribed size since they contain the name of a chemical compound which may be very long. Until now the hierarchical graph drawing paradigm is used to visualize diagrams of metabolic networks [110, 48].

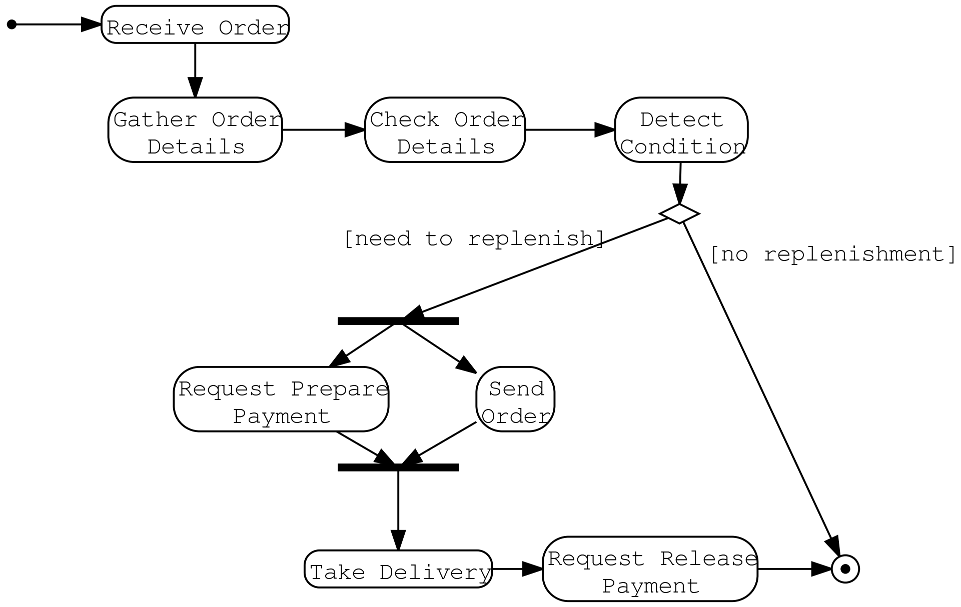


Figure 8.4: Example for an activity diagram for Supply-Chain-Management from [101].

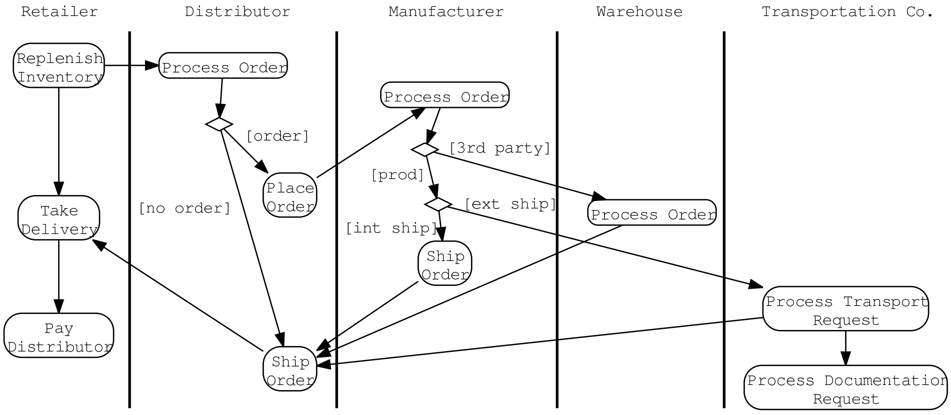


Figure 8.5: Example for an activity diagram with swim lanes from [101].

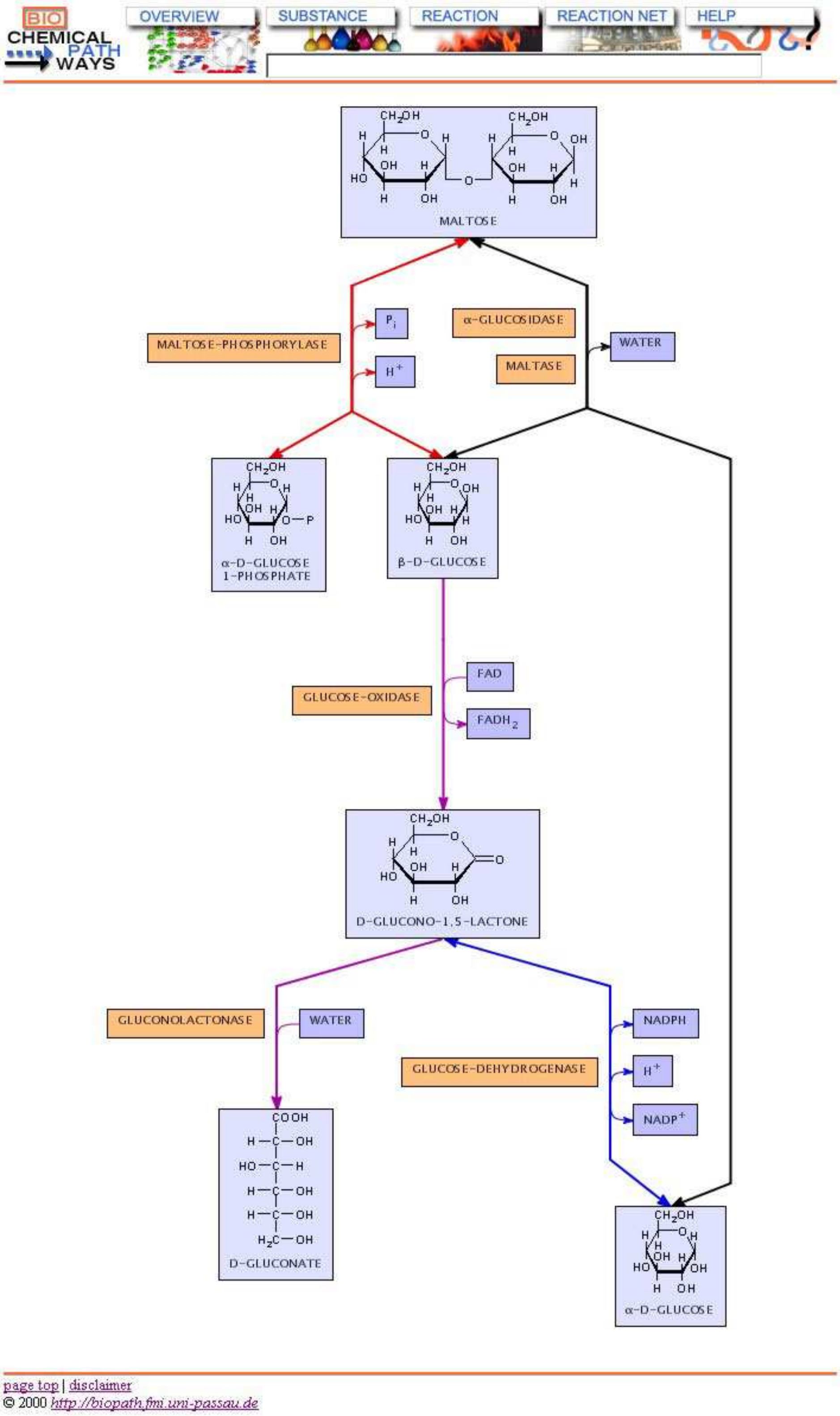


Figure 8.6: Example for a metabolic pathway diagram from [53].

# Bibliography

[1] S. S. Ahir. *UML in a nutshell.* O'Reilly, 1998.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[3] T. Asano, H. Imai, and A. Mukaiyama. Finding a maximum weight independent set of a circle graph. *IEICE Tranactions*, E74:681–683, 1991.

[4] O. Bastert and C. Matuszewski. Layered drawings of digraphs. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *LNCS Tutorial*, pages 104–139. Springer, 2001.

[5] C. Batini, E. Nardelli, and R. Tamassia. A layout algorithm for data flow diagrams. *IEEE Trans. Softw. Eng.*, SE-12(4):538–546, 1986.

[6] J. L. Bentley and T. A. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[7] P. Bertolazzi, G. D. Battista, and W. Didimo. Quasi-upward planarity. *Algorithmica*, 32(3):474–506, 2002.

[8] 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.

[9] C. Binucci, W. Didimo, G. Liotta, and M. Nonato. Labeling heuristics for orthogonal drawings. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2275 of *LNCS*, pages 139–153, 2001.

[10] C. Binucci, W. Didimo, G. Liotta, and M. Nonato. Computing labeled orthogonal drawings. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'2002)*, volume 2528 of *LNCS*, pages 66–73, 2002.

[11] G. Booch. *Object Oriented Design and Analysis.* Addison-Wesley, 1994.

[12] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[13] 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.

[14] U. Brandes. *Layout of Graph Visualizations*. PhD thesis, University of Konstanz, 1999.

[15] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, , and M. S. Marshall. Graphml progress report: Structural layer proposal. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2265, pages 501–512. Springer, 2002.

[16] 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'2002)*, volume 2528, pages 1–12. Springer, 2002.

[17] U. Brandes and D. Wagner. A Bayesian paradigm for dynamic graph layout. In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 236–247, 1997.

[18] U. Brandes and D. Wagner. Dynamic grid embedding with few bends and changes. In *Proceedings of the 9th Annual International Symposium on Algorithms and Computation (ISAAC'98)*, volume 1533 of *LNCS*, pages 89–98, 1998.

[19] J. Branke. Dynamic graph drawing. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, LNCS Tutorial, pages 228–246. Springer, 2001.

[20] 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*, 1999. To appear.

[21] S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara. INTERACTIVEGIOTTO: An algorithm for interactive orthogonal graph drawing. In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 303–308, 1997.

[22] S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. *Journal of Graph Algorithms and Applications*, 4(3):47–74, 2000.

[23] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):467–493, 1994.

[24] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.

[25] 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.

[26] M. Closson, S. Gartshore, J. Johansen, and S. Wismath. Fully dynamic 3-dimensional orthogonal graph drawing. *Journal of Graph Algorithms and Applications*, 5(2):1–35, 2001.

[27] E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[28] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[29] G. Di Battista, W. Didimo, and A. Marcandalli. Planarization of clustered graphs. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2275 of *LNCS*, pages 60–74, 2001.

[30] G. Di Battista, W. Didimo, M. Patrignani, and M.Pizzonia. Drawing database schema with DBdraw. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, LNCS, pages 451–452. Springer, 2001.

[31] G. Di Battista, W. Didimo, M. Patrignani, and M.Pizzonia. Drawing relational schema. *Software Practice and Experience*, to appear.

[32] 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 *LNCS*, pages 297–310. Springer, 1999.

[33] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[34] 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.

[35] G. Di Battista, W.-P. Liu, and I. Rival. Bipartite graphs, upward drawings, and planarity. *Information Processing Letters*, 36(6):317–322, 1990.

[36] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In *Proceedings of the 17th International Colloqium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 598–611, 1990.

[37] W. Didimo, M. Patrignani, and M.Pizzonia. Industrial plant drawer. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, LNCS, pages 475–476. Springer, 2001.

[38] R. Diestel. *Graph Theory.* Springer, 2nd edition, 2000.

[39] H. do Nascimento and P. Eades. User hints for directed graph drawing. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2265, pages 124–138. Springer, 2001.

[40] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. *Proc. Compugraphics '91*, pages 24–33, 1991.

[41] H. Eichelberger. Automatisches Zeichnen von UML Klassendiagrammen mit dem Sugiyama-Algorithmus in Tagungsband des GI-Workshops Softwarevisualisierung 2000. Technical Report A/01/2000, Universität des Saarlandes, 2000.

[42] H. Eichelberger. Aesthetics of class diagrams. In *Proc. of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis, Vissoft 2002*, pages 23–31, 2002.

[43] H. Eichelberger. Evaluation-report on the layout facilities of uml tools. Technical Report 298, Lehrstuhl für Informatik II, Universität Würzburg, 2002.

[44] H. Eichelberger. Nice class diagrams admit good design ? In *Proceedings of ACM 2003 Symposium on Software Visualization, SoftVis 2003*, pages 159–167. ACM, 2003.

[45] M. Eiglsperger, S. Fekete, and G. Klau. Orthogonal graph drawing. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, LNCS Tutorial, pages 121–171. Springer, 2001.

[46] M. Eiglsperger, U. Foessmeier, and M. .Kaufmann. Orthogonal graph drawing with constraints. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 3–11, 2000.

[47] M. Eiglsperger and M. Kaufmann. An approach for mixed upward planarization. In *Proceedings of the 7th International Workshop on Algorithms and Datastructures (WADS 2001)*, pages 352–364. Springer, 2001.

[48] M. Eiglsperger and M. Kaufmann. Visualization of biodegradation pathways in the um-bbd. In *Currents in Computational Molecular Biology*, pages 223–224. Les Publications CRM, 2001.

[49] 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, 2002.

[50] M. Eiglsperger, M. Kaufmann, and F. Eppinger. An approach for mixed upward planarization. *Journal of Graph Algorithms and Applications*, 7(2):203–220, 2003.

[51] 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, 2003.

[52] 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 *LNCS*, pages 213–226. Springer, 1995.

[53] M. Forster, A. Pick, M. Raitner, F. Schreiber, and F. Brandenburg. The system architecture of the biopath system. *Silicio Biology*, 2(3):415–426, 2002.

[54] U. Fößmeier. *Orthogonale Visualisierungstechniken für Graphen.* PhD thesis, Eberhard-Karls-Universität zu Tübingen, 1997.

[55] U. Fößmeier, C. Heß, and M. Kaufmann. On improving orthogonal drawings: The 4M-algorithm. In S. H. Whitesides, editor, *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *LNCS*, pages 125–137. Springer, 1998.

[56] U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F. J. Brandenburg, editor, *Proceedings of the 3rd International Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 254–266. Springer, 1996.

[57] U. Fößmeier and M. Kaufmann. Algorithms and area bounds for nonplanar orthogonal drawings. In G. Di Battista, editor, *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 134–145. Springer, 1997.

[58] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 2nd edition, 1999.

[59] K. Freiwalds, U. Dogrusoz, and P. Kikusts. Disconnected graph layout and the polyomino packing approach. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2265 of *LNCS*, pages 378–391. Springer, 2001.

[60] 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.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[62] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[63] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$–Completeness.* W.H. Freeman & Co, 1979.

[64] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312–316, 1983.

[65] A. Garg and R. Tamassia. On the complexity of upward and rectilinear planarity testing. In *Proceedings of the 2nd International Symposium on Graph Drawing (GD'94)*, volume 894 of *LNCS*, pages 286–297, 1995.

[66] 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 *LNCS*, pages 201–216, 1997.

[67] N. Gelfand and R. Tamassia. Algorithmic patterns for orthogonal graph drawing. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *LNCS*, pages 138–152. Springer, 1998.

[68] O. Goldschmidt and A. Takvorian. An efficient graph planarization two-phase heuristic. *Networks*, 24:69–73, 1994.

[69] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. Caesar automatic layout of uml class diagrams. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2265 of *LNCS*. Springer, 2002.

[70] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A new approach for visualizing uml class diagrams. In *Proceedings of ACM Symposium on Software Visualization'2003*. ACM, to appear.

[71] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. In *Proceedings of the Twelfth ACM-SIAM Symposium on Discrete Algorithms, (SODA '2001)*, pages 246–255. ACM Press, 2001.

[72] P. Healy and N. Nikolov. A branch-and-cut approach to the directed acyclic graph layering problem. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'2002)*, volume 2528 of *LNCS*, pages 98–109. Springer, 2002.

[73] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 1974.

[74] M. D. .Hutton and A. Lubiw. Upward planar drawing of single source acyclic digraphs. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA'91)*, pages 203–211, 1991.

[75] J. Ignatowicz. Drawing force-directed graphs using Optigraph. In F. J. Brandenburg, editor, *Proceedings of the 3rd International Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 333–336. Springer, 1996.

[76] R. Jayakumar, K. Thulasiraman, and M. N. S. Swamy. $o(n^2)$ algorithms for graph planarization. In *IEEE Trans. on CAD*, volume 8 (3), pages 257–267, 1989.

[77] M. Jünger and P. Mutzel. Solving the maximum weight planar subgraph problem by branch & cut. In *Proc. of the 3rd conference on integer programming and combinatorial optimization (IPCO)*, pages 479–492, 1993.

[78] 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.

[79] G. Kant. An $O(n^2)$ maximal planarization algorithm based on PQ-trees. Technical Report RUU-CS-92-03, CS Dept., Univ. Utrecht, Netherlands, 1992.

[80] R. Karp. Reducibility among combinatorical problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[81] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models.* LNCS Tutorial. Springer, 2001.

[82] G. W. Klau, K. Klein, and P. Mutzel. An experimental comparison of orthogonal compaction algorithms. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'2000)*, number 1984 in LNCS, pages 37–51, 2001.

[83] G. W. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report 98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[84] G. W. Klau and P. Mutzel. Combining graph labeling and compaction. In J. Kratochvil, editor, *Proceedings of the 7th International Symposium on Graph Drawing (GD'99)*, number 1731 in LNCS, pages 27–37. Springer, 1999.

[85] G. W. Klau and P. Mutzel. Optimal compaction of orthogonal grid drawings. In *Integer Programming and Combinatorial Optimization (IPCO'99)*, number 1610 in LNCS, pages 304–319, 1999.

[86] P. N. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *Proceedings ACM Symp. on Theory of Computing*, 1994.

[87] K. Kuratowski. Sur le problme des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[88] U. Lauther and A. Stübinger. Generating schematic cable plans using springembedder methods. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'2001)*, volume 2265 of *LNCS*, pages 465–466. Springer, 2001.

[89] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Applicable Theory in Computer Science. Wiley-Teubner, 1990.

[90] D. Lütke-Hüttemann. Knickminimales zeichnen 4-planarer clustergraphen. Master's thesis, Universität des Saarlands, 1999.

[91] T. Masui. Graphic object layout with interactive genetic algorithms. In *Proceedings of the 1992 IEEE Workshop on Visual Languages (VL '92)*, pages 74–87, 1992.

[92] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, 1994.

[93] X. Mendonça and P. Eades. Learning aesthetics for visualization. In *Anais do XX Seminário Integrado de Software e Hardware*, pages 76–88, Florianópolis, Brazil, 1993.

[94] S. C. North. Incremental layout with DynaDag. In *Proceedings of the 3rd International Symposium on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418, 1996.

[95] OMG. Unified Modeling Language v1.4, 2001. `http://www.omg.org/technology/documents/formal/uml.htm`.

[96] A. Papakostas. Upward planarity testing of outerplanar dags. In *Proceedings of the DIMACS International Workshop on Graph Drawing (GD'94)*. Springer Lecture Notes in Computer Science 894, pages 298–306, 1994.

[97] A. Papakostas and I. G. Tollis. Issues in interactive orthogonal graph drawing. In *Proceedings of the 3rd International Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 419–430. Springer, 1995.

[98] G. Paris. Cooperation between interactive actions and automatic drawing in a schematic editor. In S. H. Whitesides, editor, *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *LNCS*, pages 394–402. Springer, 1998.

[99] 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.

[100] M. Patrignani. On the complexity of orthogonal compaction. *Computational Geometry: Theory and Applications*, 19(1):47–67, 2001.

[101] S. Polyak, J. Lee, M. Gruninger, and C. Menzel. Applying the process interchange format (pif) to a supply chain process interoperability scenario. In A. Gomez-Perez and R. Benjamins, editors, *Proceedings of the Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98*, 1998.

[102] 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 *LNCS*, pages 248–261, 1997.

[103] H. C. Purchase, J. A. Allder, and D. Carrington. User preference of graph layout aesthetics: A uml study. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'2000)*, volume 1984 of *LNCS*, pages 5–18. Springer, 2000.

[104] 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.

[105] H. C. Purchase, M. McGill, L. Colpoys, and D. Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study". In *Proceedings of the Australian Symposium on Information Visualization*, volume 9. Australian Computer Society Inc., 2001.

[106] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[107] M. Resende and C. Ribeiro. A grasp for graph planarization. *Networks*, 29:173–189, 1997.

[108] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts of planar graphs and bipolar orientations. *Discrete & Computational Geometry*, 1(4):342–351, 1986.

[109] K. Ryall, J. Marks, and S. Shieber. An interactive system for drawing graphs. In S. C. North, editor, *Proceedings of the 4th International Symposium on Graph Drawing (GD'96)*, volume 1190 of *LNCS*, pages 387–394. Springer, 1997.

[110] F. Schreiber. *Visualisierung biochemischer Reaktionsnetze*. PhD thesis, University of Passau, 2001.

[111] J. Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 415–424, 1997.

[112] M. Sirava, T. Schfer, M. .Eiglsperger, M. .Kaufmann, O. .Kohlbacher, E. .Bornberg-Bauer, and H.-P. Lenhof. Biominer - modeling, analyzing, and visualizing biochemical pathways and networks. *Bioinformatics*, 19(10):219–230, 2002.

[113] J. Six and I. Tollis. Circular drawings of biconnected graphs. In *Proceedings.of Alenex'99*, volume 1619 of *LNCS*, pages 57–73. Springer, 1999.

[114] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, February 1981.

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

[116] R. Tamassia and I. G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete & Computational Geometry*, 1(4):321–341, 1986.

[117] F. Wagner and A. Wolff. A combinatorial framework for map labeling. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, number 1547 in LNCS, pages 316–331. Springer, 1998.

[118] R. Wiese, M. Eiglsperger, and M. Kaufmann. yfiles: Visualization and automatic layout of graphs. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, LNCS, pages 453–454. Springer, 2001.

[119] R. Wiese, M. Eiglsperger, and M. Kaufmann. yfiles: Visualization and automatic layout of graphs. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*. Springer, to appear.

## Lebens- und Bildungsgang

| | |
|---|---|
| **Name:** | Markus Eiglsperger |
| **Familienstand:** | verheiratet |

| | |
|---|---|
| **13.06.1973** | geboren in Singen am Hohentwiel |
| **1979 - 1983** | Besuch der Grundschule in Öhningen |
| **1983 - 1992** | Besuch des Ambrosius-Blarer-Gymnasiums in Gaienhofen |
| **05/1992** | Abitur (Note: 1.4)<br>Leistungskurse: Mathematik und Physik |
| **07/1992 - 10/1993** | Zivildienst beim Mobilen Sozialen Hilfsdienst der Arbeiterwohlfahrt in Singen am Hohentwiel |
| **10/1993 - 12/1999** | Studium der Informatik mit Nebenfach Mathematik an der Eberhard-Karls-Universität Tübingen |
| **10/1995** | Vordiplom in Informatik, Nebenfach Mathematik (Note: sehr gut) |
| **09/1996 - 09/1997** | Auslandsstudium an der Université de Franche-Comté in Besançon, Frankreich |
| **06/1997 - 08/1997** | Praktikum bei der Hewlett-Packard GmbH in Böblingen:<br>*„Erstellung von Utlities zum Komprimieren von Software“* |
| **09/1997** | Maîtrise (Note: bien) |
| **07/1999** | Diplomarbeit (bei Prof. Kaufmann) im Fach Informatik (vgl.[46]):<br>*„Constraints im Kandinsky-Algorithmus“* |
| **11/1999** | Diplom in Informatik, Nebenfach Mathematik (Note: sehr gut) |
| **seit 12/1999** | Promotion an der Fakultät für Informatik, Universität Tübingen, Arbeitsbereich Paralles Rechnen (Prof. M. Kaufmann)<br>*Publikationen:* [15, 16, 45, 46, 47, 48, 49, 50, 51, 112, 118, 119] |