Sugiyama Layouts for Prescribed Drawing Areas

M.Sc. Ulf Rüegg

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2018

Kiel Computer Science Series (KCSS) 2018/1 v1.0 dated 2018-7-12

URN:NBN urn:nbn:de:gbv:8:1-zs-00000341-a4
ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via uruurumail@gmail.com

Published by the Department of Computer Science, Kiel University

Real-Time and Embedded Systems Group

Please cite as:

▻ Ulf Rüegg. *Sugiyama Layouts for Prescribed Drawing Areas* Number 2018/1 in Kiel Computer Science Series. Department of Computer Science, 2018. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Rueegg18,
  author    = {Ulf R{\"u}egg},
  title     = {Sugiyama Layouts for Prescribed Drawing Areas},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2018},
  number    = {2018/1},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering,
               Kiel University.}
}
```

© 2018 by Ulf Rüegg

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Reinhard von Hanxleden Institut für Informatik Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Prof. Dr. Petra Mutzel Fakultät für Informatik Technische Universität Dortmund

Datum der mündlichen Prüfung: 26. Juni 2018

Zusammenfassung

Der Forschungsbereich des Graphzeichnens beschäftigt sich mit der Anordnung von Graphelementen auf einer Zeichenfläche. Ein Ziel kann dabei sein, eine Zeichnung zu erzeugen, die für Menschen leicht lesbar ist und bestimmte Aufgaben vereinfacht. Aus theoretischer Sicht sind die genutzten Methoden in der Regel gut erforscht. Um bei Problemstellungen aus der Praxis Anwendung zu finden, fehlen ihnen jedoch meist bestimmte Funktionalitäten und eine angemessene Integration in entsprechende Werkzeuge. Hierfür gibt es verschiedene Gründe, etwa, dass häufig wohldefinierte und abgeschlossene (Teil-)Probleme betrachtet werden, die nicht alle in der Praxis relevanten Facetten abdecken. Die zu zeichnenden Graphen gehen oft aus unstrukturierten und komplexen realen Diagrammen hervor und bestehen daher aus mehr Elementen als Knoten und Kanten. Mit aktuellen Methoden erstellte Zeichnungen weisen häufig eine schlechte Kompaktheit auf.

Diese Arbeit beschäftigt sich mit einer dementsprechenden Erweiterung und Verbesserung des ebenenbasierten Zeichenverfahrens, das 1981 von Sugiyama et al. vorgestellt wurde. Im Fokus stehen dabei vor allem auch die speziellen Anforderungen von Datenflussdiagrammen. Das Verfahren lässt sich in fünf Schritte gliedern: Kreisentfernung, Ebenenzuweisung, Kreuzungsminimierung, Koordinatenzuweisung und Kantenrouten.

Die zentralen Ergebnisse dieser Ausarbeitung sind die folgenden. Im Rahmen der Ebenenzuweisung wird eine neue Methode vorgestellt, die durch die Kombinierung der ersten beiden Schritte weniger starre Ebenenzuweisungen zulässt und so die Kompaktheit der resultierenden Zeichnungen erhöht. In einer weiteren Methode werden gegebene Ebenenzuweisungen speziell auf vorgeschriebene Zeichenflächen zugeschnitten, indem der Graph schlangenartig in diese eingepasst wird. Im Rahmen der Koordinatenzuweisung werden zwei existierende Methoden so verändert und erweitert, dass festgelegte Andockpunkte an Knoten sowie orthogonal gezeichnete

Kanten besser unterstützt werden; beides sind zentrale Elemente von Datenflussdiagrammen. Des Weiteren wird ein Nachbearbeitungsschritt basierend auf eindimensionaler Kompaktierung vorgeschlagen, der die Fläche von bereits erzeugten Zeichnungen reduziert und damit deren Kompaktheit verbessert. Schließlich werden weitere Forschungsfragen identifiziert und umrissen, die es zu beantworten gilt, um die Situation in der Praxis weiter zu verbessern.

Abstract

The area of graph drawing is concerned with positioning the elements of a graph on a canvas such that the resulting drawing is well-readable by humans and aids their execution of certain tasks. While known methods are usually well-studied from a theoretical perspective, both their applicability to graphs from practice and their integration into tools from practice are not always satisfactory. This is due to various reasons, for instance, due to known methods usually solving well-defined, self-contained problems that do not cover all of the bits and pieces that must be considered in practice. There, the diagrams the graphs originate from often comprise more than just simple nodes and simple edges, they tend to be messy and complex, and existing methods regularly compute drawings with poor compactness.

This thesis is concerned with improving the well-known layer-based layout approach, originally proposed by Sugiyama et al., and devotes special attention to the requirements of dataflow diagrams. The layer-based approach is a pipeline of five steps: cycle removal, layer assignment, edge crossing minimization, coordinate assignment, and edge routing.

The main contributions of this thesis are the following: In the context of layer assignments, it presents a method that combines the first two steps allowing less rigid layerings, thereby improving on compactness, and a method that tailors a layering to a prescribed drawing area by wrapping it in a snake-like fashion. In the context of coordinate assignments, it presents extensions to and modifications of two existing methods that better address ports and orthogonally routed edges, both of which are central elements of dataflow diagrams. In the context of a generally poor compactness, it presents a post-processing step applied to a final drawing that reduces the width of left-to-right drawings by using one-dimensional compaction. Finally, based on the gained experiences, research tasks are identified and illustrated that are essential to further better the situation in practice.

Contents

1 Introduction **1**
1.1 Contributions 4
1.2 Compactness 9
1.3 Definitions 14
1.3.1 Graphs 14
1.3.2 Measures 17
1.4 Diagram Types 20
1.4.1 Dataflow Diagrams 20
1.4.2 Control Flow Diagrams 22
1.5 Layout Methods 26
1.5.1 Force-Directed Layout 27
1.5.2 Layer-Based Layout 29
1.5.3 Other Popular Methods 30
1.6 Optimization Problems 31
1.7 Boxplots 32
1.8 The Eclipse Layout Kernel 33
1.9 Test Graphs 34

2 Layer-Based Layout **39**
2.1 Cycle Removal 40
2.2 Layer Assignment 41
2.3 Crossing Minimization 46
2.4 Coordinate Assignment 49
2.5 Edge Routing 51
2.6 Port Constraints 52

3 The Layer Assignment Step **57**
3.1 Restricted Number of Nodes per Layer 58
3.1.1 Evaluating Existing Heuristics 61
3.1.2 Considering Actual Node Dimensions 70
3.1.3 Discussion 74

3.2 A Generalization of the Directed Layering Problem 76
3.2.1 IP Approach 79
3.2.2 Heuristic Approach 80
3.2.3 Evaluation 85
3.2.4 Targeting a Specific Drawing Area 90
3.2.5 Discussion 96
3.3 Wrapping 99
3.3.1 Single-Edge 103
3.3.2 Multi-Edge 112
3.3.3 Discussion 125
3.4 High-Degree Nodes 128

4 The Coordinate Assignment Step 133
4.1 Network Simplex Approach 135
4.1.1 Ports 138
4.1.2 Flexible Port Positions and Node Sizes 140
4.1.3 Straightening Edges 142
4.1.4 Remarks 147
4.2 Brandes and Köpf Approach 149
4.2.1 Ports and Node Sizes 154
4.2.2 Straightening Edges 159
4.2.3 Remarks 162
4.3 Discussion 166

5 Post-Processing: Width Reduction 171
5.1 One-Dimensional Compaction 173
5.2 Dataflow Diagrams 177
5.2.1 Edge Length 181
5.2.2 Vertical Segments 184
5.2.3 Spacings 189
5.3 State Diagrams 194
5.3.1 Splines in ELK Layered 196
5.3.2 Vertical Segments 197
5.3.3 Spacings 199
5.3.4 Sloppy Splines 200
5.4 Discussion 201

6 Further Topics of Practical Relevance **207**
6.1 Hierarchical Layout 209
6.1.1 Drawing Area Awareness 213
6.1.2 Short Hierarchical Edges 219
6.2 O-NO Drawing Avoidance 227
6.3 Knowledge-Based Drawings 238
6.4 Layout Method Configuration 247

7 Conclusions **251**
7.1 Lessons Learned 254
7.2 Future Work 255

Glossary **259**

Bibliography **261**

Chapter 1

Introduction

The area of *graph drawing* is concerned with the task of arranging the elements of a *graph*, commonly referred to as *nodes* and *edges*, within a given drawing area [DET+99; KW01; Tam13]. This can be a computer screen or a sheet of paper. The resulting drawings are supposed to aid the viewer's understanding of the graph. As such a drawing must be "tidy" and "nice" to look at, and its underlying intention may be to highlight the connectivity or the symmetry within the graph. An example can be seen in Figure 1.1, which shows a small *dataflow diagram*, one of this thesis's main objects of research.

Textual general purpose programming languages, such as C and Java, are well-known in public. Less known, but nevertheless extensively used by both computer scientists and scientists from other fields, are visual programming languages. Prominent examples of tools employing visual languages are National Instrument's LabVIEW and MathWork's Simulink, see Figure 1.5, p. 21, for two screenshots. The visual programming task involves moving and connecting elements of a *diagram* on a canvas. It has been found that graph drawing methods can significantly improve the productivity of a programmer by arranging a diagram's elements automatically by using an algorithm [KD10; FH10]. However, different visual languages and different fields of application demand different drawing styles, resulting in a plethora of interesting and often hard problems for the graph drawing community. Many of these problems can be formalized as succinct, self-contained problems with a precise objective, for instance: Arrange the nodes and edges of a graph in such a way that the total number of edge crossings is minimized. Things become messier in practice since one usually has to consider the objectives of multiple, often competing, sub-

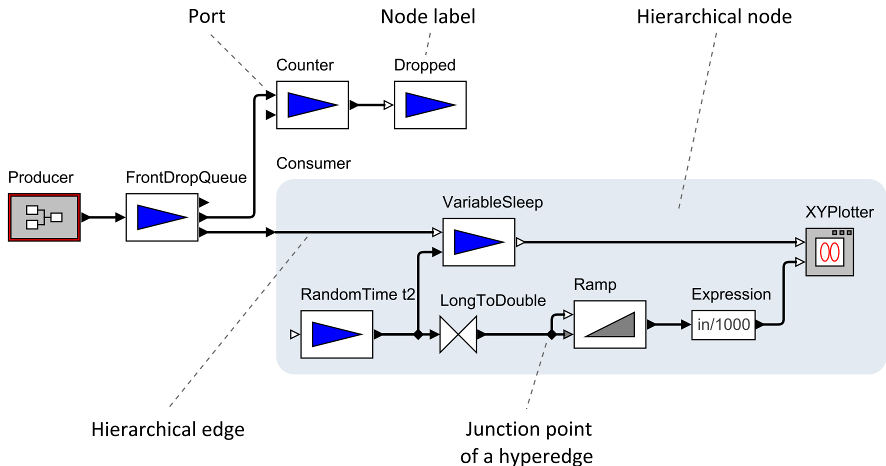


Figure 1.1. An example of a dataflow diagram taken from the Ptolemy II project.

problems simultaneously to obtain a "good" drawing. Also, the question of what makes up a good drawing has no definite answer. For one thing, it depends on application-specific drawing standards. For another thing, human individuals have different perceptions of what a good drawing is, starting with longstanding habits: "We always arranged it like that." The human intuition seems to have a reasonable feeling of how the elements of a diagram can be arranged in a way pleasing to the eye. Yet, it is often hard to capture this intuition in an algorithmic way, and it is neither clear if an intuitive arrangement aids the understanding of the diagram, nor if it helps solving a certain task more efficiently.

The task to lay out a diagram in itself is already complex and the used methods are complicated. A common way to master the complexity is to formulate simplified subproblems and solve the overall task step by step. The downside of this is that the subproblems loose touch to the big picture. An optimal solution to one of the subproblems may be suboptimal when combined with the solutions of the other subproblems. To give an example, the *layer-based layout approach* (cf. Chapter 2) computes a placement of nodes in the second of five steps, and it estimates the area that is likely to be used

by a final drawing to assess the quality of this placement. Evaluations show, however, that the obtainable estimations can deviate significantly from the area of the final drawing (cf. Section 3.2). Apart from this, state-of-the-art graph drawing methods have several issues when applied in practice, three of which that are addressed in this thesis are briefly outlined next. First, they often lack support for elements that frequently occur in diagrams from practice: *ports*, *edge labels*, *node labels*, and *comment boxes*, to name a few. Second, they yield drawings of real-world diagrams that have a poor *compactness*: the provided drawing area is not used efficiently, and a lot of whitespace remains [GHM+14]. This issue is especially imminent when *hierarchical graphs* are to be laid out, which are graphs whose nodes can contain nested graphs and whose edges can connect nodes across different hierarchies. Third, they usually optimize a set of criteria as good as possible. However, for a user looking at a drawing that shows hundreds of crossing edges it usually makes no difference if an alternative drawing is possible with one crossing less. Instead, users are more bothered with little oddities of a drawing that are "easy" to fix from their subjective perspective. As an example, consider a short edge that can obviously be drawn straight but contains a small kink.

Concluding the introduction, this thesis revolves around three central topics: the layer-based approach, dataflow diagrams, and compactness. Formulated as a sentence, it is in large parts concerned with improving the compactness of dataflow diagrams being laid out with the layer-based approach. To this end, the identified points for improvement are tackled by modifying and enhancing the line of action of the layout method itself. It should be noted, however, that there are usually multiple means to tackle a certain problem. For instance, improving the compactness of a drawing can also be achieved by *filtering*, i. e. temporarily omitting diagram elements that are irrelevant for the current task, or by *distorting*, i. e. shrinking the very same elements [SB92; LA94; Fur06]. Furthermore, blindly improving on one objective usually does not yield the desired outcome: a maximally compact drawing may lack whitespace to structure the drawing and to aid a human's *perception*. While such alternative approaches are interesting, they define their own research areas and are beyond the scope of this thesis.

1.1 Contributions

The thesis covers contributions to four areas:

Layer Assignments (Chapter 3) The second step of the layer-based approach assigns the nodes of a graph to *layers*. The resulting *layering* directly influences the width and height of a final drawing, the drawing's *shape* so to say. In a traditional layering as many edges as possible should point into the same direction, and the objective is to either use a small number of layers or to keep edges short. Certain use cases, however, require alternative layer assignment strategies, three of which are discussed here. First, it is shown how to address varying node sizes in heuristics to compute layerings with a preferably small number of nodes per layer; so far only the case of equally-sized nodes has been addressed in the literature. Second, having as many edges point into the same direction as possible restricts the number of possible drawing shapes for a particular graph. Here, the *Generalized Layering Problem (GLP)* is presented, which is allowed to reverse more edges than absolutely necessary in case it aids the creation of a certain (user-desired) drawing shape. Third, whenever a long sequence of actions is represented with a graph, the corresponding drawing can become very wide and narrow (assuming the nodes are placed horizontally). To create drawings that are better suited for common display mediums, e. g. computer screens, a *wrapping* process is presented, in which the layering of a graph is split into *chunks* that are to be drawn one below the other.

Coordinate Assignments (Chapter 4) Given a layering and an order of the nodes within each layer, the fourth step of the layer-based approach computes concrete coordinates for the nodes. The step is sensitive to ports and directly impacts the edges' routes, in particular, it controls if it is possible to draw an edge straight (parallel to one of the axes). When drawing edges orthogonally, i. e. with horizontal and vertical segments only, edge straightness is an essential factor to avoid visual clutter. Therefore, existing coordinate assignment methods are adapted and extended in this thesis, with special attention to ports and orthogonally routed edges.

Drawing Compaction (Chapter 5) Drawings of diagrams that are laid out with the layer-based approach and contain nodes with varying dimensions often contain significant amounts of whitespace. To a great extent, this is due to the rigid layering, in which a layer has to reserve enough space to fit its largest node, leaving space empty around smaller nodes. A method based on *one-dimensional compaction* is presented to improve matters. It is able to consider the peculiarities of dataflow diagrams, e. g. ports, orthogonally routed edges, and prescribed spacings between diagram elements. While particularly suited for drawings created with the layer-based approach, the method can be adapted to other drawing styles as well.

Motivation of Further Research (Chapter 6) Many points remain to be addressed when it comes to putting layout algorithms into practical use. Based on my experiences from improving and fine-tuning layout algorithms to be suitable for practical use and from discussions with practitioners and colleagues who work with diagrams on a daily basis, I summarized problematic topics and detailed possible avenues of research: the layout of *hierarchical graphs*, avoiding *obviously not optimal* layouts, incorporating *external knowledge* into the layout process, and properly *configuring* layout algorithms.

Publications

I have contributed to the following publications. The first list represents publications where the majority of the content is due to my participation.

[GHM+14] Carsten Gutwenger, Reinhard von Hanxleden, Petra Mutzel, Ulf Rüegg, and Miro Spönemann. “Examining the Compactness of Automatic Layout Algorithms for Practical Diagrams”. In: *Proceedings of the Workshop on Graph Visualization in Practice (GraphViP ’14)*. 2014, pp. 42–52.

The observation that drawings of graphs from practice often have an unfortunate aspect ratio and generally a bad compactness is confirmed by a set of empirical experiments. Possible improvements to overcome this problem are suggested for the layer-based approach and the *topology-shape-metrics approach* [Tam87].

[RKD+14a] Ulf Rüegg, Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. "Stress-Minimizing Orthogonal Layout of Data Flow Diagrams with Ports". In: *Proceedings of the 22nd International Symposium on Graph Drawing (GD '14)*. 2014, pp. 319–330.

An approach to draw dataflow diagrams using force-directed methods is presented. The required "flow" is realized by using separation constraints. It is found that compared to the layer-based approach, which in general lends itself well for dataflow diagrams, the new method shows an improved aspect ratio and compactness. Further details can be found in an accompanying technical report [RKD+14b].

[RSC+15] Ulf Rüegg, Christoph Daniel Schulze, John Julian Carstens, and Reinhard von Hanxleden. "Size- and Port-Aware Horizontal Node Coordinate Assignment". In: *Proceedings of the 23rd International Symposium on Graph Drawing and Network Visualization (GD '15)*. 2015, pp. 139–150.

Existing strategies for the coordinate assignment step of the layer-based approach are not aware of ports and different node dimensions. In this work, the successful coordinate assignment method of Brandes and Köpf [BK02] is extended to overcome these two limitations (Section 4.2).

[RES+16a] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. "A Generalization of the Directed Graph Layering Problem". In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD '16)*. 2016, pp. 196–208.

The work relaxes the requirement that traditional layerings of the layer-based approach have to have as many edges point into the same direction as possible. It presents an integer programming formulation and a heuristic to solve the relaxed problem and performs extensive comparisons of parameters and different layout strategies based on final drawings (Section 3.2). Further details can be found in accompanying technical reports [RES+15; RES+16b].

[RSG+16a] Ulf Rüegg, Christoph Daniel Schulze, Daniel Grevismühl, and Reinhard von Hanxleden. “Using One-Dimensional Compaction for Smaller Graph Drawings”. In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS '16)*. 2016, pp. 212–218.

The rigid layering of the layer-based approach often leaves behind significant amounts of whitespace. In this work, a method based on one-dimensional compaction is presented and tailored to dataflow diagrams that removes most of the superfluous whitespace (Chapter 5). Further, the applicability of the method to other scenarios permitting compaction is discussed. An accompanying technical report contains further details [RSG+16b].

[RES+17] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. “Generalized Layerings for Arbitrary and Fixed Drawing Areas”. In: *Journal of Graph Algorithms and Applications* 21.5. 2017, pp. 823–856.

Journal version of the conference paper [RES+16a] that additionally examines the suitability of the alternative layerings for fixed drawing areas (Section 3.2.4).

[RH18a] Ulf Rüegg and Reinhard von Hanxleden. “Wrapping Layered Graphs”. In: *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*. 2018.

Whenever a long sequence of actions is represented with a graph, the corresponding drawing can become very wide and narrow (assuming the nodes are placed horizontally). To create drawings that are better suited for common display mediums, e. g. computer screens, a *wrapping* process is presented, in which the layering of a graph is split into *chunks* that are to be drawn one below the other (Section 3.3). An accompanying technical report contains further details [RH18b].

Further Publications

[SSR+14a] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. "Counting Crossings for Layered Hypergraphs". In: *Proceedings of the 8th International Conference on the Theory and Application of Diagrams (DIAGRAMS '14)*. 2014, pp. 9–15.

Being able to count the number of edge crossings is essential to be able to minimize the number of edge crossings in a graph's drawing. Counting crossings between *hyperedges* is problematic in the context of the layer-based approach since the final number of edge crossings is unknown until the very end of the algorithm and can thus only be estimated during crossing minimization. This work presents estimating algorithms for counting hyperedge crossings that turn out to yield less crossings in the final drawing than exact counting algorithms. Further details can be found in an accompanying technical report [SSR+14b].

[FHK+14] Patrick Frey, Reinhard von Hanxleden, Christoph Krüger, Ulf Rüegg, Christian Schneider, and Miro Spönemann. "Efficient Exploration of Complex Data Flow Models". In: *Proceedings of Modellierung 2014*. 2014, pp. 321–336.

The work discusses strategies to efficiently explore complex dataflow models. To this end, it states requirements for both tools and layout algorithms and makes use of *transient views*.

[AGR14] Mohamed Almory, John Grundy, and Ulf Rüegg. "HorusCML: Context-aware Domain Specific Visual Languages Designer". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*. 2014, pp. 133–136.

A prototype of a tool that supports the development of domain-specific visual languages (DSVLs) is presented. It integrates abstract specifications of layout algorithms and thereby allows DSVL designers to select and customize automatic layout to their needs.

[JMM+16] Adalat Jabrayilov, Sven Mallach, Petra Mutzel, Ulf Rüegg, and Reinhard von Hanxleden. "Compact Layered Drawings of General Directed Graphs". In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD '16)*. 2016, pp. 209–221.

The work follows up on the idea of generalized layerings [RES+16a] and pursues two slightly altered objectives that are faster to solve in terms of execution time.

[RLP+16] Ulf Rüegg, Rajneesh Lakkundi, Ashwin Prasad, Anand Kodaganur, Christoph Daniel Schulze, and Reinhard von Hanxleden. "Incremental Diagram Layout for Automated Model Migration". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS '16)*. 2016, pp. 185–195.

Modifying the syntax of visual programming languages entails having to repair the drawings of existing models, for instance, because the dimensions of a graphical element changed and are afterwards overlapping in the drawing. The process is referred to as *model migration* and the work presents and compares two applicable layout techniques.

Advised Theses

- Sven Oliver Reimers. "Port-Aware Node Placement in a Layer-Based Layout Algorithm". 2015. Diploma thesis.
- Sandra Skrlac. "Enhanced Port Constraints in a Layer-Based Layout". 2015. Bachelor's thesis.
- Alan Schelten. "Hierarchy-Aware Layer Sweep". 2016. Master's thesis.
- Carsten Sprung. "Edge Bundling Techniques for Dataflow Diagrams". 2016. Master's thesis.
- Michael Cyruk. "Graph Layout of Connected Components". 2017. Master's thesis.
- Astrid Mariana Flohr. "Edge Routing with Immutable Node Positions". 2017. Master's thesis.
- Daniel Grevismühl. "Stable Diagram Compaction". 2017. Master's thesis.
- Kim Christian Mannstedt. "Alternative Layering Strategies for Sugiyama Layout". 2017. Master's thesis.

1.2 Compactness

A recurring goal of the methods presented in this thesis is to improve the *compactness* of the resulting drawings. Some of the methods target compactness straight on, others influence it in a more subtle way. But what exactly is meant by compactness? According to oxforddictionaries.com[1] the linguistic definition of the adjective *to be compact* is, amongst others:

> Closely and neatly packed together; dense.
>
> Having all the necessary components or features neatly fitted into a small space.

The word stems from the latin verb *compingere*, a composition of *com-*, meaning together, and *pangere*, meaning fasten.

This definition fits well to the underlying goals of the process of drawing diagrams – placing all elements neatly in a small area. The word neatly can be interpreted in different ways. It may solely refer to the packing of the elements as such, or alternatively, one can further include various *aesthetics criteria* [WPC+02; BRS+07]. The most compact drawing of a diagram does not necessarily have to be the best readable or most aesthetic one. Compactness should therefore be seen as one of many criteria that make up a good drawing. For instance, in addition to a compact placement, the diagram's elements should be evenly distributed within the used space. Different criteria compete with each other, are often subjective, and must be weighted in a way suitable to a certain use case. Also, the factors that make up a good drawing depend on the specific type of diagram. To give an example, a diagram type may require all edges to point from left to right. One can imagine that it may be difficult to compactly draw such a diagram within a very tall but narrow drawing area. Consequently, trade-offs are necessary between compactness and diagram-specific requirements.

Nevertheless, compactness must be measured for an objective evaluation. The next section discusses different ways to measure the compactness of a drawing.

[1] Accessed 2017/06/11.

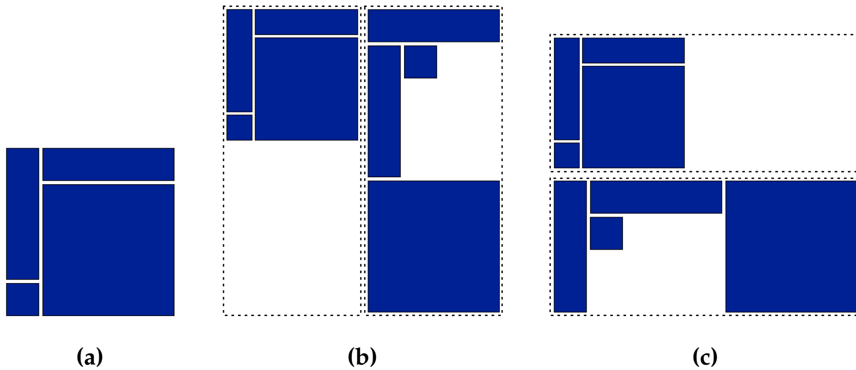


Figure 1.2. Different arrangements of four rectangles. In (a) the rectangles are placed such that the used area is minimal. In (b) and (c) a predefined drawing area, for instance a paper sheet or computer screen, is outlined by the dashed box. Note that the rectangle arrangement as given in (a) has to be scaled down in both cases in order to fit into the drawing area, whereas the two alternative arrangements can be placed with the same scaling as used in (a).

Compactness Metrics

Given a drawing of a graph, the width w of a diagram is the difference between the largest and the smallest x coordinate of any diagram element; the height h is determined analogously using y coordinates. In the area of graph drawing, compactness has traditionally been understood as the goal to either minimize the area $w \cdot h$ or to minimize w or h independently, cf. for instance the optimization goals for orthogonal compaction discussed by Klau [Kla01, Chapter 4].

One problem with this definition is that it does not allow a complete valuation of a drawing, since it is not known how much area is required for a “good” drawing of a particular graph. Another problem emerges when a prescribed drawing area is given: It depends on the drawing’s aspect ratio whether it is well-suited for printing on a sheet of paper or well-suited to be displayed on a computer screen. Figure 1.2 illustrates this. With no specific drawing area in mind, (a) represents an arrangement of the four depicted rectangles with minimum area and an aspect ratio of one. In Figure 1.2b

a drawing area is prescribed, which has an aspect ratio of about 0.4 and is outlined by a dashed box. With some imagination one can think of a computer screen rotated by 90°. To fit (a)'s rectangle arrangement into the prescribed drawing area such that everything is visible, the arrangement has to be scaled down. This can be seen on the left side of (b). Arranging the rectangles in a different manner, as depicted on the right side of (b), the overall arrangement fits into the drawing area without the need to be scaled down. (c) illustrates the same matter as in (b), this time for a drawing area with an aspect ratio of 2.25.

Which way of arranging the boxes is the best? One can argue that arrangement (a) feels more natural and that the individual rectangles can be distinguished easily despite being scaled down in order to fit them into the drawing area in (b). However, diagrams representing visual languages are comprised of more than just rectangles. They may contain various types of elements that contain text: labels and comment boxes to give two examples. It is important that these text elements can be read by a user. As such it helps if they are displayed as large as possible.

Another interesting observation is that while the bounding area of (a)'s arrangement is smaller than the one of the right side of (b), the amount of *whitespace*, the part of the drawing area that remains unused, is larger for the scaled version of (a). This may result in the feeling that the available drawing area is not used to its full potential. Compared to the previous observation regarding label sizes, this point is rather subjective, however.

In summary, the notion of compactness depends on whether a particular drawing area is prescribed or not. Additionally, when a prescribed drawing area exists, the area's aspect ratio and the question of how well a particular drawing fits this area have to be taken into account when measuring compactness. Therefore, different compactness measures should be used for the following two use cases. The measures are detailed further in the next paragraphs.

1. Minimize the area used by the drawing with no specific drawing area in mind.

2. Fit the drawing into a prescribed drawing area as good as possible.

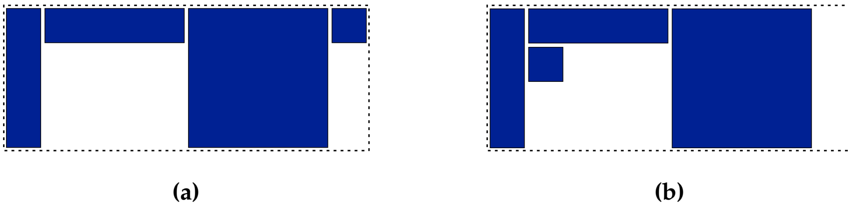


Figure 1.3. While the aspect ratio of the rectangle arrangement's bounding box in (a) perfectly matches the desired drawing area (dotted), it cannot be scaled larger than the arrangement in (b). Still, the bounding box in (b) is smaller.

Minimal Area In this scenario, the sole goal is to achieve a drawing with minimal area. Thus, the area as such is sufficient to compare the performance of different algorithms. However, since area is an absolute measure it is not possible to derive from the used area of a particular drawing of a graph how compact the drawing is in relation to all possible drawings of the graph. An alternative measure that captures this in a better way is the *whitespace ratio*, which can be defined as the ratio between a drawing's whitespace and the drawing's total area. If it is desired that the created (compact) drawing has a certain aspect ratio, the goal should be a weighted combination of area and aspect ratio. Neglecting the area, a drawing could be enlarged artificially until it matches the desired aspect ratio. The effect can be observed in Figure 1.3, where the drawing with overall larger area (a) would be the preferable one if solely optimizing for aspect ratio.
Metrics: area, whitespace ratio, aspect ratio

Prescribed Drawing Area In this scenario, a certain display medium is given, and the created drawing should use the available area as good as possible. Above, the quality of the rectangle arrangements in Figure 1.2 were discussed in terms of how a particular arrangement has to be scaled to fit a given drawing area. This scaling factor is what will be defined as *max scale* measure in Section 1.3, and it can be used to measure the compactness if a drawing area is prescribed.
Metrics: max scale

1.3 Definitions

This section formally introduces the common terminology used throughout the chapters of this thesis. More intricate definitions that are only relevant to a certain section are defined there.

1.3.1 Graphs

Definition 1.1 (Graph)**.** A graph $G = (V, E)$ is a set of *nodes* V connected by *edges* E. Edges can be *undirected*, $E \subseteq \{\{u, v\} : u, v \in V\}$, or *directed*, $E \subseteq V \times V$. Correspondingly, the graph is either called *undirected graph* or *directed graph*.

The following definitions assume a directed graph.

Definition 1.2 (Source node, target node)**.** For an edge $e = (u, v) \in E$, u is called *source node* of e and v is called *target node* of e.

Definition 1.3 (Self loop)**.** An edge $e = (u, v)$ is called *self loop* if $u = v$.

Definition 1.4 (Outgoing, incoming, and incident edges; in-degree, out-degree, and degree)**.** Let $v \in V$ be a node. The set of *outgoing edges* $E_o(v)$ is $\{(s, t) \in E : s = v\}$. The set of *incoming edges* $E_i(v)$ is $\{(s, t) \in E : t = v\}$. The set of *incident edges* or *connected edges* $E(v)$ is $E_i \cup E_o$.

The *in-degree* of v, $d^-(v)$, is $|E_i(v)|$, the *out-degree* of v, $d^+(v)$, is $|E_o(v)|$, and the *degree* of v, $d(v)$, is $|E(v)|$.

Definition 1.5 (Source, sink)**.** A node v is called *source* if $E_i(v) = \emptyset$, and it is called *sink* if $E_o(v) = \emptyset$.

Ports

Definition 1.6 (Port, port-based graph, source port, target port)**.** A *port* is a dedicated attachment point of an edge on the border of a node. A *port-based graph* consequently is a tuple (V, P, E, π), where P is a set of ports uniquely linked to nodes via the mapping $\pi : P \to V$. Edges connect pairs of ports instead of nodes: $E \subseteq P \times P$. For an edge $(p, q) \in E$, p is called *source port* of e and q is called *target port* of e.

Definition 1.7 (Port side, port constraint)**.** Ports are located on one *side* of the border of a node: west, north, south, or east. The side of a port p is denoted by $\text{side}(p)$. *Port constraints* can restrict ports to certain sides or even fixed positions and are written as $\text{pc}(p)$.

Schulze et al. name five port constraint levels [SSH14], four of which are relevant in this thesis:

FREE The ports of a node can be distributed arbitrarily on the node's border.

FIXEDSIDE For each port the side is prescribed. The order of the ports sharing a common side can be chosen freely though.

FIXEDORDER The side of a port is fixed and the order of the ports sharing a common side is prescribed.

FIXED For each port of a node explicit x and y coordinates are prescribed relative to the node.

Definition 1.8 (Hyperedge, port-based hypergraph)**.** Given a port-based graph, a *hyperedge* connects a set of source ports $S \subseteq P$ with a set of target ports $T \subseteq P$. A *port-based hypergraph* is a tuple $HG = (V, H, P, \pi)$, where H is a set of hyperedges.

To simplify matters for a layout algorithm, a port-based hypergraph can be turned into a port-based graph. Each hyperedge $h = (S, T) \in H$ is represented by a set of edges: for every pair $s \in S$ and $t \in T$ a directed edge $e = (s, t)$ is introduced. The advantage is that layout methods can be re-used for hypergraphs that are not aware of hyperedges at all.

Hierarchy

In certain applications, the nodes of a graph can contain further nodes; they form a hierarchy. The following definitions of a *hierarchical graph* and its elements are illustrated for better comprehensibility in Figure 1.4. Note that a hierarchical graph is also known as *compound graph* in the graph drawing literature [JM04, Sec. 2.6].

Definition 1.9 (Hierarchical graph, inclusion tree)**.** A *hierarchical graph* is a triple (V, E, I) that consists of a graph (V, E) and an *inclusion tree* $T = (V, I)$, where I is a set of directed *inclusion edges* that map nodes to *parent nodes*. $V_p(v)$ denotes the parent node of a node $v \in V$ in the inclusion tree, and $V_{ch}(v)$ denotes the set of v's *children*.

Definition 1.10 (Hierarchical node, simple node)**.** A node v of a hierarchical graph is called *hierarchical node* if $V_{ch}(v) \neq \emptyset$, otherwise it is a *simple node*.

Definition 1.11 (Simple edge, hierarchical edge)**.** In a hierarchical graph, a *simple edge* (u, v) connects nodes that have the same parent node: $V_p(u) = V_p(v)$. A *hierarchical edge*, on the other hand, connects nodes with different parent nodes, and it is *short* if $V_p(u) = v \vee V_p(v) = u$ and *long* otherwise.

Ports can be added to a hierarchical graph straightforwardly.

Definition 1.12 (Simple port, hierarchical port)**.** A port in a hierarchical graph is *simple* if it either belongs to a simple node or has no incident hierarchical edges. A port is *hierarchical* if it has incident hierarchical edges.

Drawing

So far all definitions were concerned with the structural properties of a graph. Next, it is defined what a drawing of a graph is and how it can be created.

Definition 1.13 (Drawing, layout)**.** A *drawing* D of a graph G is an embedding of G in $\mathbb{R}^2$. A drawing is also referred to as the *layout* of G. Note that throughout this thesis the values on the y axis grow downwards, as they usually do in the context of computer screens.

Definition 1.14 (Coordinates, dimensions)**.** The x and *y coordinates* of a node v refer to its top left corner and are written as $x(v)$ and $y(v)$. Each node additionally has a *width* $w(v)$ and a *height* $h(v)$. The same is true for ports, with the difference that the port's coordinates are relative to the top left corner of the parent node.

Definition 1.15 (Layout algorithm)**.** A *layout algorithm* computes a drawing for a given input graph. The terms layout algorithm, *layout approach*, and *layout method* are used interchangeably throughout this thesis.

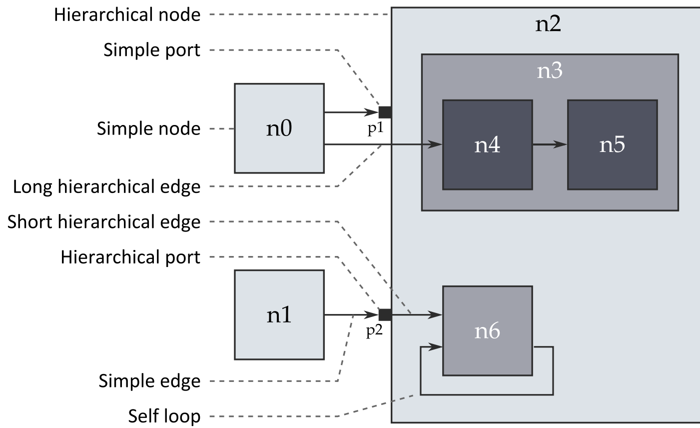
1.3.2 Measures

It is hard to evaluate art. Some people may enjoy the paintings of a particular painter, others may not like them at all. Somewhat the same applies to the readability and comprehensibility of a diagram's drawing. Different users and different engineers read diagrams in different ways in order to extract the information they desire. A certain task may be solved more efficiently if the diagram is specially tailored for this task, as opposed to simply pleasing the user's eye. Nevertheless, the visual cognition of humans adheres to certain laws and it should be possible to identify general guidelines that should be followed when creating a drawing.

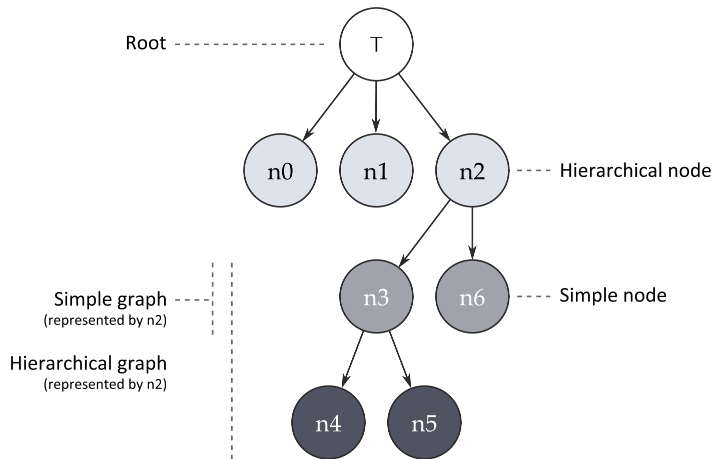
In the context of graph drawing, one tries to assess the quality of a drawing using measures commonly referred to as *aesthetics criteria* [WPC+02; BRS+07]. It is hard, however, if not impossible, to find universal criteria that apply to each type of diagram and each human being. For instance, early user studies suggest that the number of edge crossings ranks amongst the most important aesthetics criteria [Pur97]. Recent work puts this into perspective and notes that the number of edge crossings is less relevant in drawings of large and dense graphs [KPS14]. Bennet et al. present a summary of often used aesthetics criteria alongside available studies supporting them [BRS+07]. In accordance with what was said above, the authors note that domain-specific semantics and the specific task may have a significant influence on how to draw a diagram to fit its purpose, and they note that metrics may compete with each other such that they have to be individually weighted for the specific application at hand.

Large parts of this thesis focus on compactness. Therefore, proper formalizations of measures to assess the compactness of a drawing are given next. A rationale for the selected measures has already been given in Section 1.2. Throughout the thesis further measures are used which are closely related to the specific problem and are hence defined in the corresponding section.

Definition 1.16 (Width, height, area)**.** The *width* w of a drawing is the difference between the smallest and the largest x coordinate of any element of the drawing. Likewise, the *height* h is the difference between the smallest and the largest y coordinate. The used *area* of a drawing is $w \cdot h$.



(a) Drawing of the graph G, where inclusion is represented by containment



(b) Underlying inclusion tree $T = (V, I)$

Figure 1.4. Terminology in the context of a hierarchical graph $G = (V, E, I)$. (Figures adapted from ELK documentation[a])

[a] http://www.eclipse.org/elk/documentation/tooldevelopers/graphdatastructure.html [Acc. 09/01/2018]

Definition 1.17 (Whitespace, whitespace ratio)**.** The *whitespace* of a drawing is the amount of space that remains unused, thus not occupied by any element of the diagram. The *whitespace ratio* is the ratio between the number of pixels that are not used and the area in pixels.

Definition 1.18 (Aspect ratio, normalized aspect ratio)**.** The *aspect ratio* a of a drawing is defined as $a = w/h$. The *normalized aspect ratio* $\hat{a}$ of an aspect ratio a is defined as:

$$\hat{a} = \begin{cases} a - 1 & a \geqslant 1 \\ 1 - \frac{1}{a} & \text{otherwise.} \end{cases}$$

The normalized aspect ratio is useful when aspect ratios are to be compared in a plot. Without normalization the range $(0, 1)$ would include the same number of data points as $(1, \infty)$, making it hard to see how tendencies above 1 and below 1 correlate.

As discussed in Section 1.2, the aspect ratio does not allow to assess how well a drawing uses the available space of a given drawing area. To capture this, the *max scale measure* is defined next. Let $\mathcal{R} = (r_w, r_h)$ denote the width and height of a reference drawing area and $a_{\mathcal{R}} = r_w/r_h$ its aspect ratio. For instance, an A4 paper sheet (portrait) has $\mathcal{R}_{A4} = (210\text{mm}, 297\text{mm})$ and $a_{\mathcal{R}_{A4}} = 1/\sqrt{2}$. When only the relation of the two dimensions to each other is of interest, $\mathcal{R}_{A4}$ can be simplified to $(1, \sqrt{2})$. Now, for a given drawing area, the "best" drawing is the drawing which can be displayed within the given drawing area with maximum scaling factor.

Definition 1.19 (Max scale)**.** The *max scale value* s of a drawing D with width w and height h in relation to a reference frame $\mathcal{R} = (r_w, r_h)$ is:

$$s = \min \left\{ \frac{r_w}{w}, \frac{r_h}{h} \right\}.$$

Where convenient, e. g. when used as part of a minimization problem, it can alternatively be defined as the inverse of s: $q = \max \{w/r_w, h/r_h\}$.

Definition 1.20 (Max scale ratio)**.** Given two drawings, D and D', with their max scale values s and s', the *max scale ratio* $r = s/s'$ of the two drawings indicates which of the two drawings can be displayed with a larger scale

factor within the given frame. In other words, if $r > 1$, the drawing D can be displayed larger than D'. Thus, from this measure's perspective, D is preferable. To illustrate, a max scale ratio of 2 indicates that the elements of the better drawing can be displayed twice as large as the elements of the inferior one.

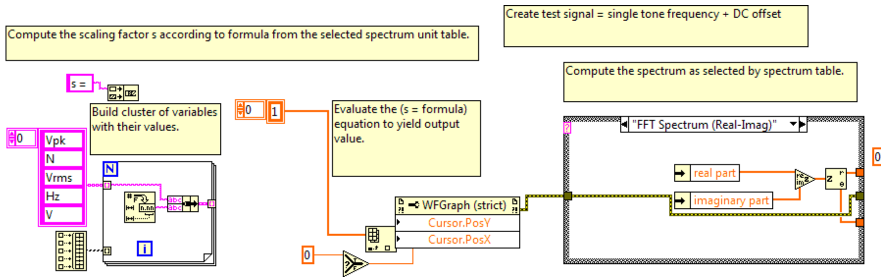
1.4 Diagram Types

Complex systems are often modeled using visual languages. In other words, programming is done by creating diagrams and by arranging the diagram's elements in a way that represents the desired functionality. Compilers translate the formal models behind the diagrams into code that can be executed by a machine, quite often this is C code. A large variety of visual languages exists and is used for various use cases; corresponding diagram types that occur throughout this thesis are introduced in the next sections.

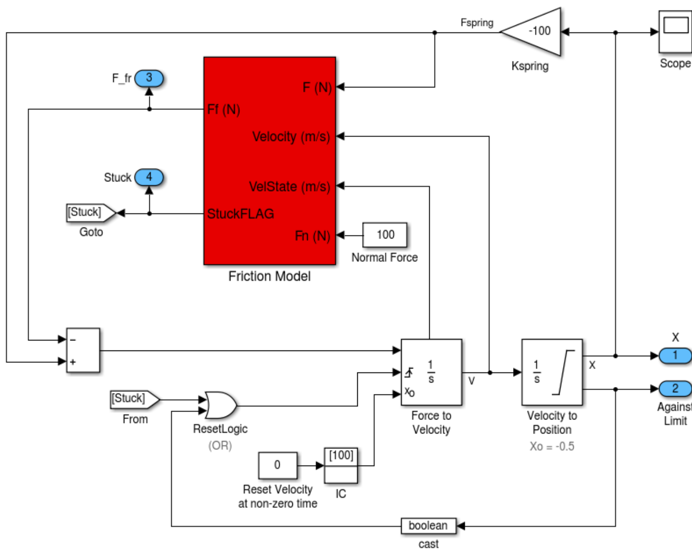
1.4.1 Dataflow Diagrams

Dataflow diagrams model the flow of data through a system. The main building blocks are *actors* which represent computational units that receive, process, and emit data. The data is transmitted between arbitrary numbers of actors via *channels*. Often channels connect to actors via dedicated connection points on their boundary, called *ports*. A dataflow diagram has already been seen in Figure 1.1 (page 2), two further examples can be seen in Figure 1.5. Actors, channels, ports, and the diagram itself can carry different types of *labels*. Labels are used to improve the diagram's comprehensibility. Actors usually have at least one label that represents their name. Compared to textual programming languages, the name of an actor corresponds to the name of a variable. Furthermore, actors can be *atomic* or *composite*. Composite actors are composed of other actors: an actor is internally represented by a self-contained dataflow diagram. The ports of the actor represent the actor's interface, connecting the internals to the outside. This allows to modularize and reuse code.

Examples of software systems that make use of dataflow diagrams are



(a) LabVIEW



(b) Simulink (source: [Spö15])

Figure 1.5. Two types of dataflow diagrams.

National Instrument's *LabVIEW*, MathWorks's *Simulink*, ETAS's *EHAND-BOOK*, and UC Berkeley's *Ptolemy II* [Pto14]. Note that terminology may differ across different systems.

Layout From the point of view of a layout algorithm, a dataflow diagram can be represented by a hierarchical port-based (hyper-)graph. The actors are nodes and the channels are (hyper-)edges. Common conventions to draw dataflow diagrams are the following:

Clearly visible flow The often directed edges are preferably drawn from left to right, with *feedback edges* being an exception.

Orthogonal edge routes (Hyper-)edges are to be drawn in an orthogonal fashion, i. e. using axis-parallel vertical and horizontal segments only.

In addition to the drawing conventions, a layout algorithm for dataflow diagrams has to provide functionality to address the following requirements:

Hierarchy Composite actors result in nodes that contain sub-diagrams. Such sub-diagrams must be laid out properly and the surrounding node's size must be adjusted accordingly.

Cross-hierarchy edges Edges can cross hierarchies, usually through *hierarchical ports*, ports on the border of a composite actor.

Port constraints The positioning of ports must conform to certain relative ordering constraints or to absolute positioning constraints, commonly referred to as *port constraints* [SSH14].

Positioning of labels Every graph element (node, edge, and port) can have an arbitrary number of labels associated with it. They have to be positioned in a reasonable way.

1.4.2 Control Flow Diagrams

The central elements of dataflow diagrams are data and how data moves between computational units. As opposed to this, *control flow diagrams* focus on which element controls a system at a certain point in time and under which circumstances control is given to another element. The most basic

version is a *state diagram*. It is made up of *states* and *transitions*. States, as their name suggests, represent the current state of the system and how the system behaves at that time. Transitions represent the possibilities to alter the behavior under certain conditions, which are directly associated with the transition. State diagrams can, for instance, be used to depict *Mealy machines*. Automatic layout of state diagrams has been discussed by Castello et al. [CMT02; CMT04].

In 1987, Harel introduced *statecharts* as an extension of simple state machines [Har87]. As new elements he added *hierarchy*, *concurrency*, and *communication*. A recent dialect of statecharts is *Sequentially Constructive Charts (SCCharts)* [HDM+14]. They were introduced by von Hanxleden et al. in 2014. The idea is to relax some of the rather restrictive semantics of statecharts with the goal to ease the programming process. SCCharts express concurrency in the form of *regions*, an example of which can be seen in Figure 1.6a, where the state WaitAandB contains two regions, HandleA and HandleB. During the compilation of SCCharts to machine code, *Sequentially Constructive Graphs (SCGs)* are constructed, see Figure 1.6b for an example. For one thing, SCGs are used for static analysis, for another thing, they can be used to debug errors in user-created SCCharts as well as to find bugs in the compiler itself.

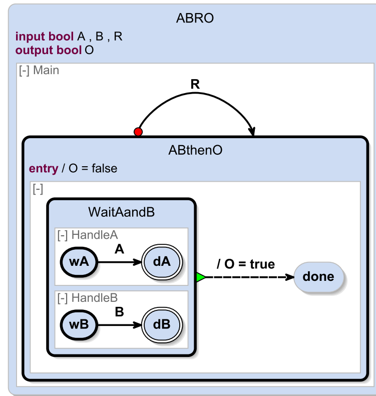
Of these four different types of control flow diagrams, only two are of further interest in this thesis: SCCharts and SCGs. As the two of them have significantly different requirements when being drawn, they are discussed separately in the next paragraphs.

Layout of SCCharts SCCharts can be represented by hierarchical graphs. States and regions are nodes, transitions are edges. By convention, SCCharts are drawn as follows:

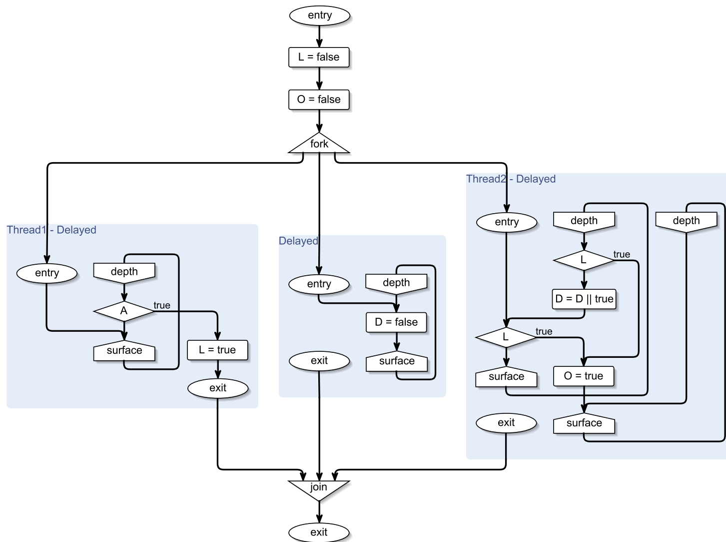
Even distribution of states The placement of states should provide a good overview of all possible states and how to transition between them.

Smooth routes for transitions It feels natural to use smooth spline routes to connect evenly distributed states.

A layout algorithm must additionally support several requirements:



(a) SCChart



(b) Corresponding SCG

Figure 1.6. Two types of control flow diagrams.

Labels A node label represents a state's name. The size of a node has to be adjusted to provide enough space for the label. The labels on edges represent conditions and actions that are executed as soon as the conditions are met. In complex models labels can become quite long.

Regions Regions are used to model concurrency and are not connected to each other by edges. Thus, arranging the regions can rather be seen as a *packing problem*.

Hierarchy States and regions form a hierarchy and occur alternating: States contain regions, regions contain states.

Layout of SCGs Just as SCCharts, SCGs are a form of control flow diagram. Nevertheless, they are drawn with significantly different conventions:

Clearly visible flow The layout of SCCharts focuses on the overall connectivity. In SCGs one is interested in the actual sequence of actions. Therefore, the edges of SCGs should point downwards. An exception are edges that are part of loops.

Orthogonal edges Edges are drawn in an orthogonal fashion as it is often the case for flow diagrams. Compared to other edge routing styles it is also more compact: when using spline routes or arbitrary polylines one usually tries to avoid sharp bends, i. e. bends with an angle close to or smaller than 90°. As a consequence an edge's path requires more space within a drawing.

Again, several elements must be supported by a layout algorithm.

Edge attachment points SCGs include nodes that represent conditionals. Edges leaving at the bottom represent the path taken when a condition evaluates to false. Edges leaving on the right side represent the true path. Consequently, the algorithm must ensure that such edges connect to the correct side.

Hierarchy SCGs contain hierarchical nodes that may stem from different origins, concurrent threads being one example.

Cross-hierarchy edges Edges can cross hierarchies but as opposed to Ptolemy diagrams no hierarchical ports are involved.

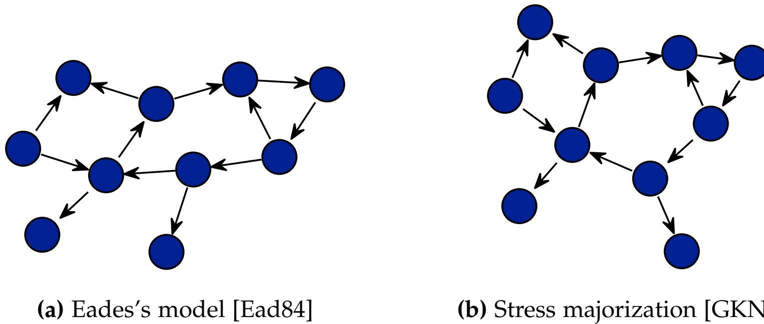
Labels Again, nodes contain labels and must be resized accordingly. Edge labels are not as relevant as for SCCharts: in their current form, SCGs only have `true` labels on one of the two outgoing edges of a node representing a conditional.

This concludes the short summary of the diagram types used during the course of this thesis. The enumerated drawing conventions are not to be seen as basic truth. They either conform to generally accepted drawing styles or reflect the way we draw these diagrams in our research group. The question whether these conventions actually aid the comprehension of a diagram is an interesting question that is certainly not easy to answer and is beyond the scope of this thesis.

1.5 Layout Methods

As discussed in the previous section, one follows different conventions and goals when laying out different diagram types. One reason for this is that a certain diagram type is used to address a certain task and that certain layouts lend themselves better to support this task than other layouts do. To give an example, identifying symmetries within a diagram is easier if the layout algorithm places the elements in a symmetric way. Similarly, a user can immediately identify a cycle within a graph if the layout algorithm arranges the cycle's nodes on the perimeter of an imaginary circle.

Many different layout approaches have been proposed over the years and range from physics-emulating approaches to purely combinatorial ones. Figures 1.7 and 1.8 show different drawings of the same graph. No approach known to date supports all imaginable drawing requirements and conventions. Thus, an approach is either suited for a certain diagram type or not. Furthermore, a drawing created by a layout algorithm should be well-readable by a user. This legibility aspect is covered and measured by *aesthetics criteria* [Pur97; WPC+02; BRS+07]. Prominent examples of aesthetics criteria are an even distribution of nodes and a low number of edge crossings. Often the criteria are conflicting both with each other and with the conventions and requirements of a certain diagram type. Therefore, they cannot be satisfied equally.



(a) Eades's model [Ead84] **(b)** Stress majorization [GKN05]

Figure 1.7. A small graph drawn with two different force-directed layout methods.

The following sections outline a set of well-known layout methods. The interested reader is referred to several summarizing books for descriptions of further methods [DET+99; KW01; Tam13].

1.5.1 Force-Directed Layout

The most used and most intuitive layout approach is the *force-directed* method, which has its roots in work of Tutte in 1963 [Tut63] and Eades in 1984 [Ead84]. An example can be seen in Figure 1.7a. Eades uses a metaphor from physics: nodes are modeled as rings that are connected by springs, the edges. The springs exert attracting and repulsing forces on the nodes. That is, adjacent nodes v and w attract each other with the force

$$C_1 \cdot \log\left(\frac{b(v,w)}{C_2}\right),$$

and non-adjacent nodes v and w repel each other with the force

$$\frac{C_3}{b(v,w)^2},$$

where $b(v,w)$ is the Euclidean distance between the nodes v and w, and C_1, C_2, and C_3 are experimentally determined constants. In an iterative procedure, these forces are calculated repeatedly and the nodes' positions

are adjusted until an equilibrium is reached. Another well-known approach of this kind was proposed by Fruchterman and Reingold [FR91]. The approaches are also known as *spring embedders*.

Kamada and Kawai introduced a related approach in 1989 [KK89]. They seek a drawing in which the *spring energy* is minimal. Energy is defined as the difference between the Euclidean distance and the graph-theoretic distance of each pair of nodes. The graph-theoretic distance of a pair of nodes equals the shortest path between them in the graph. In the following equations, let $G = (V, E)$ be a graph with node set $V = \{v_1, \ldots, v_n\}$. For a pair of these nodes, v_i and v_j, let $b(v_i, v_j)$ denote their Euclidean distance within a drawing and let d_{ij} denote the graph-theoretic distance between them. The energy of a drawing is then defined as:

$$\sum_{i<j\leqslant n} \frac{1}{2} k_{ij} \big(b(v_i, v_j) - l_{ij}\big)^2,$$

where l_{ij} is the desired distance between v_i and v_j, defined as $l_{ij} = L \cdot d_{ij}$ with L being the desired edge length, and k_{ij} is defined as K/d_{ij}^2, K being some constant.

In 2005, Gansner et al. noted that the energy term used by Kamada and Kawai is equal to what is known as *stress* in the area of *multi-dimensional scaling (MDS)* [GKN05]. There, *majorization* is used to minimize the stress, which, as reported by Gansner et al., shows better convergence and stability compared to the original solution process of Kamada and Kawai. An example drawing can be seen in Figure 1.7b. Throughout this thesis, I will refer to this method as *stress minimization*. The stress of a drawing is defined as:

$$\sum_{i<j\leqslant n} \omega_{ij} \big(b(v_i, v_j) - d_{ij}\big)^2,$$

where ω_{ij} is a normalization constant, which is often set to $1/d_{ij}^2$.

In a series of papers, Dwyer, Koren, and Marriott combined stress minimization with linear separation constraints [DK05; DKM06b; DKM06a; DKM09]. The result is a very flexible method that supports a large variety of layout requirements. To better cope with highly constrained problems, Dwyer, Marriott, and Wybrow later slightly modified the stress equation,

introducing *P-stress* [DMW09b]:

$$\sum_{i<j\leqslant n} \omega_{ij} \left(\left(\ell d_{ij} - b(v_i, v_j) \right)^+ \right)^2 + \sum_{(v_i,v_j)\in E} \ell^{-2} \left(\left(b(v_i, v_j) - \ell \right)^+ \right)^2,$$

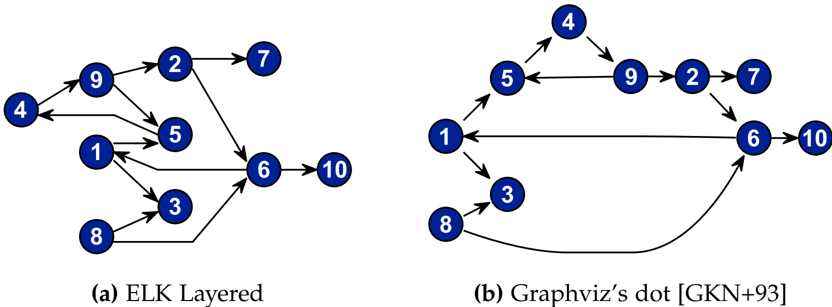
where ℓ is an ideal edge length, $\omega_{ij} = (\ell d_{ij})^{-2}$, and $(z)^+ = \max(z, 0)$. Intuitively, the left term represents repulsive forces between all pairs of nodes and the right term represents attractive forces between pairs of nodes connected by an edge.

As computing all stress contributions is time-consuming, several authors considered variations of the problem that allow faster execution [GHN12; KHK+12; GHK13; OKB17; WWS+17].

1.5.2 Layer-Based Layout

The *layer-based layout approach* was introduced by Sugiyama, Tagawa, and Toda in 1981 [STT81]. It is also simply known as the *Sugiyama approach* and is the most used approach to draw directed graphs as it emphasizes directionality. See Figure 1.8 for two example drawings and note how most of the edges point rightwards. The basic idea is to split the layout task into three consecutive steps: the *layer assignment* step distributes the nodes into consecutive layers such that edges only point from lower to higher layers; the *crossing minimization* step orders the nodes in each layer such that the total number of edge crossings is minimized; finally, the *coordinate assignment* step determines the actual node coordinates. In practice, an initial *cycle removal* step as well as a final *edge routing* step are often added to support cyclic graphs and different edge routing styles. The order of the steps directly defines the order of importance of aesthetics: a coherent edge direction is more important than a low number of edge crossings, for instance.

As this thesis requires a detailed understanding of the layer-based approach, Chapter 2 thoroughly introduces each of the five steps.



(a) ELK Layered

(b) Graphviz's dot [GKN+93]

Figure 1.8. Drawings of the same graph using two different implementations of the Sugiyama layout approach. The graph is cyclic. Both approaches selected two edges to break cycles, resulting in two leftward pointing edges. As they selected different edges, the drawings look quite different.

1.5.3 Other Popular Methods

Certain classes of graphs, such as trees and circles, may ask for drawing aesthetics that are either not provided by general layout algorithms or that can be achieved by much simpler algorithms. Trees, for instance, can be drawn without edge crossings and regarding drawing aesthetics it is usually desired that nodes sharing a level in the tree are placed on parallel lines, plus that children are placed balanced below their common parent [RT81]. A recent summary of tree layout algorithms is given by Rusu [Rus13]. Figure 1.9 shows a drawing of a tree produced with ELK's MrTree algorithm, which is based on ideas presented by Walker [Wal90].

Another fundamentally different method is the *topology-shape-metrics* approach originally introduced by Tamassia et al. [Tam87; TDB88]. It foremost focuses on producing drawings with a low number of edge crossings. Additionally, nodes are placed on a grid and edges are routed in an orthogonal fashion. The approach is well-suited to draw UML class diagrams. Adaptations to support the special requirements of class diagrams, such as upward pointing edges of generalizations, have been presented by several authors [GJK+03; EGK+04].

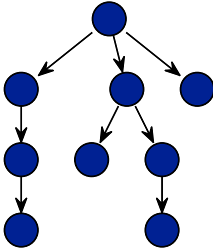


Figure 1.9. A drawing of a tree using a dedicated tree layout algorithm.

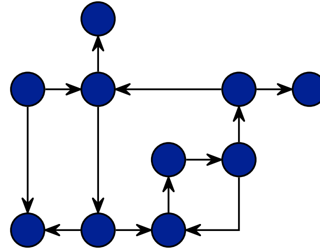


Figure 1.10. Orthogonal-style drawing created using the topology-shape-metrics approach.

1.6 Optimization Problems

In an *optimization problem* one seeks for the best solution among all valid solutions for a specific problem. To that end, an *objective function*, or simply *objective*, defines what the best solution is, and a set of *constraint functions*, of simply *constraints*, defines what the valid solutions are. To give a simple example, the objective could be to make as much profit as possible by planting and selling potatoes, and the constraining factors are the soil and the machinery that are available.

Depending on the characteristics of the objective and the constraints, different classes of optimization problems are distinguished. For instance, if both the objective and the constraints are linear, and the variables that describe the problem are reals, one speaks of a *linear optimization problem* or *linear program (LP)*. If the set of variables is additionally required to be integral, it is referred to as *integer linear program (ILP)*. In general, optimization problems are difficult to solve; solving ILPs is NP-hard, even if the variables are further restricted to zero and one [Kar72]. However, there are exceptions: LPs can be solved efficiently, for instance, with *interior-point* and *ellipsoid methods*; nevertheless, in practice often the *simplex method* turns out advantageous despite its worst-case exponential runtime [BV04]. Well-known commercial tool suites to formulate and solve optimization problems are IBM's *CPLEX* and *Gurobi*, both of which provide academic licenses and have been used to solve optimization problems discussed in this thesis.

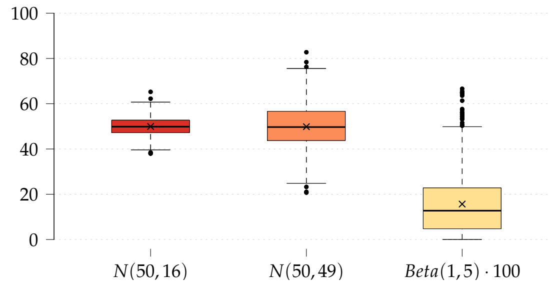


Figure 1.11. Exemplary boxplot of random values taken from two normal distributions and one beta distribution.

Coming up with a good formulation for a certain problem is often a difficult task in itself and slight modifications of or extensions to the problem may result in significant differences in solving time. Consequently, much research devotes itself to improving the formulations of crucial problems. In contrast to this, the purpose of the optimization problems occurring throughout this thesis is to serve as reference point to either assess the quality of heuristics or to discuss what is theoretically possible. Accordingly, efficiency is of secondary importance and this brief overview of the area of optimization problems should suffice.

1.7 Boxplots

Throughout this thesis, several evaluations are conducted that try to assess the effectiveness of a proposed method based on certain sets of graphs that originate from practice and do not represent a normally distributed sample. Thus, stating averaged values of the results is often not enough since it hides the results' distribution, in particular the one of outliers. A representation that works better in this case is a *boxplot* [VH81], see Figure 1.11 for an example. In boxplots a set of data points is visualized by dividing it into four groups that contain the same number of data points; the three split points are called *quartiles*. The range of the data points between the first and the third quartile is represented by a box where a bar within it represents

the second quartile, the median. *Whiskers* (dashed lines) potentially extend to both sides of this box representing the two outer groups. The length of a whisker is at most $1.5 \cdot IQR$ (there are also other variants), where IQR is the *inter-quartile range*, which is the distance between the first and the third quartile. Any data point that lies outside of the whiskers is considered to be an outlier and is indicated by a small black circle. The boxplots presented here additionally show the mean in the form of a small cross.

There are several variants and extensions of boxplots [MTL78]. An example of the ones used here can be seen in Figure 1.11 and are created with *R v3.2.3* [R C15]. The left two boxes hold 1000 random values from a normal distribution with a mean of 50 and different variances as indicated in the caption, and the right box holds 1000 random values from a beta distribution to illustrate the different indicators for median and mean.

1.8 The Eclipse Layout Kernel

The *Eclipse Layout Kernel (ELK)*[2] is a Java-based open-source project. It provides an infrastructure that allows to connect diagram editors using various technologies to layout algorithms written in different languages. Internally, it uses a central graph structure, the *ELK Graph*, which allows to model many types of node-link diagrams. Numerous *layout options* can be attached to each element of the graph. The options can be used to carefully tune a layout algorithm, e. g. to address diagram-specific drawing requirements. Figure 1.12 shows a block diagram of ELK, and further details on the underlying concepts can be found in the doctoral thesis of Spönemann [Spö15]. In addition to the infrastructure, ELK comes with a set of layout algorithms itself: *ELK Layered*, *ELK Force*, and *ELK MrTree*. While ELK is part of the Eclipse ecosystem and well-integrated with the platform itself, the layout algorithms are available as plain Java libraries as well.

ELK Layered is an implementation of the Sugiyama approach (cf. Section 1.5.2), is highly configurable to address a large variety of aesthetics, and contains several extensions specifically tailored to address dataflow

[2] `http://www.eclipse.org/elk`

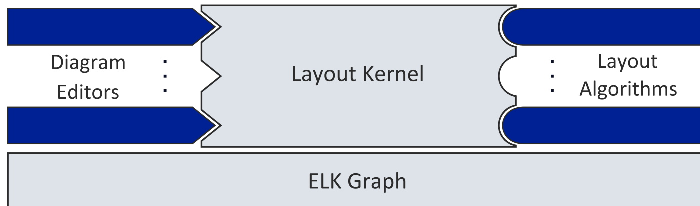


Figure 1.12. Structure of the Eclipse Layout Kernel (ELK). (Figure courtesy of C. D. Schulze)

diagrams with ports [SSH14]. Most of the methods presented in this thesis found their way into ELK Layered.

1.9 Test Graphs

Layout algorithms must be tested, not only with regard to bugs but also with regard to the quality of the resulting drawings. Two sets of graphs that are frequently used for this purpose in the graph drawing community are the *North graphs* and the *Rome graphs*.[3] The sets originate from practical applications but consist of nodes and edges only, with the nodes being dimension-less [DGL+97a; DGL+97b]. Diagrams from practice usually have nodes of varying dimensions and are richer in the number of features they include: ports and labels, for example. Over the period of this thesis, I came in contact with several diagram types (cf. Section 1.4) and used them to evaluate the methods presented here. Due to tooling changes, diagram notation changes, and improvements of the algorithms' implementations, it is possible that re-running the evaluations today may produce slightly different results. Still, to make it easier to access the various diagram types for future work and others, I converted them into ELK's textual graph format and removed diagram features that are irrelevant from the point of view of a layout algorithm. The resulting files are available in the KIELER models repository[4]. The remainder of this section introduces the sets and

[3] `http://www.graphdrawing.org/data.html`

[4] `https://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/models-public`

names their particular characteristics, an overview of which is given in Table 1.1. For further details on common layout conventions, see Section 1.4.

North and Rome Graphs The *North graphs* were collected by North during his work at the AT&T Bell Labs [DGL+97a], and are referred to as *AT&T graphs*. The *Rome graphs* were collected by DiBattista et al. The graphs are based on a set of 112 graphs from real world applications that were altered in various ways to obtain a larger number of graphs [DGL+97b].

Both sets of graphs consist of dimension-less nodes and edges, nothing else. The North graphs do not contain directed cycles and are connected, something that is not necessarily true for the Rome graphs. However, Healy and Nikolov extracted a subset of the Rome graphs that fulfills these two criteria as well [HN02b]. The filtered set is denoted as RomeF in this thesis. Furthermore, Rüegg et al. extracted a subset from the North graphs in which all graphs have at least 20 nodes and yield a large aspect ratio when drawn left-to-right with the classic layer-based approach [RES+16a]; the set is referred to as ATTar here.

Ptolemy The assembled set of Ptolemy diagrams originates from the models shipping with the Ptolemy II suite[5] itself. Remember that Ptolemy diagrams are dataflow diagrams and have been introduced in detail in Section 1.4.1. The prevalent layout direction is left-to-right. In addition to nodes and edges, the underlying graphs comprise ports with certain port constraints; usually their order is fixed. The edges are directed and connect to nodes via ports, and an edge can connect more than two ports, in which case the edge is a hyperedge. The nodes have different dimensions and labels associated with them. The graphs can be hierarchical and contain short hierarchical edges. Each hierarchical node can contain multiple disconnected subgraphs. The graphs may contain directed cycles and self loops.

In his dissertation, Spönemann discussed a way to *flatten* the Ptolemy diagrams by moving the content of all hierarchical nodes to the uppermost

[5] https://ptolemy.eecs.berkeley.edu/ptolemyII/

hierarchy level and reconnecting hierarchical edges properly [Spö15, Section 2.2.4]. The procedure mainly serves as a way to derive larger graphs for testing and evaluation.

Characteristics: ports and port constraints, node dimensions, node labels, hyperedges, short hierarchical edges, directed cycles, self loops, hierarchy, disconnected subgraphs.

SCCharts SCCharts are control flow diagrams and represent state machines with hierarchy. For a detailed introduction see Section 1.4.2. The assembled set originates from student submissions during practical courses at university. States are nodes, transitions are edges. Hierarchy is realized by (multiple) regions that are contained in a state and can in turn contain further state machines, i. e. subgraphs. There are no hierarchical edges. From a layout algorithm's perspective regions are unconnected boxes, and arranging them can be seen as a packing problem. An interesting feature of SCCharts therefore is the alternation of a "regular" graph layout problem and a packing problem. Edges have labels that represent guards and actions and can be rather space-consuming. The graphs may contain directed cycles and self loops.

SCCharts provide a set of *extended* features for ease of modeling. In an initial step of the compilation chain to machine code an SCChart is modified to contain only *core* features. From the perspective of the underlying graph, the transformation to a core SCChart usually increases both the number of nodes and edges.

Characteristics: node dimensions, edge labels, alternating hierarchy (regions, state machines), directed cycles, self loops.

SCGs SCGs are control flow diagrams emerging during the compilation of SCCharts (cf. Section 1.4.2). The prevalent layout direction is top-down. While usually not explicitly visible, edges attach to nodes via ports. Most nodes have one input port on the north side and one output port on the south side to both of which multiple edges can connect. If multiple edges connect, the edges form a hyperedge. An exception are nodes that represent conditionals. They have an additional edge connecting to the east side or

west side of the node. This edge carries a label to indicate which case, true or false, of the conditional it represents.

During compilation, SCGs take different forms. At one point several nodes are combined to *basic blocks*, which can be modeled for layout using hierarchy. Another form is a *sequentialized* SCG, which is guaranteed to be acyclic.

As two evaluations in this thesis use two sets of SCGs that originate from unit tests of the SCCharts compiler, they are listed in the overview table as well. Note that while the two sets share most of the originating models, they were assembled independently at different points in time, thus the different number of graph instances.

Characteristics: node dimensions, ports with port constraints, hierarchy, directed cycles, edge labels.

Table 1.1. Summary of the features and structural properties of the assembled graph sets. Figures in brackets denote standard deviations. Node counts include simple nodes and hierarchical nodes. Edge counts include simple edges and short hierarchical edges (there are no long hierarchical edges in the graphs). The node dimensions only include the contribution of simple nodes, thus when it comes to laying out hierarchical graphs, larger variances and larger maximum values are to be expected.

	Instances	Acyclic	Connected	Ports	Hierarchy
Rome	11534				
Rome Nikolov	5911	✓	✓		
North	1276	✓	✓		
North ATTar	146	✓	✓		
Ptolemy	256			✓	✓
Ptolemy Flat	294			✓	
SCCharts	331				✓
Core SCCharts	331				✓
SCGs	331			✓	✓
SCGs Basic Blocks	330			✓	✓
Seq. SCGs	170	✓	✓	✓	
SCGs (Unittests)	127			✓	✓
Seq. SCGs (Unittests)	135	✓	✓	✓	

Graphs	Node Counts			Edge Counts			Node Widths			Node Heights		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Rome	10	110	52.3 [25.7]	9	158	69.1 [36.1]	00.0	00.0	00.0 [00.0]	00.0	00.0	00.0 [00.0]
Rome Nikolov	10	100	48.3 [25.6]	9	158	62.7 [35.7]	00.0	00.0	00.0 [00.0]	00.0	00.0	00.0 [00.0]
North	10	100	32.1 [21.3]	9	241	45.1 [35.7]	00.0	00.0	00.0 [00.0]	00.0	00.0	00.0 [00.0]
North ATTar	20	99	39.7 [17.9]	20	168	60.3 [33.9]	00.0	00.0	00.0 [00.0]	00.0	00.0	00.0 [00.0]
Ptolemy	11	488	39.7 [60.7]	8	799	58.0 [101.9]	10.0	611.0	58.4 [17.1]	10.0	225.0	38.2 [03.5]
Ptolemy Flat	11	451	33.6 [50.8]	7	661	42.2 [73.4]	10.0	611.0	58.2 [16.8]	12.0	225.0	37.7 [04.3]
SCCharts	4	72	18.9 [14.0]	1	110	21.3 [21.3]	02.0	760.0	87.1 [48.1]	02.0	245.0	37.4 [07.4]
Core SCCharts	7	617	84.1 [83.9]	5	723	95.7 [101.0]	34.0	534.0	85.3 [12.9]	34.0	190.0	34.6 [00.5]
SCGs	7	577	95.2 [87.8]	7	833	123.6 [116.9]	75.0	1848.5	109.7 [32.4]	25.0	145.0	25.2 [01.1]
SCGs Basic Blocks	11	1006	156.3 [143.6]	15	2167	295.3 [284.9]	08.0	1848.5	109.6 [32.5]	08.0	145.0	25.1 [01.1]
Seq. SCGs	14	941	135.6 [128.5]	15	1057	154.0 [146.1]	75.0	2021.0	142.1 [24.5]	25.0	40.0	25.0 [00.0]
SCGs (Unittests)	6	563	72.8 [76.3]	6	827	98.0 [111.5]	75.0	586.0	77.2 [04.1]	25.0	61.0	25.0 [00.1]
Seq. SCGs (Unittests)	10	227	41.1 [31.8]	11	253	45.5 [35.8]	75.0	1223.0	104.8 [22.3]	25.0	25.0	25.0 [00.0]

Chapter 2

Layer-Based Layout

The following chapters present various improvements and ameliorations to the layer-based approach, for which a more thorough overview is required than the brief overview given in Section 1.5.2. In the area of graph drawing the method is also known as *hierarchical layout*, *Sugiyama style layout*, and *layered layout*. I avoid the term hierarchical layout throughout this thesis since "hierarchical" shall refer to graphs in which nodes can contain further children (cf. Section 1.3). The reader is also referred to one of several summarizing books [DET+99; KW01; HN13].

The underlying idea of the approach is to seek a layout for a directed graph in which most edges point into the same direction, highlighting the graph's inherent directionality. As a reminder, the layer-based approach was originally defined by Sugiyama et al. for acyclic graphs as a pipeline of three steps [STT81]. Two additional steps are necessary to allow practical usage, which are marked with asterisks:

1. *Cycle removal*:* Eliminate all cycles by reversing a preferably small subset of the graph's edges. This step adds support for cyclic graphs as input.
2. *Layer assignment:* Assign all nodes to indexed *layers* such that edges point from layers of lower index to layers of higher index. Edges connecting nodes that are not in consecutive layers are split by introducing *dummy nodes*.
3. *Crossing minimization:* Find an ordering of the nodes within each layer such that the total number of edge crossings is minimized.
4. *Coordinate assignment:* Determine explicit node coordinates, e. g. with the goal to minimize the distance between edge endpoints.
5. *Edge routing*:* Compute bendpoints for edges, e. g. in an orthogonal style.

Splitting the overall layout task into several steps significantly reduces the complexity of the individual steps and allows well-structured implementations. However, it also constrains the steps to take local decisions that may turn out to be unfortunate for subsequent steps. The first three steps can be seen as *topological steps* since they compute the relative positioning of the nodes to each other. The last two steps calculate explicit x and y coordinates, thus they can be seen as *geometrical* steps. The order of the steps directly reflects the trade-off of aesthetics: directionality is more important than edge crossings which are in turn more important than balanced node positions.

The next paragraphs explain the individual steps in further detail and summarize weaknesses of the current state-of-the-art. Note that as opposed to most of the existing literature, the explanations in this thesis assume a **left-to-right** layout, i. e. layers are vertical strips and layers with lower indices are left of layers with larger indices. Additionally, most of the time a port-based graph is assumed, i. e. all edges connect to nodes via ports. A graph without ports can easily be transformed into a port-based graph by introducing a source port and a target port for every edge.

2.1 Cycle Removal

Given a possibly cyclic graph, edges are to be reversed temporarily such that an acyclic graph is handed to the remaining steps. The idea behind this is that many graph problems are easier to solve for acyclic graphs. The original direction of the reversed edges is restored after the final step, resulting in edges that point into the "wrong" direction within the final drawing. Consequently, a small number of edges to be reversed is usually preferred in order to preserve the overall flow.

The problem of finding a minimum set of such edges is also known as the Feedback Arc Set Problem (FASP) and is NP-complete [GJ79]. Several approaches have been proposed to solve it either optimally or heuristically [HN13]. The most popular heuristic used in the context of graph drawing was introduced by Eades et al. [ELS93]. It yields reasonable results and runs in linear time. It has been noted by Gansner et al., however, that reversing a minimal number of edges does not necessarily yield the

best drawings, and application-inherent information might make certain edges better candidates to be reversed [GKN+93]. Moreover, the decision which edges to reverse can result in significantly different outcomes of the subsequent layer assignment step (remember Figure 1.8, p. 30).

Interesting challenges of the cycle removal step are:

P-CR1 Often, more than one set of edges exists to make a graph acyclic. Since the cycle removal step is not aware of the remaining steps, the selected set of edges may turn out to be an unfortunate choice for other steps.

P-CR2 Several diagram types use *feedback edges*, for instance, to model continuous systems where computations partly depend on the previous system state. Inevitably, the underlying graph to lay out is cyclic. The information which edge of a cycle is the feedback edge could be used to reverse exactly this edge and help the user to identify the feedback.

This thesis does not present new methods for the cycle removal step itself but discusses new methods for the subsequent layer assignment step which incorporate cycle removal.

2.2 Layer Assignment

A large variety of approaches to compute a layering has emerged over the years. After further definitions, the approaches are discussed in further detail in the next paragraphs, structured according to the pursued optimization goal.

Definition 2.1 (Layering, proper layering, short and long edge)**.** A *layering* of a directed graph $G = (V, E)$ is a mapping $L : V \rightarrow \mathbb{N}$ for which $\forall (u, v) \in E$: $L(v) - L(u) \geqslant 1$ holds. An edge $e = (u, v)$ in this layering is called *short* if $L(v) - L(u) = 1$, e is called *long* if $L(v) - L(u) > 1$. A layering is *proper* if all edges are short.

A proper layering can be constructed by splitting all edges that span multiple layers, introducing a *dummy node* in each spanned layer.

Definition 2.2 (Width and height of a layering)**.** Given a layering L, the *width* of L is the number of used layers, the *height* of L is the maximum number of nodes in any layer. For the latter the contribution of dummy nodes can either be considered or neglected.

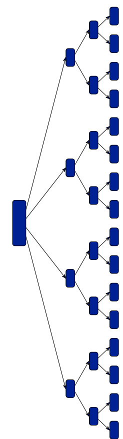
The two best-known layering strategies are the following. Eades and Sugiyama employ a method that is known as *longest path layering*. It requires linear time and the resulting number of layers equals the number of nodes of the graph's longest path [ES90]. The width of these layerings is thus minimal. Gansner et al. solve the layer assignment step by minimizing the number of dummy nodes that have to be introduced, which is tantamount to minimizing the total edge length [GKN+93]. They show that the problem is solvable in polynomial time and present a dedicated *network simplex algorithm* which in turn has not been proven to be polynomial, although it runs fast in practice. Alternatively, their approach can be formulated and solved as its dual: a minimum cost flow problem. Dummy node minimization was found to inherently produce compact drawings and performed best for the general case in comparison to other layering approaches [HN02b].

Restricted Height Coffman and Graham's scheduling algorithm with precedence constraints can be used to find a layering with a prescribed bound on the number of original nodes per layer [CG72]. The algorithm does not consider the contribution of dummy nodes to a layer's height. Healy and Nikolov tackle the problem of finding a layering subject to bounds on the number of layers and the maximum number of nodes in any layer with consideration of dummy nodes using an integer linear programming approach [HN02b]. The problem is NP-hard, even without considering dummy nodes. In a subsequent paper they present a branch-and-cut algorithm to solve the problem faster and for larger graph instances [HN02a]. Later, Nikolov et al. propose and evaluate several heuristics to find layerings with small height [NTB05]. Section 3.1 takes a closer look at these heuristics. Another approach to create narrow layerings is presented by Andreev et al., which uses ant colony optimization [AHN07].

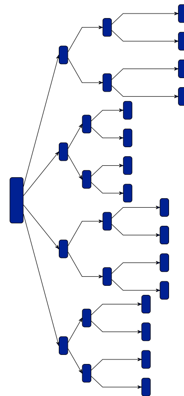
Aspect Ratio Nachmanson et al. present an iterative algorithm to produce drawings with an aspect ratio close to a previously specified value [NRL08]. The idea is to execute all steps of the layer-based approach for a given graph, measure the aspect ratio of the final drawing, and start over if the measured aspect ratio is too far from the desired aspect ratio. Internally, they use a slight variation of the Coffman-Graham algorithm to compute a layering, where the input parameter is a real number h. The sum of node heights of any layer must not exceed h. The authors use a binary search to find the best h for the desired aspect ratio. Consequently, their algorithm is not able to advance towards the desired aspect ratio if a solution to the unconstrained Coffman-Graham ($h = \infty$) is the closest amongst all possible input parameters.

Restricted Width With known methods, the width of a layering can be restricted in two ways: the integer program of Healy and Nikolov provides a parameter to restrict the number of layers, and the longest path algorithm of Eades and Sugiyama creates a layering with the minimal number of layers in either case. However, both require the input graph to be acyclic upfront, and they are bound to a minimum number of layers equal to the longest path of the graph. This implies that the bound on the number of layers in the Healy and Nikolov's integer program cannot be smaller than the graph's longest path. This restriction led to the research presented in Sections 3.2 and 3.3.

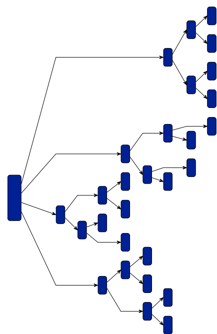
Another contributor to the width of a layering is the width of nodes. A rigid layering can result in unfortunate drawings if the width of the nodes significantly varies in size (cf. Figure 2.2b). A common attempt at a solution here is to assign wide nodes to multiple layers, for instance, by splitting them into multiple small chunks [MEL+95; NW02; FS04]. This comes at the cost that the crossing minimization step has to keep edges from crossing nodes, and that the node coordinate assignment step has to assert that all chunks receive the same y coordinate. Thus, not only do the subsequent steps have to know about the special layering, their internal complexity rises as well.



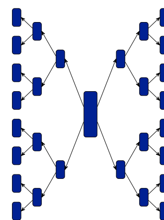
(a) Gansner et al. [GKN+93]



(b) Coffman-Graham [CG72]



(c) Nikolov et al.;
MinWidth heuristic [NTB05]



(d) Alternative drawing,
half of the edges point leftwards

Figure 2.1. Different drawings of the same tree. The example illustrates the challenges faced when the height of a tree's drawing should be kept small. (a)–(c) are created with existing layering methods, (d) demonstrates what would be possible if a subset of the graph's edges were allowed to point leftwards.

Trees A general directed graph can contain several subtrees. Handling the subtrees in a particular way is not the prime objective of a layering algorithm, however, there is enough to say about the matter to deserve its own paragraph. A significant amount of work has been devoted to forging layout algorithms for pure trees and to identifying criteria that make up a good drawing of a tree. A class of such algorithms is called *level-based* [Rus13]. The idea is to assign nodes with the same distance to the graph's root to the same level. Naturally, such drawings look similar to those of the layer-based approach. A drawback of the level-based algorithms is that the drawings become rather high when drawn left-to-right. Computing a layering for a general graph that contains subtrees with one of the layering methods discussed above computes layerings for the individual subtrees as well. How well do the traditional layering methods perform for trees? The longest path algorithm would place all leaves of each subtree in the right-most layer, possibly elongating edges unnecessarily, and thus resulting in unfortunate drawings. The approach of Gansner et al. on the other hand assigns the layers in the same way level-based tree algorithms would do, resulting in natural drawings of the subtrees.

It is not immediately clear what a good way to re-arrange the nodes of a leveled drawing of a tree is to create a drawing with less height. Dedicated tree layout algorithms solve this problem, for instance, by placing the nodes on concentric circles (*radial layout*) or by alternating the layout direction for subtrees (*horizontal-vertical layout*) [Rus13]. In the context of the layer-based approach, one is not able to move away from assigning nodes to layers since it would alter the approach's core concept.

Nevertheless, the problem has been identified before: Nachmanson et al. explicitly mention the desire to reduce the fan-out of large trees in their paper but note that additional heuristics are required to produce acceptable layouts [NRL08]. Consider Figure 2.1: Drawings (b) and (c) are created with layering methods that specifically aim at smaller heights. While being slightly narrower, the overall drawing loses quality. In drawing (d) half of the graph's edges point left. As a result, a drawing of half of (a)'s height is possible. Still, the aforementioned problems are just the same for the "leftward pointing" subgraph and the "rightward pointing" subgraph.

From a theoretical point of view, the height of a tree's layering cannot be altered when dummy nodes contribute just as much to the height of a layer as original nodes. In practice it is hardly the case that dummy nodes contribute the same height as original nodes; dummy nodes represent edges and edges are usually significantly narrower than nodes. Nevertheless, many coordinate assignment algorithms (subsequent step) aim at placing consecutive dummy nodes at the same y coordinate, which is synonymous with the desire to have straight edges. As a consequence, they often space the dummy nodes further apart than theoretically necessary. Another point to consider is that subtrees at different places in the graph likely demand individual treatment based on the immediate neighborhood.

Challenges In summary, restrictions and challenges of the layer assignment step are:

P-LA1 The width of any layering is bound by the longest path, see Figure 2.2a.

P-LA2 Layerings produced with existing methods are not always well-suited for certain display mediums such as a computer screen. Figures 2.2a and 2.2c give an idea of how this can become a problem for larger graphs.

P-LA3 In the presence of nodes with significantly different widths, the rigid layering can unnecessarily increase the overall width, see Figure 2.2b.

P-LA4 In the presence of nodes with significantly different heights, layers can become quite tall. Minimizing the number layers or the number of dummy nodes may not be desired anymore. An example can be seen in Figure 2.2d.

Chapter 3 presents several new methods and extensions of existing approaches to tackle these challenges.

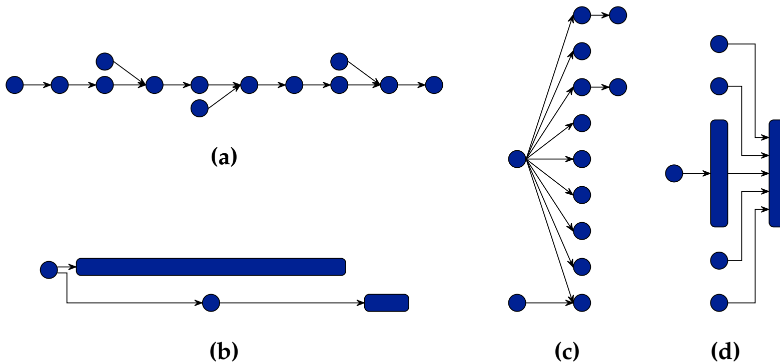


Figure 2.2. Challenges during layer assignment. (a) and (c) illustrate how certain layerings result in unfortunate aspect ratios of the drawing. With larger diagrams this can become a problem. In (b) the drawing is unnecessarily wide due to one node being pushed to the right. The height of (d) could be reduced by adding an additional layer between the two taller nodes to accommodate four small nodes.

2.3 Crossing Minimization

Minimizing the number of straight-line edge crossings of a layered graph has been shown to be NP-complete, even for only two layers with a fixed node order in one of the layers [GJ83; EW94]. As a consequence, a variety of heuristic procedures has been proposed, among which the *layer-sweep* strategy proposed by Sugiyama et al. [STT81] is among the most popular ones. It sweeps back and forth through the layers and locally minimizes the number of edge crossings between each pair of layers by fixing one of the two layers and reordering the nodes within the other layer until no progress is made or a certain termination criterion is reached. To do so, it requires two elements: (1) a strategy to select a new order for the nodes within the free layer based on the order of the nodes in the fixed layer and (2) an algorithm to count the number of crossings between each pair of layers. For (1), Sugiyama et al.'s original *barycenter heuristic* has proven to be fast and efficient in practice [JM97], nevertheless, many other heuristics have been proposed [EK86; Dre94; EW94; Cat95]. Further details can also be found

in comparative studies of Jünger and Mutzel and of Martí and Laguna [JM97; ML03]. Regarding (2), Barth et al. presented an algorithm to count the number of straight-line crossings between a pair of layers that runs in $O(n \log m)$, n being the number of edges to consider and m being the number of nodes in the layer with fewer nodes [BJM02]. For the case that a graph comprises hyperedges, Eschbach et al. note that counting crossings is problematic since the final number of crossings cannot be determined until the edge routing step has finished [EGB03]. Spönemann et al. discuss this topic further and present heuristic counting algorithms that yield fewer hyperedge crossings in final drawings [SSR+14a]. The current state-of-research in the area of crossing minimization is also summarized in the book chapter of Healy and Nikolov [HN13].

In the presence of ports and hierarchical graphs further interesting problems arise:

In-layer edge crossings To properly handle ports on the north and south side of a node, it might be necessary to introduce *in-layer edges* and to count the number of edge crossings they are involved in [SSH14]. Schelten shows in his Master's thesis how the counting of the edge crossings *between* layers can be transformed to the problem of counting crossings between in-layer edges, which allows to count the regular edge crossings and the in-layer edge crossings between a pair of layers at the same time [Sch16]. He further presents an algorithm to do so in $O(n \log n)$, n being the number of edges, which thus is competitive with Barth et al.'s algorithm at the benefit of having to implement and maintain only a single algorithm. On a side node, the problem of counting in-layer edge crossings can also be seen as counting the crossings in a *1-page book embedding*.

Adjustable port sides While westward and eastward ports correlate to the layout direction and thus are fixed to their respective side prior to the layering step, it is possible to move northern and southern ports to the opposite side, e. g. in case it decreases the number of edge crossings. A possible implementation of such a behavior has been presented by Skrlac in her Bachelor's thesis [Skr15].

Hierarchical edges In hierarchical graphs, edges can span hierarchy levels via ports on the hierarchical nodes (cf. Figure 1.4, p. 18). Laying out

such a graph in a bottom-up fashion can easily result in unnecessary edge crossings: when the order of the hierarchical ports is decided on, the implications for the outer hierarchy levels are unknown. A solution would be to perform crossing minimization on the whole graph at once, however, for reasons explained in Section 6.1 it is often desired to employ a bottom-up strategy. To improve matters, Schelten presents the *Hierarchy-Aware Layer Sweep (HALS)* method that not only sweeps the layers back and forth but, while doing so, sweeps in and out of hierarchical nodes.

2.4 Coordinate Assignment

The coordinate assignment step is the first of the two geometrical steps. The relative order of nodes to each other has been decided on: horizontally by the layering and vertically by the order of the nodes within each layer. Now the task is to calculate explicit y coordinates. Existing approaches mostly pursue the goal to place nodes in a balanced fashion, to keep the edges short, and to reduce the number of bends edges have to make. Additionally, it seems to be generally agreed on that long edges should be drawn as straight as possible [HN13, p. 441]. No qualitative proof has been given for this assumption though. The paper of Brandes and Köpf gives a good overview of the various approaches and their sometimes subtle differences [BK02]. According to Healy and Nikolov, Brandes and Köpf's approach is also the algorithm of choice for coordinate assignment. It extends ideas of Buchheim et al. [BJL01] and runs in linear time.

Central elements of this thesis are node dimensions, ports, and orthogonal edges. They add both a new element of complexity and a new degree of freedom to the coordinate assignment problem. The goals stated so far are not necessarily applicable to an orthogonal edge routing style. Consider Figure 2.3a where three nodes in two layers are connected by two edges. One may prefer the straightness of one of the two edges over a balanced node positioning in which both edges require two bendpoints. Regarding node dimensions and ports, Gansner et al. use a clever transformation of the coordinate assignment problem to the layer assignment problem and re-use their network simplex approach to assign y coordinates. They describe

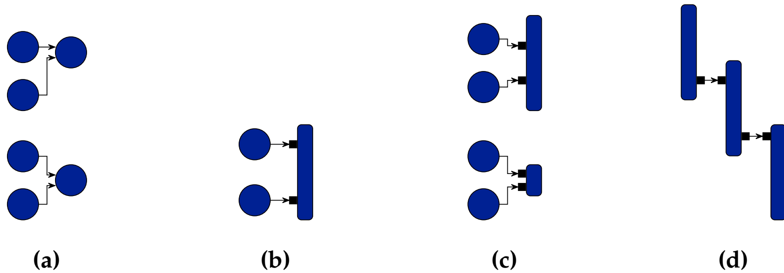


Figure 2.3. Challenges during coordinate assignment. The two drawings in (a) illustrate that balanced node positions may not be desired when routing edges orthogonally. (b) shows how more than one incident edge can be drawn straight per node when a node is tall enough. (c) shows that moving ports or increasing a node's height would allow further straight edges. (d) outlines how tall nodes and fixed port positions can result in tall drawings when aiming for straight edges.

how their approach can be used to handle different node dimensions and fixed ports [GKN+93]. Klauske uses a linear program inspired by Gansner et al.'s formulation to address flexible ports in Simulink diagrams. Nodes can be resized in height in order to reposition the ports and allow a larger number of straight edges [Kla12, Section 3.3.3]. In summary, challenges of the coordinate assignment step are:

P-CA1 Balanced node positions are not necessarily desired when edges are routed in an orthogonal fashion, see Figure 2.3a.

P-CA2 As shown in Figure 2.3b, the presence of differently sized nodes allows multiple straight edges per node side.

P-CA3 If ports were allowed to be moved on a node's border during coordinate assignment, or if nodes were allowed to be resized, even more edges may be drawn straight, see Figure 2.3c.

P-CA4 Straightening edges between tall nodes, however, may significantly increase the height of a drawing as sketched in Figure 2.3d.

Chapter 4 presents new methods that address the first three challenges.

2.5 Edge Routing

In addition to the positions of a graph's nodes, the paths of the edges are a central element of a well-readable drawing, and unless the edges are to be drawn as straight lines, the task to come up with good edge paths can become arbitrarily difficult. Four imaginable edge routing styles are:

straight-line each edge is drawn with a single straight line segment;

polyline each edge's path is a sequence of straight line segments that are joined with arbitrary angles;

orthogonal each edge's path is a polyline that is restricted to axis-parallel segments only, the style is well-known from circuit diagrams;

spline each edge's path consists of arbitrary curves, also referred to as *open jordan curves* [DET+99], with the goal to achieve smooth and visually pleasing overall edge paths.

The only thing that potentially has to be done for straight-line edge routings is to find a reasonable attachment point for the edges on a node's perimeter if the node has a dimension. The other three styles require more effort: routing the edges between adjacent layers in an orthogonal fashion restricted to one vertical segment per edge, for instance, includes the problem to order the vertical segments such that the number of edge crossings is kept small, which is NP-complete [San04]. Edge routing techniques for the orthogonal style, potentially supporting hyperedges, are discussed by Sander [San96a; San04] and Eschbach et al. [EGB03; EGB06]. Gansner et al. [GKN+93] discuss an edge routing technique that computes splines from scratch, subject to the node positions computed by the previous steps. Nachmanson et al. present a method that proceeds slightly differently, which uses the positions of long edge dummies as hints for the splines' paths [NRL08]. In practice, further interesting subproblems emerge: the routing of self loops, taking into account the edges' labels, and mixing different routing styles within the same diagram.

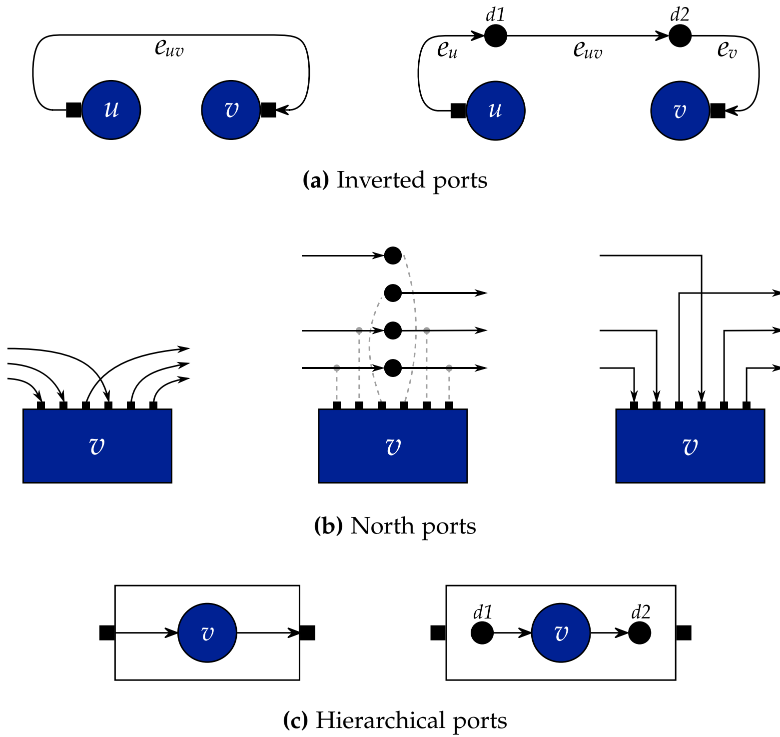
2.6 Port Constraints

The layer-based approach lends itself well for drawing dataflow diagrams, whose drawing conventions and requirements are discussed in Section 1.4.1. The layered layout highlights the overall flow of the diagram, and techniques to draw edges orthogonally exist. The requirements specific to the diagram type can be handled by a set of extensions [SFH+10; KSS+12; SSH14], of which handling port constraints is one of the more challenging tasks and entails three special scenarios that have to be addressed: *inverted ports*, *north/south ports*, and *hierarchical ports*. Small examples of the three port types can be seen in Figure 2.4. A common idea of the extensions is to transform the special requirements into constructs that can be understood by existing methods and implementations, e. g. by introducing further dummy nodes.

Remember the different port constraint levels introduced in Section 1.3 that can be specified by a user and range from completely free to rigidly fixed. During the course of layout execution, the port constraints of a node are tightened more and more (unless they are fixed from the beginning). For instance, before the crossing minimization step at least the side of a port on a node's border has to be fixed. The order of the ports on a particular side of a node can, however, still be altered to reduce the number of edge crossings. Consequently, after the crossing minimization step finishes the order of ports must not be changed anymore.

The next paragraphs explain how the three special port types are modeled during layout to the extent required for the understanding of this thesis. The paragraphs mainly summarize the results of three papers [SFH+10; KSS+12; SSH14].

Inverted Ports A port is called *inverted* either if it is fixed to the west side of a node and has outgoing edges or if it is fixed to the east side of a node and has incoming edges. A small example of a graph with two inverted ports can be seen in Figure 2.4a. The node u has an outgoing edge connected to a port on the west side, and the node v has an incoming edge connected to a port on the east side. After the layer assignment step the side of a port is guaranteed to be fixed and two dummy nodes *d1* and *d2* are introduced.



(a) Inverted ports

(b) North ports

(c) Hierarchical ports

Figure 2.4. Special requirements of dataflow diagrams that must be handled by a layout algorithm. Figures modified based on [SSH14].

The original edge e_{uv} now connects the two dummy nodes and two *in-layer edges* are added to connect the dummy nodes to their corresponding origins. In-layer edges are not supported by the traditional layer-based approach as they break the properness of the layering. Thus, the three subsequent steps must be aware of in-layer edges: Spönemann thoroughly discusses how they can be addressed during crossing minimization and edge routing and ignores them during coordinate assignment [Spö15]. Nevertheless, if properly considered during coordinate assignment, edge lengths can often be improved (cf. Chapter 4).

North and South Ports The layer-based approach draws edges from one layer to the next layer. Thus, it makes sense that edges start at a node's east side and end at a node's west side if the layout direction is left-to-right and port constraints permit.

Wherever a port is fixed to the north side or the south side of a node, care must be taken that the space required to route the incident edges is not blocked by other nodes of the same layer. For orthogonally routed edges a bendpoint must be added to the edge's path. An example of north/south ports can be seen in Figure 2.4b, where several edges connect to ports on the north side of the node v. The figure in the middle illustrates how *north/south port dummies* are introduced after layer assignment. *Ordering constraints* assert that the dummy nodes stay on the required side during crossing minimization and prevent other nodes from being placed between the dummy nodes and the original node [SSH14].

Contrary to inverted ports, no in-layer edges are added that connect the dummy nodes to the node they originate from. Thus, during coordinate assignment care has to be taken to prevent the dummies from moving too far from the node they belong to, unnecessarily elongating the edges. How this can be handled is described in Chapter 4.

Hierarchical Ports Edges can traverse the hierarchies of hierarchical graphs via hierarchical ports (remember the definitions of Section 1.3). Hierarchical graphs can be laid out in a bottom-up fashion using layout algorithms that are not aware of the hierarchy (cf. Section 6.1). Nevertheless, in the presence of hierarchical ports care has to be taken that the edges are routed properly from an inner graph to the hierarchical port. For the layer-based approach this can be achieved by introducing *hierarchical port dummies*. Figure 2.4c shows an example where the dummy nodes *d1* and *d2* represent the west side hierarchical port and the east side hierarchical port, respectively. It must be guaranteed that these dummies end up at the border of the node. For west and east ports of the hierarchical node this means that the layer assignment step has to place the dummy nodes in either a separate initial or a separate last layer. For north and south ports the crossing minimization step must place them at the beginning or end of a layer. For free or fixed side port constraint levels, the positions of the dummy nodes can directly

be transferred to the hierarchical ports. For more restrictive port constraints, a dedicated edge routing connects the ports to their corresponding dummy nodes, see Klauske et al. [KSS+12].

This concludes the overview of the layer-based approach and the discussion of the special requirements of ports and port constraints. The next chapters keep an eye on these requirements wherever necessary while they present new methods for individual steps of the approach.

Chapter 3

The Layer Assignment Step

The second step of the layer-based approach is the layer assignment step. As a reminder, the step assigns the nodes of a graph to disjoint *layers* such that (a) nodes connected by an edge end up in different layers and (b) all edges point into the same direction. The latter point is always viable due to the preceding cycle removal step. The two classical approaches are:

- The *longest path approach* by Eades and Sugiyama [ES90]. It assigns nodes to layers as "early as possible" starting with the graph's sinks. As such it creates a layering with the minimal number of required layers.
- The *network simplex approach* by Gansner et al. [GKN+93]. It minimizes the number of dummy nodes that have to be inserted to create a proper layering for the next step. In other words, it minimizes the total edge length.

The two approaches are abbreviated as LP and NS throughout this chapter.

Remember that the *width* of a layering is defined as the number of used layers and the *height* is defined as the maximum number of nodes in any layer. To avoid confusion with the width and height of a final drawing, which are usually measured in pixels and strongly depend on the remaining steps of the layer-based approach, the width and height of a layering are referred to as *estimated width* $\widetilde{w}$ and *estimated height* $\widetilde{h}$. The dimensions of the final drawing may also be referred to as *effective width* and *effective height*.

Port constraints and the resulting inverted ports and north/south ports come in effect after the layering step and can therefore be neglected for now, and hierarchical ports have been converted to dummy nodes prior to this step. Also, the ports themselves play no relevant role during layer assignment, which is why throughout this chapter G usually represents a

directed, port-less graph with nodes V and edges $E \subseteq V \times V$.

This chapter is concerned with three main topics: computing layerings with a restricted number of nodes per layer (Section 3.1), explicitly allowing further edges to point backwards within the layering (Section 3.2), and cutting a layering into multiple chunks (Section 3.3). All three topics aim at improving the compactness of final drawings and at better fitting drawings to prescribed drawing areas.

3.1 Restricted Number of Nodes per Layer

The classical layering approaches have no control over the width and height of the resulting layering. As such, it is possible that the layering of a particular graph has a small number of layers but many nodes per layer: the layering results in a rather high but narrow drawing. This gives rise to the desire for a layering approach that creates a layering with a restricted number of nodes per layer, addressing one facet of challenge P-LA2 (p. 46). As mentioned in Chapter 2, the Coffman-Graham algorithm allows to restrict the number of original nodes per layer [CG72]. However, the dummy nodes introduced for long edges have a significant impact on the height of a layer and should not be neglected. Branke et al. showed that finding a layering with minimal height is NP-complete when dummy nodes are considered [BLM+02]. Nikolov et al. presented heuristics for the problem and coined the term *minimum-width layering*, assuming a top-down layout direction [NTB05]. The prevalent layout direction in this thesis is left-to-right, therefore the term "width" is avoided where possible in this chapter. One of Nikolov et al.'s heuristics is called MinWidth. The reader is kindly asked to not be confused by the fact that here it actually refers to the desire to reduce the layering's height. The problem looked at in this section thus is the following, exemplary drawings can be seen in Figure 3.1.

Problem Layering With a Restricted Number of Nodes per Layer

Given a graph G, find a layering L such that the maximum number of nodes in any layer, i. e. the layering's height, is restricted in some sense.

The exact kind of restriction is left open on purpose and can be made concrete in various ways. It can, for instance, be a constant specified by a user that restricts the maximum number of nodes per layer. If dummy nodes are disregarded, the previously mentioned algorithm of Coffman and Graham can be used to compute the layering. When dummy nodes are regarded, however, it is not known upfront to a user how many nodes per layer are required at the minimum and how many of them are required for a reasonable layering. Similarly, a fixed bound on the height in pixels is hard to realize since the height can only be estimated at this point. The heuristics of Nikolov et al. do not use a hard restriction but try to come up with a layering whose height is smaller than the height of layerings created with traditional approaches.

This section seeks to answer the following questions:

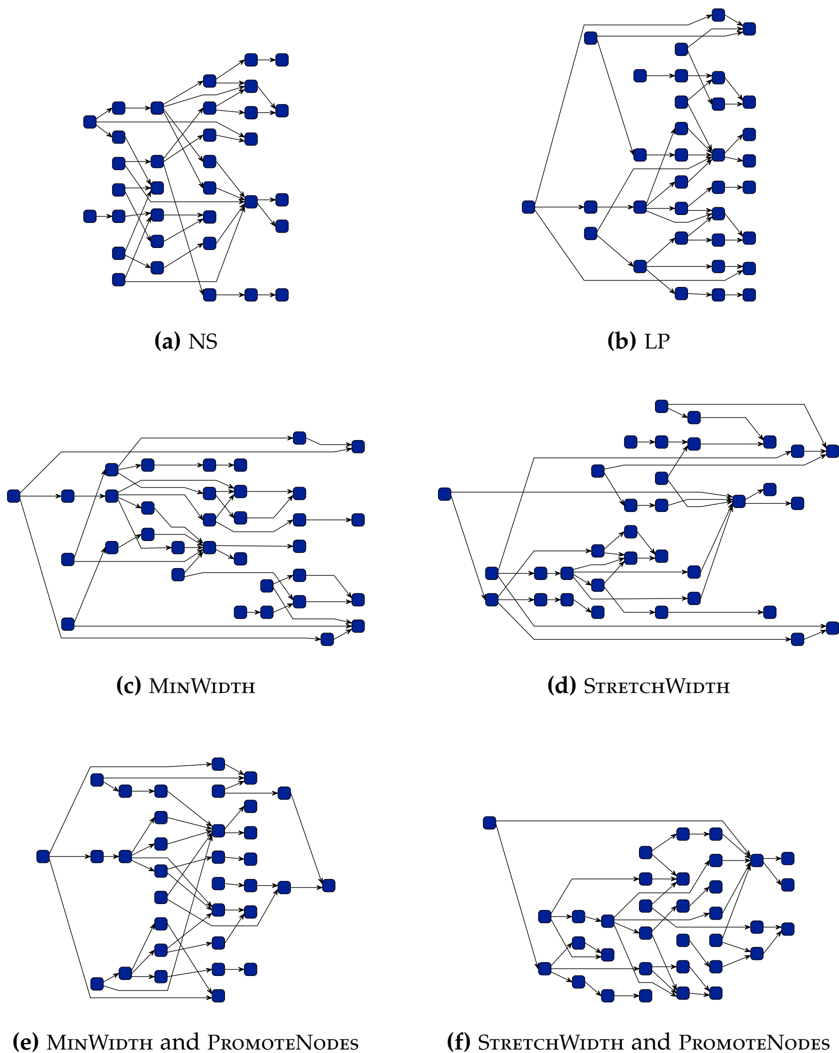
1. While Nikolov et al. call it "minimum-width layering", their heuristics do not necessarily find a layering with the minimal number of nodes in any layer. After all they are heuristics. What is the gap between heuristic solutions and optimum solutions? How do the traditional layering methods compare to optimal solutions? Both points were not yet examined by Nikolov et al.

2. Can the heuristics be used to target a certain drawing area, for instance a computer screen?

3. Is there a need for and an easy way to extend the heuristics to consider differently-sized nodes?

After a brief introduction of the graphs that are used to evaluate the presented methods, the three questions are addressed successively in the following subsections.

Test Graphs The evaluations of this chapter are based on the RomeF graphs assembled by Nikolov et al. (see Section 1.9) and on three subsets of these graphs that originate from partitioning them as follows. Laying out the graphs from left to right using layer and coordinate assignment techniques presented by Gansner et al. [GKN+93] and a simple polyline edge routing results in a final drawing with a certain aspect ratio. Based on this aspect



(a) NS

(b) LP

(c) MinWidth

(d) StretchWidth

(e) MinWidth and PromoteNodes

(f) StretchWidth and PromoteNodes

Figure 3.1. Different layerings of the `grafo11465.34` graph from the Rome graphs collection, produced with traditional methods, (a) and (b), and with the heuristics of Nikolov et al. [NTB05], (c)–(f).

ratio, three graph sets were composed, each with 1000 graphs:

LOW*:* the 1000 graphs with the lowest aspect ratios (roughly 0.3–1.0);

MIDDLE*:* the 1000 graphs with aspect ratios around 1.6, the aspect ratio of up-to-date computer screens (roughly 1.5–1.7); and

HIGH*:* the 1000 graphs with the highest aspect ratios (roughly 2.2–7.1).

The rationale for this partitioning is that the layering heuristics of Nikolov et al. ought to perform differently based on the aspect ratio of a "traditional" drawing. Consider the following example: a low aspect ratio value of 0.5 in a left-to-right drawing may be an indication that the maximum of the number of nodes in any layer is larger than the number of layers. To increase the aspect ratio, thus to better match a computer screen, the maximum number of nodes per layer would have to be decreased by moving certain nodes to new layers. Exactly this is what the heuristics try to achieve.

Additionally, a set of graphs made up of SCGs is used, which, contrary to the Rome graphs, comprises nodes with varying dimensions. Remember that SCGs are drawn top-down with orthogonal edges. The set is a subset of the SCGs that originate from unit tests (cf. Section 1.9), and it comprises of 40 diagrams with final drawings with aspect ratios between 1.3 and 4.5 and one outlier of 8.5. The graphs have between 32 and 563 nodes, 84.3 on average. An average of 9.2 nodes per graph are hierarchical nodes, i. e. nodes that contain further nodes. The edge counts are between 40 and 827, 113.8 on average. In Section 3.1.2 the set is used to evaluate the presented extensions of Nikolov et al.'s heuristics to consider individual node sizes.

3.1.1 Evaluating Existing Heuristics

Nikolov et al. were the first to introduce heuristics that create layerings with a restricted number of nodes per layer while considering dummy nodes [NTB05]. In their 2005 paper they discuss the following three heuristics: MINWIDTH (MW), STRETCHWIDTH (SW), and PROMOTENODES (PN).

The next paragraphs briefly introduce the heuristics and their main differences. For an in-depth discussion, the reader is referred to one of the

original papers [TNB04; NTB05; NT06]. Afterwards, the aforementioned questions are tackled.

MinWidth The actions of the MW heuristic are roughly based on the LP algorithm. The LP algorithm iteratively places all sinks of a graph in a layer, removes the sinks from the graph, and continues with a new layer and the newly created sinks until the graph is empty. MW adds two new elements to this procedure: (1) instead of adding all available sinks to the current layer, it only adds nodes as long as a certain threshold on the height (and the predicted height of future layers) is not exceeded, and (2) the next node to be added to the current layer is chosen to be the one with the largest out-degree. The idea behind the second point is that the node with the largest out-degree has the most connections to already placed nodes; selecting it keeps the number of dummy nodes that have to be introduced to create a proper layering low.

The last, more subtle difference, is that MW makes use of two input parameters explained in more detail later. The authors conducted extensive parameter studies and found a set of eight promising parameter combinations. They propose to run the algorithm for all eight combinations and select the layering with smallest height.

StretchWidth The second layering heuristic, SW, constructs the layering in a similar fashion to MW but does not depend on input parameters. Instead, SW starts with a low value for the threshold on the height of a layer and increases it iteratively whenever it finds that the threshold cannot be realized.

PromoteNodes Nikolov et al. found that the two previously described heuristics tend to produce layerings with a large number of unnecessary dummy nodes, the removal of which would not necessarily harm the height of the layerings. They therefore suggest to execute a post-processing algorithm: PN. PN takes a given layering as input and then recursively checks whether moving groups of nodes to earlier layers decreases the

overall number of dummy nodes. When used in conjunction with MW or SW, nodes are only promoted if the height of the layering is not increased.

Differences between MW and SW The most obvious difference is that MW uses two input parameters (in addition to the graph itself) while SW does not use any input parameters. Apart from that, both heuristics are based on the idea of the LP layering algorithm and essentially have two further decisions to make: (1) select the node which should be placed next, and (2) start a new layer when a certain height is reached.

Regarding (1), out of all nodes that could possibly be placed in the current layer, MW picks the node v with the highest out-degree $d^+(v)$. SW, on the other hand, picks the node v with the highest *rank*, which is defined as $\max\{d^+(v), \max_{(u,v)\in E} \ d^+(u)\}$. The rank allows to look at nodes that cannot be placed yet but will have a significant influence on the layering's height as soon as they are placed. At the same time it yields shorter edges. Essentially, the rank realizes a one node lookahead with relation to the nodes' degrees.

Regarding (2), both heuristics internally use two variables: the height h_c of the currently constructed layer and an estimate of the height of future layers h_u. MW compares its two input parameters with these internal variables and starts a new layer if (a) h_c exceeds the input parameter for the upper bound on a layer's height *ubh* and no node with outgoing edges can be placed, or if (b) h_u exceeds *ubh* multiplied by the second input parameter *ubc*. Note that therefore *ubh* is not a bound on the height in a strict sense since it allows further nodes to be placed in the current layer as long as they have outgoing edges. SW, on the other hand, starts with a lower bound on the allowed height h_c, e. g. the maximum out-degree of any node in the graph, and increases this bound until it can construct a feasible layering. A new layer is created if either (a) h_c exceeds the current lower bound or if (b) h_u exceeds the current lower bound multiplied by the average out-degree of all of the graph's nodes.

It should also be noted that MW first assigns a selected node to the current layer and then checks the condition if a new layer should be started. SW proceeds vice versa, first checking its condition and potentially starting a new layer, then assigning the selected node.

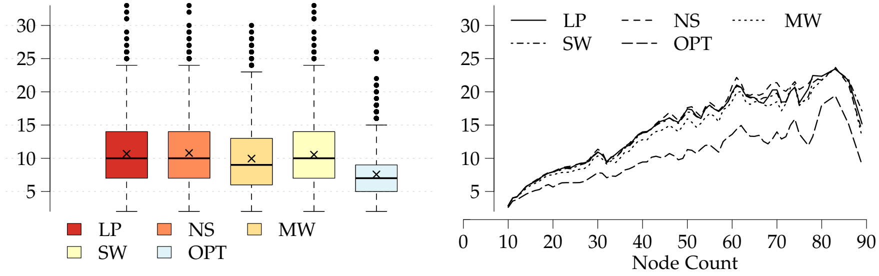
True Minimums

The first question to be inspected in this section is how narrow layerings can become when dummy nodes are considered. The heuristics of Nikolov et al. aim at finding layerings with minimal height, however, the authors never compare the heights of their layerings to optimal solutions. Here, an optimization problem, described in the next section, is used to compute a layering with minimal height (OPT). To further evaluate how far the height can be reduced in a relaxed setting, the requirement that edges have to point into a common direction is dropped. OPT2 denotes the results of this variation. Results can be seen in Figure 3.2. In (a) the results for 3519 RomeF graphs are shown for which OPT finished within a set time limit. The results of MW, SW, and LP (with subsequent node promotion) as well as NS are close to each other, with MW resulting in slightly narrower layerings. This conforms to the results of Nikolov et al. [NTB05]. Concerning the actual minimum height, there is a gap of about 22 % between MW and OPT. The results including OPT2 can be seen in Figure 3.2b, finishing for 1902 graphs. This time there is a gap of about 75 % between the results of MW and OPT2.

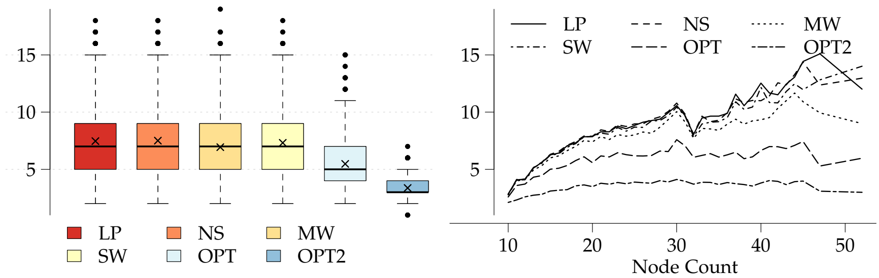
Concluding, there is a significant gap between the heuristics' results and optimal results. It must be noted, however, that the heuristics are not allowed to let edges point backwards, as OPT2 is, and that it is possible that other metrics such as the number of dummy nodes and a layering's width may be significantly worse for OPT and OPT2. Nevertheless, it gives a feeling of what is theoretically possible when seeking layerings with a small height.

Optimization Problem As a first try, Healy and Nikolov's integer program for the minimum-width layering problem [HN02a] was straightforwardly adjusted to minimize the maximum number of nodes in any layer instead of the number of dummy nodes. However, an approach that combines *Constraint Programming (CP)* with *SAT solving* [OSC07; FS09] turned out to yield a larger number of optimal results within the same time limit. The new model is defined in the *MiniZinc language*[1], a high-level language

[1] `http://www.minizinc.org/` [Acc. 20/02/2018]



(a) 3519 graphs (without OPT2)



(b) 1902 graphs (with OPT2)

Figure 3.2. The y axis shows the estimated heights (maximum number of original nodes and dummy nodes in any layer) for the different layering methods. OPT refers to the minimal possible estimated height, OPT2 refers to the minimal possible estimated height if the requirement is dropped that edges have to point into a common direction. OPT and OPT2 were implemented as optimization problems which did not finish for all graphs within a set time limit, thus only subsets of the graphs are plotted here. The boxplots and the line plots contain the same data.

for specifying optimization problems independent of a solver. It was executed using the open-source solver *chuffed*[2].

Inputs Let $G = (V, E)$ be a graph with a set of nodes V and a set of edges E. Let $\{1, \ldots, n\}$ denote the nodes, where $n = |V|$.

Parameters The layer a node $i \in V$ is assigned to is stored in a variable x_i and can take values in $\{1, \ldots, n\}$, where n is a trivial upper bound on the number of layers. For a layer $1 \leqslant l \leqslant n$, the variable l_l holds the number of regular nodes in this layer. Further, the variable h_l holds the sum of regular nodes as well as dummy nodes in layer l. *b2i* is an operator that converts a boolean expression to an integer: true to 1 and false to 0.

Objective

$$\text{Minimize} \quad \max_{1 \leqslant l \leqslant n} h_l \tag{A}$$

Constraints

$$x_i < x_j \qquad \text{for all } (i,j) \in E \tag{B}$$

$$l_l = \sum_{i \in V} b2i(x_i = l) \qquad \text{for all } 1 \leqslant l \leqslant n \tag{C}$$

$$h_l = l_l + \sum_{(i,j) \in E} b2i(x_i < l \wedge x_j > l) \qquad \text{for all } 1 \leqslant l \leqslant n \tag{D}$$

The constraints can easily be altered to allow edges with an arbitrary direction. For this, constraints (B) and (D) are replaced by the following two constraints.

$$x_i \neq x_j \qquad \text{for all } (i,j) \in E \tag{E}$$

$$h_l = l_l + \sum_{(i,j) \in E} b2i(\min\{x_i, x_j\} < l \wedge \max\{x_i, x_j\} > l) \qquad \text{for all } 1 \leqslant l \leqslant n \tag{F}$$

[2] `http://github.com/geoffchu/chuffed` [Acc. 20/02/2018]

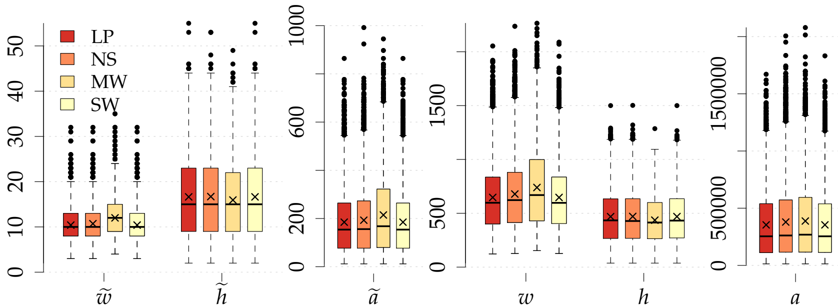


Figure 3.3. The width, height, and area (y axis) of the drawings of RomeF graphs when drawn with different layering methods. Estimated measures (left side) are marked with a tilde.

Assessing Final Drawings

Nikolov et al. used estimated values of the width and height of a layering for their evaluations. As mentioned before, it is not enough to assess the performance of a layering heuristic using estimations of the final dimensions of a drawing when it comes to practical applications. The layering step only defines parts of the topology of a drawing and it is not until the coordinate assignment and edge routing step that explicit coordinates are determined. For this reason, the evaluation of Nikolov et al. is extended here by further measures, such as the effective height in pixels.

To obtain final drawings of the RomeF graphs, a coordinate assignment technique presented by Gansner et al. [GKN+93] and a simple polyline edge routing were used. The prescribed spacing between pairs of nodes is set to 20 pixels, both vertically and horizontally. The parameters *ubh* and *ubc* of MW were set to 4 and 2, respectively.

Summarizing boxplots of the results can be seen in Figure 3.3. The four layering algorithms behave similarly for the presented measures with MW being a minor exception: the heights of the layerings created with MW are slightly smaller but the widths of layerings are larger. This is true for both estimated and effective values. Looking at the area, the reduction in height cannot compensate for the increase in width, resulting in an overall larger

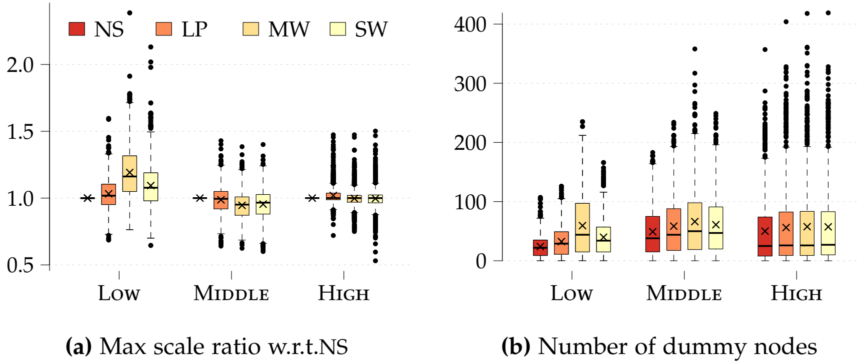
area for MW. Furthermore, while the estimated height is on average larger than the estimated width for all layering algorithms, it is the opposite way around for the effective width and effective height. A possible explanation for this would be different spacing values between pairs of layers and between pairs of nodes within the same layer. This is not the case here since both spacings are set to the same value as mentioned above. It is thus more likely that the additional width is contributed by the final edge routing step, which makes enough room between pairs of layers to tidily route the edges. Consequently, the estimated values obtained after the layering step cannot be used as a reliable indicator for the aspect ratios of the final drawings.

Concluding, it can be said that while the estimations of width and height are similar to the width and height of a final drawing for the RomeF graphs, care has to be taken when drawing conclusions based on combinations of the measures, e. g. the aspect ratio. Moreover, MW indeed slightly reduces the height of a final drawing, however, at the cost of an overall larger area. Generally speaking, no significant differences can be observed between the layering algorithms, particularly no advantages in terms of the drawing area. This leads to the question whether MW and SW are useful in practice. The next section addresses this question; after all, reducing the height while increasing the width can help to better fit the drawing of a graph into a particular drawing area.

Targeting a Specific Drawing Area

Usually the drawing of a graph ends up being displayed on a screen or printed on a page. In both cases it is not enough to look at the width and height of a drawing in isolation to evaluate the drawing's overall quality. The max scale measure introduced in Section 1.3.2 grasps this desire to assess the quality of a drawing with respect to a particular reference frame.

The findings of the previous section lead to the question whether MW and SW are usable in practice. If one would plot max scale values with respect to a regular computer screen for the RomeF graphs layered with the layering methods discussed in this section in the same way as in Figure 3.3, the conclusion would likely be that NS works better on average than the heuristics. Nevertheless, even if not usable as general purpose layering



(a) Max scale ratio w.r.t.NS

(b) Number of dummy nodes

Figure 3.4. Boxplots for the subsets of the RomeF graphs. A reference frame $\mathcal{R}_{(16,10)}$ and a left-to-right layout direction were used. The max scale ratio values are relative to the max scale values of NS.

algorithms, MW and SW may be able to create layerings that are *more suitable* for a certain drawing area where NS fails to produce good results. To evaluate this, the RomeF graphs, partitioned as explained at the beginning of this section, are used: Low, Middle, and High. MW and SW try to create layerings with a small height, thus they are expected to perform well for graphs for which NS uses few layers and places many nodes within the layers, i. e. the Low graphs.

A reference frame $\mathcal{R}_{(16,10)}$ serves as the basis for the evaluation, resembling an up-to-date computer screen. The layout direction is left-to-right. The results seen in Figure 3.4 confirm the presumptions on the one hand and indicate that one has to be careful when to apply a certain layering method on the other hand. The boxplots in (a) show max scale ratio values for MW, SW, and LP relative to NS's results. The boxplots in (b) show the number of dummy nodes for each method. For the Low graphs, MW produces better results than NS for over three quarters of the graphs. However, a larger number of dummy nodes (thus edge length) is necessary which may worsen readability. The picture is completely different for Middle, where MW produces worse results. NS guarantees a minimum number of dummy nodes, and thus produces very compact drawings. Because the graphs of

Middle were selected to roughly fit $\mathcal{R}_{(16,10)}$ when drawn with NS, it feels natural that MW performs worse. For High, no conclusion can be drawn from the results. Still, since the graphs in this set are already quite narrow, MW and SW would have to behave in the opposite way (putting more nodes in layers) in order to improve the max scale measure for $\mathcal{R}_{(16,10)}$.

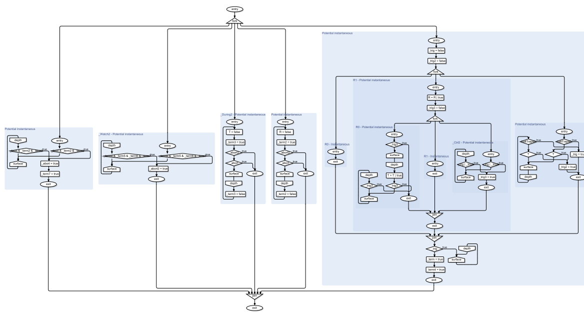
To conclude, it makes sense to use MW and SW in certain cases where NS is not able to produce drawings that feature the desired aspect ratio.

3.1.2 Considering Actual Node Dimensions

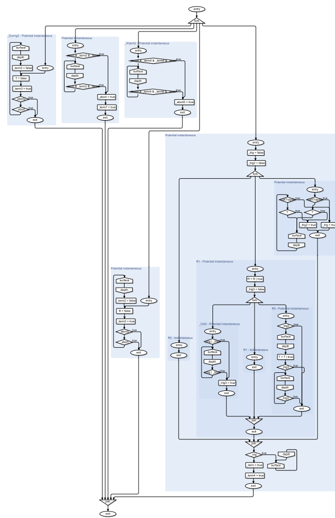
During the previous sections regular nodes and dummy nodes were considered to have the same dimensions. In practice, however, the dimensions of a graph's nodes may vary significantly depending on the application the graph originates from. Furthermore, the introduced dummy nodes effectively represent edges and usually require less space in a final drawing than regular nodes. An example of this can be seen in Figure 3.5. The depicted graph also contains hierarchical nodes.

Nikolov et al. already consider different node heights during their motivation of the problem but return to a unit height as soon as they discuss the heuristics. The remainder of this section discusses how actual node heights can be incorporated into each of the three heuristics. Note that it is essential that the node promotion heuristic uses the same, or at least a very similar, notion of a layer's height as the prior layering heuristic. Otherwise, it is not possible to prevent nodes from being promoted that would inadvertently increase the height of the layering. The alterations to incorporate varying node dimensions are similar for MW and SW, the next paragraph therefore starts with a general explanation before details specific to each method are outlined.

The two heuristics essentially use the out-degree and the in-degree of a node v as predictions for the number of introduced dummy nodes, i. e. the contribution to the height of layers adjacent to v's layer. Let $h(v)$ denote the height of a node v and let $\check{h}$ denote the smallest height of any node of a graph. During a pre-processing step, the node heights are normalized with respect to $\check{h}$. The normalized heights are referred to as $\overline{h}(v)$. Furthermore, let the height of dummy nodes h^d be a user-specified minimum separation



(a) NS



(b) MinWidth and PromoteNodes

Figure 3.5. Example of an SCG drawn with different layering methods. Depending on the available drawing area, one can imagine that either (a) or (b) is advantageous. For instance, (b) can be scaled larger when fitted on a sheet of paper in portrait orientation. Node labels would thus be more legible. Remember that as SCGs are drawn top-down, the maximum number of nodes in any layer determines the drawing's width. Also, the graph is hierarchical and is laid out in a bottom-up fashion using multiple layout executions.

between edges and be normalized as well. Wherever necessary, one can have individual heights for dummy nodes, e. g. if edges have different line widths. This is neglected during the upcoming explanations since it is easy to incorporate. Now, let h_c denote the height of the currently constructed layer and h_u denote the predicted height of future layers. Finally, let $\overline{d^-}(G)$ be the average in-degree of all nodes in G and let $\overline{d^+}(G)$ be the average out-degree. With these definitions, the decision whether to start a new layer can be adjusted for both MW and SW in the way discussed below. Additionally, the parts of the original heuristics that keep track of the current layer's height h_c and the predicted height of future layers h_u must multiply the node degrees by h^d and use the real height of a node. This can, however, be amended straightforwardly. Apart from these two modifications, the way the node is selected to be placed next remains unaltered.

MinWidth Let ubh and ubc be the two input parameters to MW as discussed in Section 3.1.1. Further, let v be the node currently looked at.

Size-Un-Aware The original condition to start a new layer as defined by Nikolov et al. is:

$$\begin{aligned} & h_c \geqslant ubh \wedge d^+(v) < 1 \\ \text{or} \quad & h_u \geqslant ubh \cdot ubc \end{aligned}$$

Size-Aware This is changed to consider actual node heights as follows. The upper bound on the height of a layer ubh is multiplied by the average height of all nodes of the graph. The resulting value is denoted by ubh'. If a node's height exceeds ubh', it can still be placed since MW first assigns nodes to the current layer and then checks the condition.

$$\begin{aligned} & h_c \geqslant ubh' \wedge d^+(v) \cdot h^d < \overline{h}(v) \\ \text{or} \quad & h_u \geqslant ubh' \cdot ubc \end{aligned}$$

StretchWidth Let h_m denote the current maximally allowed height of a layer that is iteratively increased if it is too small (cf. Section 3.1.1). Further, let v be the node currently looked at.

Size-Un-Aware The original condition to start a new layer as defined by Nikolov et al. is:

$$\begin{array}{rcl} h_c - d^+(v) + 1 & > & h_m \\ \text{or} \quad h_u + d^-(v) & > & h_m \cdot \overline{d^+}(G) \end{array}$$

Size-Aware This is changed to consider actual node heights as follows:

$$\begin{array}{rcl} h_c - d^+(v) \cdot h^d + \overline{h}(v) & > & h_m \\ \text{or} \quad h_u + d^-(v) \cdot h^d & > & h_m \cdot \overline{d^+}(G) \cdot h^d \end{array}$$

During the evaluations it could be observed that the second part of the condition to start a new layer is hardly true as h_u is often too small. A brief evaluation of a further modification seemed to improve things slightly but would require more thorough evaluations to make a final statement: In the original version of the algorithm, the h_u variable only accumulates the in-degrees of the nodes placed within the current layer and is reset every time a new layer is started. As a consequence, long edges spanning multiple layers are only considered once and neglected afterwards, i. e. they only contribute height to one layer no matter how long they are. An alternative could be to not reset the h_u variable when a new layer is started but to "correct" it instead: once a node v is placed in the current layer, v's out-degree is subtracted from h_u and its in-degree is added.

PromoteNodes The PN heuristic tries to improve existing layerings by reducing the overall number of dummy nodes. When applied after either MW or SW, which try to create layerings with small height, it is sensible to prevent PN from increasing the layering's height again.

As soon as MW and SW are aware of the height of regular nodes and dummy nodes, PN must consider this as well. It is not hard to incorporate this into the heuristic. The height of every layer of the given layering can be computed upfront, and it makes no difference whether regular nodes and dummy nodes factor in with a height of one or an individual height. Then, while promoting nodes, it can be checked if moving a node would increase the maximal height of the original layering.

Results The example in Figure 3.5 highlights the importance of considering actual node dimensions: Note that the depicted graph is hierarchical and is laid out in a bottom-up fashion, consecutively executing the layout algorithm for the different hierarchy levels. As such, the shaded areas representing hierarchical nodes are at some point "black boxes" from the perspective of the layout algorithm and are significantly larger than simple nodes. Looking at the top-level graph with nine nodes and twelve edges, the estimated height of the layering in both (a) and (b) is five when individual node heights are not considered. In their original form, neither MW nor SW would create the drawing (b), which in terms of pixels is significantly narrower.

The enhanced versions of MW and SW were evaluated using the set of SCGs, as introduced at the beginning of this section. Four configurations were used: either without node promotion (NONP) or with node promotion (NP), and either without size-awareness or with size-awareness (SA). Remember that SCGs are drawn top-down and that the test graphs were assembled such that they are rather wide (large aspect ratio). This time the goal is to produce drawings that fit computer screens in portrait-orientation, i. e. $\mathcal{R}_{(10,16)}$. The results are presented in Figure 3.6. The boxplot shows max scale ratio values relative to the results produced with the NS approach. The size-aware heuristics clearly produce better drawings for the defined setting when compared to the original versions, with drawings that can be displayed about 1.5 times larger. When not considering node sizes, the results of the heuristics are worse than NS for the majority of the graphs. No clear winner can be identified between MW and SW based on the tested SCGs. The execution of a subsequent node promotion only improves matters for SW. However, the latter two observations are likely to depend on the particular set of graphs.

3.1.3 Discussion

The findings of these sections can be summarized by three points. First, for many of the RomeF graphs a noteworthy gap exists between the heuristic and an optimal solution. Nevertheless, it is not clear if optimal solutions are always desirable. After all, a very narrow left-to-right drawing may corrupt

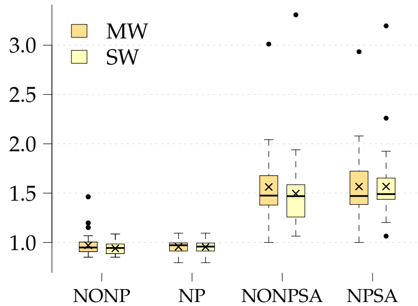


Figure 3.6. Assessing the impact of considering individual node sizes based on the set of SCGs. A reference frame $\mathcal{R}_{(10,16)}$ is assumed. The plotted max scale ratio values (y axis) are relative to NS's results.

other important layout aesthetics, such as the layering's width, the number of edge crossings, and the overall edge length. Second, in the general case the layering heuristics perform inferior to the traditional layering method NS. However, when used with care, MW and SW are able to produce drawings that are well suited for certain drawing areas. Third, the heuristics can be extended to consider individual node dimensions. The experiments with a set of SCGs suggest that this is both necessary and successful.

The section shall close with a proposal for future work in this area. Both heuristics are based on the idea of the LP algorithm. Superfluous dummy nodes are removed by the node promotion post-processing step. To a certain extent, the latter point mimics the underlying goal of NS: minimizing the number of dummy nodes. It has been shown that NS produces good and compact layerings [HN02b]. Therefore, it may be more promising to build a heuristic on the idea of NS from the start. Another point to consider is whether MW's parameters should be linked to the graph instance at hand instead of being absolute values. The parameter values suggested by Nikolov et al. are based on extensive experiments with the RomeF graphs [NTB05], which have a specific range of node and edge counts and are sparse. Denser graphs may require larger parameter values for good results.

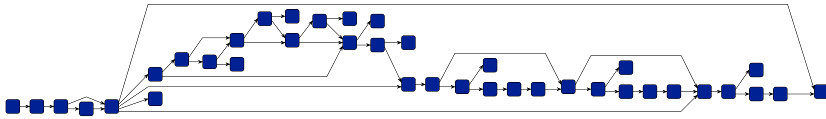
3.2 A Generalization of the Directed Layering Problem

The observations of the previous section led to the conclusion that the traditional definition of a layering, where all edges point forwards and the input graph must be acyclic, is not flexible enough to address scenarios in which compactness is of high importance. Also, while the layer-based approach is explicitly designed for directed graphs and emphasizes inherent directionality, the approach can well be used to lay out undirected graphs and graphs where the direction is of minor or no importance such that compactness takes precedence over directionality.

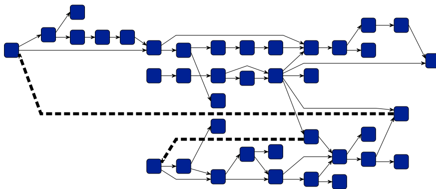
This section therefore proposes to relax the definition of a layering when faced with challenging graph instances or peculiar drawing properties: It presents a new layer assignment method, the *Generalized Layering Problem (GLP)*, that can handle cyclic graphs and that is able to consider compactness properties for selecting an edge reversal set. It addresses the posed challenges P-CR1, P-LA1, and P-LA2 (see Items P-CR1 and P-LA1). To elaborate: First, it can overcome the lower bound on the number of layers arising from the longest path of a graph, second, it can be flexibly configured to either favor elongated or narrow drawings, thus improving on aspect ratio, and third, compared to previous methods, it is able to reduce both the number of dummy nodes and the number of reversed edges for certain graphs simultaneously. See Figures 3.7 and 3.8 for examples.

The section proceeds as follows. First, preliminaries are discussed, followed by an integer programming model and a heuristic to solve the new method's underlying objective. Both are then evaluated on various sets of graphs. The evaluations serve as a basis to assess the flexibility of GLP to target drawing areas with certain aspect ratios. Last, a specialized variant of GLP that directly optimizes the max scale measure is discussed and evaluated. The results of this variant indicate that there is room for improvement and led to the research discussed in Section 3.3.

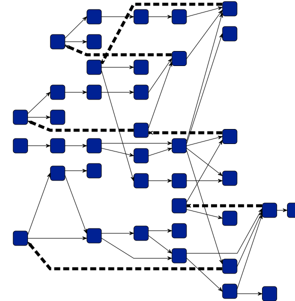
The NS approach of Gansner et al. assigns nodes to layers such that the overall edge length is minimized [GKN+93]. Layerings created with NS are generally already compact [HN02b], which is why it serves as the basis for



(a) 0 reversed edges, 71 dummy nodes

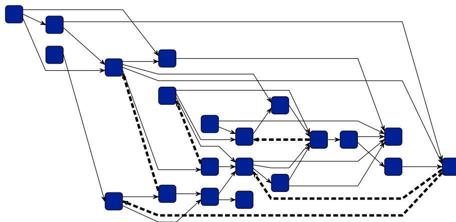


(b) 2 reversed edges, 35 dummy nodes

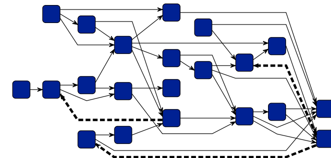


(c) 6 reversed edges, 16 dummy nodes

Figure 3.7. Different drawings of the `g.39.29` graph from the North graphs collection (see Section 1.9). (a) is drawn with NS, (b) and (c) are drawn with GLP. Reversed edges are bold and dashed.



(a) 5 reversed edges, 55 dummy nodes



(b) 3 reversed edges, 34 dummy nodes

Figure 3.8. A graph drawn with (a) EaGa (Eades et al.'s cycle removal [ELS93] and NS) and (b) GLP. This example illustrates that GLP can perform better in both metrics: reversed edges (bold and dashed) and dummy nodes.

the new method. The following problem formalizes the NS approach. Here, a layering L is said to be *valid* if $\forall(u,v) \in E$: $L(v) - L(u) \geqslant 1$.

PROBLEM DIRECTED LAYERING (DLP)

> Let $G = (V, E)$ be an acyclic directed graph. The problem is to find a minimum k and a valid layering L such that $\sum_{(u,v)\in E}(L(v) - L(u)) = k$.

For GLP the requirement of a graph being directed and acyclic is dropped. The first step of the layer-based approach, removing cycles, is consequently merged with the layer assignment step. Undirected graphs can be handled by assigning an arbitrary direction to each edge, thus converting it into a directed one, and by hardly penalizing reversed edges. A layering L of a general graph is called *feasible* if $\forall\{u,v\} \in E : |L(u) - L(v)| \geqslant 1$. The problem then is:

PROBLEM GENERALIZED LAYERING (GLP)

> Let $G = (V, E)$ be a possibly cyclic directed graph and let $\omega_{\text{len}}, \omega_{\text{rev}} \in \mathbb{N}$ be weighting constants. The problem is to find a minimum k and a feasible layering L such that

$$\omega_{\text{len}} \left(\sum_{(u,v)\in E} \big|L(v) - L(u)\big| \right) + \omega_{\text{rev}} \big|\{(u,v) \in E : L(u) > L(v)\}\big| \;=\; k \;.$$

Intuitively, the left part of the sum represents the overall edge length (i. e. the number of dummy nodes) and the right part represents the number of reversed edges (i. e. the FAS). After reversing all edges in this FAS, the feasible layering becomes a valid layering. Compared to the standard cycle removal step combined with DLP, the generalized layering problem allows more flexible decisions on which edges to reverse. Also note that GLP with $\omega_{\text{len}} = 1, \omega_{\text{rev}} = \infty$ is equivalent to DLP for acyclic input graphs and that while DLP is solvable in polynomial time, both parts of GLP are NP-complete [RES+15].

3.2.1 IP Approach

Next, an integer programming model is presented that solves GLP. The rough idea of the model is to assign an integer value to each node of the given graph that represents the layer in which that node is to be placed.

Input and Parameters Let $G = (V, E)$ be a graph with node set $V = \{1, \dots, n\}$. Let e be the graph's adjacency matrix, i. e. $e(u, v) = 1$ if $(u, v) \in E$ and $e(u, v) = 0$ otherwise. ω_{len} and ω_{rev} are weighting constants larger than zero.

Integer Decision Variables $l(v)$ takes a value in $\{1, \dots, n\}$ indicating that node v is placed in layer $l(v)$, for all $v \in V$.

Boolean Decision Variables $r(u, v) = 1$ if and only if edge $e = (u, v) \in E$ and e is reversed, i. e. $l(u) > l(v)$, for all $u, v \in V$. Otherwise, $r(u, v) = 0$.

Objective

$$\text{Minimize} \qquad \omega_{\text{len}} \sum_{(u,v) \in E} |l(u) - l(v)| + \omega_{\text{rev}} \sum_{(u,v) \in E} r(u, v).$$

The sums represent the edge lengths, i. e. the number of dummy nodes, and the number of reversed edges, respectively.

Constraints

$$1 \leqslant l(v) \leqslant n \qquad \forall v \in V \qquad \text{(A)}$$

$$|l(u) - l(v)| \geqslant 1 \qquad \forall (u, v) \in E \qquad \text{(B)}$$

$$n \cdot r(u, v) + l(v) \geqslant l(u) + 1 \qquad \forall (u, v) \in E \qquad \text{(C)}$$

Constraint (A) restricts the range of possible layers. (B) ensures that the resulting layering is feasible. (C) binds the decision variables in r to the layering; because r is part of the objective, and $\omega_{\text{rev}} > 0$, $r(u, v)$ gets assigned 0 unless $l(v) < l(u)$, for all $(u, v) \in E$.

Variations The model can easily be extended to restrict the number of layers by replacing the n in Constraint (A) by a desired bound $b \leqslant n$. The edge matrix can be extended to contain a weight $w_{u,v}$ for each edge $(u, v) \in E$. This can be helpful if further semantic information is available, e. g. about feedback edges that lend themselves well to be reversed.

Jabrayilov et al. present two MIP models for slight variations of GLP [JMM+16]. The first one, called CGL, considers the contribution of dummy nodes to a layer's height and adds the overall height of the layering to the objective function. Consequently, a reasonable bound on the width must be set, which is part of the input. The second one, called MML, neglects the contribution of dummy nodes to a layer's height and maximizes the length of reversed edges, significantly improving on execution time. For certain use cases the reversed edge length maximization may be desired.

3.2.2 Heuristic Approach

Interactive modeling tools providing automatic layout facilities require execution times significantly shorter than one second. As the IP formulation discussed above rarely meets this requirement, a heuristic to solve GLP is presented next. It proceeds as follows:

1. Leaf nodes are removed iteratively, since it is trivial to place them with minimum edge length and a desired edge direction. Note that therefore the heuristic is not able to improve on trees that yield a poor compactness. As outlined in Section 2.2, a compact layered layout of (sub-)trees cannot be created straightforwardly, the issue is therefore left for future research.
2. An initial feasible layering is constructed for the possibly cyclic input graph, which is used to deduce edge directions yielding an acyclic graph.
3. Using NS, a solution with minimal edge length is created.
4. A greedy improvement procedure is executed, after which edge directions are deduced again and leaves are re-attached.
5. NS is executed a second time to get a proper layering with minimal edge lengths for the next steps of the layer-based approach.

The next paragraphs discuss steps 2 and 4 in further detail.

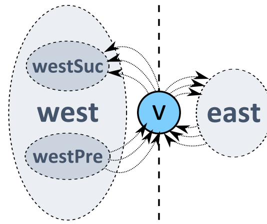


Figure 3.9. Illustrating the regions used by the heuristic.

Step 2: Layering Construction The construction of an initial feasible solution is based on an idea that was first presented by McAllister as part of a greedy heuristic for the linear arrangement problem (LAP) [McA99] and was later extended by Pantrigo et al. [PMD+12].

Nodes are assigned to distinct indices, where as a start, a node is selected randomly, assigned to the first index, and added to a list of assigned nodes. Based on the list of assigned nodes, a candidate list is formed and the most promising node is assigned to the next index. The decision criterion is the difference between the number of edges incident to unassigned nodes and the number of edges incident to assigned nodes. This procedure is repeated until all nodes are assigned to distinct indices (see Algorithm 3.1).

In contrast to McAllister, the GLP heuristic can add nodes either at the beginning of the list of assigned nodes or at the list's end. The side is decided based on the number of reversed edges that would emerge from placing a certain node on that side. In the pseudo code the beginning of the list is represented by the decreasing left index variable $lIndex$, and the end of the list is represented by the increasing right index variable $rIndex$.

Step 4: Layering Improvement At this point, a feasible layering with a minimum number of dummy nodes with respect to the chosen FAS is given since NS ran beforehand. Thus, only the number of reversed edges can be improved, which is done by identifying possible *moves* and by deciding whether to take a move based on a *profit* value. The profit value captures the trade-off between short edges and few reversed edges. For ease of presentation, consider the following notions, also illustrated in Figure 3.9.

Algorithm 3.1: Feasible Layering Construction

Input: Directed graph $G = (V, E)$
Data: Sets U, C. For all $v \in V$: $score[v]$, $incAs[v]$, $outAs[v]$
$lIndex \leftarrow -1$, $rIndex \leftarrow 0$
Output: $index[v]$: feasible layering of G

```
1  for v ∈ V do
2      score[v] ← |{w | {v,w} ∈ E}|
3      incAs[v] ← 0
4      outAs[v] ← 0
5      Add v to U
6  Remove random v from U
7  c ← v
8  Add c to C
9  while U not empty do
10     if incAs[c] < outAs[c] then
11         index[c] ← lIndex−−
12     else
13         index[c] ← rIndex++
14     Remove c from U and C
15     cScore ← ∞
16     for v ∈ {w | {c,w} ∈ E ∧ w ∈ U} do
17         Add v to C
18         score[v]−−
19         if (c,v) ∈ E then incAs[v]++ else outAs[v]++
20     for v ∈ C do
21         if score[v] < cScore then cScore ← score[v]
22         c ← v
```

An example: For a node v, *westSuc* are the nodes connected to v via an outgoing edge of v and are currently assigned to a layer with lower index than v's index. Intuitively, *westSuc* (just as *eastPre*) are nodes connected by an edge pointing into the "wrong" direction.

$$
\begin{aligned}
v.westSuc &= \{w : (v,w) \in E \wedge L(v) > L(w)\} & v.eastSuc &= \{w : (v,w) \in E \wedge L(v) < L(w)\} \\
v.westPre &= \{w : (w,v) \in E \wedge L(w) < L(v)\} & v.eastPre &= \{w : (w,v) \in E \wedge L(w) > L(v)\} \\
v.westAdj &= v.westSuc \cup v.westPre & v.eastAdj &= v.eastSuc \cup v.eastPre
\end{aligned}
$$

For every function a suffix exists allowing to query for a certain set of nodes before or after a certain index. For instance, all west successors of v before index i are $v.westSucBefore(i) = \{w : w \in v.westSuc \wedge L(w) < i\}$.

Now, let $move : V \to \mathbb{N}$ denote a function assigning a natural value to each node. The function describes whether it is possible, and if it makes sense, to move a node without violating the layering's feasibility as well as how far the node should be moved. For instance, let *westPre* be empty for a node v, and *westSuc* be non-empty. In that case, v can be moved to an arbitrary layer with lower index than $L(v)$. A good choice would be exactly one layer before any $w \in v.westSuc$ since this would alter the incident edges to all point into the desired direction.

$$
move(v) = \begin{cases} 0 & \text{if } v.westSuc = \emptyset, \\ L(v) - \min(\{L(w) : w \in v.westSuc\}) + 1 & \text{if } v.westPre = \emptyset, \\ L(v) - \max(\{L(w) : w \in v.westPre \setminus v.westSuc\}) - 1 & \text{otherwise,}^3 \end{cases}
$$

where $\max(\emptyset) = L(v) - 1$.

Let $profit : V \times \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$ denote a function assigning a quality score to a node v that decides if it is worth to move v and increase the length of some edges for a subset of the edges to point into the desired direction. For this, let the natural m be the value computed by the *move* function defined above, and let x represent the layer v would be moved to. Note that ω_{len} and ω_{rev} are used here but they were not expected to allow fine-grained control as for the IP. The evaluations verified this. The discussed results are

[3] During the evaluations, $L(v) - \max(\{L(w) : w \in v.westPre\}) - 1$ was used as the third case. This did not necessarily preserve the layering's feasibility but was turned into a proper layering by the subsequent NS execution in either case.

Algorithm 3.2: Layering Improvement

Input: Feasible layering of $G = (V, E)$ in $index[v]$
Data: Priority queue PQ
For all $v \in V$: $move[v]$, $profit[v]$
Output: $index[v]$: feasible layering of G

```
1  for v ∈ V do
2      move[v] ← move(v)
3      profit[v] ← profit(v, move[v], index[v] − move[v])
4      if profit[v] > 0 then Enqueue v to PQ
5  while PQ not empty do
6      v ← Dequeue PQ
7      index[v] −= move[v]
8      for w ∈ {w | {v, w} ∈ E} do
9          Update move[w] and profit[w]
10         if profit[w] > 0 then
11             Enqueue/update w to/in PQ
12         else
13             Possibly dequeue w from PQ
```

for values of 1 and 5.

$$profit(v, m, x) = \begin{cases} 0 & \text{if } m \leqslant 1, \\ \omega_{\text{len}}(m|v.\mathit{westAdjBefore(x)}| - m|v.\mathit{eastAdj}|) & \\ + \ \omega_{\text{rev}}|v.\mathit{westSucAfter(x)}| & \text{otherwise.} \end{cases}$$

As seen in Algorithm 3.2, the *move* and *profit* functions are computed initially for a given feasible layering. A queue, sorted based on profit values, is then used to successively perform moves that yield a profit. After a move of node n, both functions can be updated for all nodes in n's adjacency.

Time Complexity Removing leaf nodes requires linear time, $O(|V| + |E|)$. Algorithm 3.1 is quadratic in the number of nodes, $O(|V|^2)$: The while loop has to assign an index to every node and the two inner for loops are, for a complete graph, iterated $|V|/2$ times on average. Note that determining the next candidate (lines 20–22) could be accelerated using dedicated data structures. The runtime of the improvement step strongly depends on the input

graph, however, the way the move function is defined, no edge is reversed more than once. Also, a very loose bound on the runtime is $O(|E|^2 \log |V|)$ since the while loop is iterated at most $|E|$ times, neighborhood updates can be realized such that for each move they are bound by the total number of edges, and priority queue operations can be implemented with $O(\log |V|)$. The NS method runs reportedly fast in practice [GKN+93]. The evaluations showed that the heuristic's overall execution time is clearly dominated by the NS method.

3.2.3 Evaluation

The main measures of interest here are height, area, and aspect ratio. Remember that the layer-based approach is defined as a pipeline of several independent steps and that at this point the area and aspect ratio of a final drawing can only be estimated using the number of dummy nodes, the number of layers, and the maximal number of nodes in a layer.

This section evaluates three points:

1. the general feasibility of GLP to improve the compactness of drawings,
2. the quality of the used estimates of the area and aspect ratio as opposed to effective values measured in pixels, and
3. the performance of the presented IP and heuristic.

The results can be seen in Tables 3.1 and 3.2 and are discussed in more detail in the next paragraphs.

Obtaining a Final Drawing In order to collect all desired metrics, a final drawing of a graph is required. As outlined in the previous chapter, over time numerous strategies have been presented for each step of the layer-based approach that can be combined in different ways. Here, cycle removal is performed by the heuristic of Eades et al. [ELS93]. Layerings are computed either by the newly presented approach GLP (both the IP method and heuristic, denoted by GLP-IP and GLP-H) or alternatively the NS approach. The combination of Eades et al.'s cycle removal and the NS layering of Gansner et al. is written as EaGa. Note that in combination it

is an alternative to GLP which performs the cycle removal itself. Crossings between pairs of layers are minimized using a layer-sweep method in conjunction with the barycenter heuristic, as originally proposed by Sugiyama et al. [STT81]. Two different strategies are used to determine node coordinates: First, a method introduced by Buchheim et al. that was extended by Brandes and Köpf [BJL01; BK02], which is denoted by BK. It is also discussed and improved in Section 4.2 of this thesis. Second, LS, a method inspired by Sander [San96a]. Edges are routed either using polylines (Poly) or orthogonal segments (Orth). The orthogonal router is based on the methods presented by Sander [San04]. Overall, this gives twelve setups of the algorithm: three layering methods, two coordinate assignment algorithms, and two edge routing procedures. In the following, let ω_{len}-ω_{rev}-GLP denote the used weights for GLP. If GLP is not further qualified, the IP model is meant.

Test Graphs The new approach is intended to improve the drawings of graphs with a large width and relatively small height, hence unfavorable aspect ratio. Nevertheless, a set of 160 randomly generated graphs is used as well to evaluate the generality of the approach. The graphs have 17 to 60 nodes and an average of 1.5 edges per node. They were generated by creating a number of nodes, assigning out-degrees to each node such that the sum of outgoing edges is 1.5 times the nodes, and finally creating the outgoing edges with a randomly chosen target node. Unconnected nodes were removed. Second, the set of North graphs was filtered based on the aspect ratio of a final drawing created using BK and Poly. 146 graphs that have at least 20 nodes and a drawing with an aspect ratio above 2, i. e. at least twice as wide as high, were selected. The filtered set is referred to ATTar (cf. Section 1.9).

Additionally, plain paths were removed, that is, pairs of nodes connected by exactly one edge, as well as trees. For these special cases GLP in its current form would not change the resulting number of reversed edges as all edges can be drawn with length one. This is also true for any bipartite graph. Note however that GLP can easily incorporate a bound on the number of layers which can straightforwardly be used to force more edges to be reversed, resulting in a drawing with better aspect ratio.

Table 3.1. Results for final drawings of the set of random graphs when applying different layout strategies. For GLP-IP $\omega_{\text{len}} = 1$ and $\omega_{\text{rev}} = 30$ were used. Area is normalized by a graph's node count. The most interesting comparisons are between columns where EaGa and GLP use the same strategies for the remaining steps.

Edge routing	Poly						Orth					
Node coord.	BK			LS			BK			LS		
Layering	EaGa	GLP-IP	GLP-H	EaGa	GLP-IP	GLP-H	EaGa	GLP-IP	GLP-H	EaGa	GLP-IP	GLP-H
Width	1,165	1,043	898	943	824	732	790	711	652	817	746	678
Area	20,194	18,683	15,575	12,383	11,035	10,075	13,582	12,642	11,272	10,666	9,917	9,295
Aspect ratio	1.69	1.49	1.49	1.82	1.56	1.56	1.19	1.04	1.11	1.59	1.43	1.47

Table 3.2. Average values for different layering strategies employed to the test graphs. Different weights are used for GLP-IP as specified in the column head, and final drawings were created using BK and Poly. For GLP-H* no improvement was performed.

a. Random graphs

	1-10	1-20	1-30	1-40	1-50	EaGa	GLP-H	GLP-H*
Reversed edges	3.71	2.89	2.64	2.54	2.44	2.93	8.67	10.36
Dummy nodes	34.45	46.73	52.79	56.14	60.53	72.64	48.48	58.21
Edge crossings	2.41	2.18	2.10	2.04	2.02	1.98	1.93	1.84
Width	843	943	980	1,004	1,025	1,084	930	1,027
Area	631,737	672,717	691,216	700,385	708,361	737,159	656,070	720,798
Aspect ratio	1.3	1.54	1.59	1.64	1.69	1.82	1.49	1.67

b. ATTar graphs

	1-10	1-20	1-30	1-40	1-50	EaGa	GLP-H	GLP-H*
Reversed edges	2.74	1.47	1.02	0.72	0.56	0	7.07	8.55
Dummy nodes	39.91	55.47	65.73	75.66	82.47	141.30	53.53	68.91
Edge crossings	1.46	1.23	1.13	1.06	0.99	0.86	1.19	1.12
Width	1,068	1,224	1,334	1,409	1,469	1,727	1,137	1,216
Area	587,727	622,838	641,581	660,842	695,494	874,374	629,778	691,372
Aspect ratio	2.94	3.57	4.17	4.35	4.55	5.00	3.03	3.12

General Feasibility of GLP An exemplary result of the GLP approach compared to EaGa can be seen in Figure 3.8. For that specific drawing, GLP produces fewer reversed edges, fewer dummy nodes, and less area (both in width and height). For all tested setups the average effective width and area (normalized by the number of nodes) of GLP and the heuristic are smaller than EaGa's, see Table 3.1. The average aspect ratios decrease, i. e. the drawings change in the desired way.

Furthermore, altering the weights ω_{rev} and ω_{len} allows a trade-off between reversed edges and resulting dummy nodes (and thus area and aspect ratio). This can be seen well in Table 3.2a for the random graphs.

The results for the ATTar graphs are similar. Since the ATTar graphs are acyclic, the cycle removal step is not required, and current layering algorithms cannot reduce the width. The GLP approach, however, can freely reverse edges and hereby change the width and aspect ratio. Results can be seen in Table 3.2b. Clearly, EaGa has no reversed edges as all graphs are acyclic. 1-10-GLP starts with an average of 2.7 reversed edges, and the value constantly decreases with an increased weight on reversed edges. The number of dummy nodes on the other hand constantly decreases from 141.3 for EaGa to 39.9 for 1-10-GLP. For both sets of graphs the average number of edge crossings per edge slightly increases when a lower weight is set for ω_{rev}. This makes sense since more edges have to be routed between a lower number of layers. The average width and average area of the final drawings decrease with an increasing number of reversed edges. For 1-10-GLP the average width and area are 38.2 % and 33.8 % smaller than EaGa. The aspect ratio changes from an average of 5.00 for EaGa to 2.94 for 1-10-GLP.

The results show that for the selected graphs, for which current methods cannot improve on width, the weights of the new approach allow to find a satisfying trade-off between reversed edges and dummy nodes. Furthermore, the improvements in compactness stem solely from the selection of weights, not from an upper bound on the number of layers. Naturally, such a bound can further improve the width and aspect ratio.

Metric Estimations Table 3.1 presents results that were measured on the final drawing of a graph. As mentioned earlier, these values are not available when analyzing the result of the layering step, and estimations are

commonly used to deduce the quality of a result. For the example graphs, the estimated area decreases from 222.9 (EaGa) to 187.4 (1-30-GLP) on average. The estimated aspect ratios decrease on average from 1.35 to 1.19. Note that these estimations are not included in the table. Both tendencies conform to the averaged effective values in Table 3.1, i. e. GLP-IP and the GLP-H perform better. However, for 64 % of the graphs the tendency of the estimated area contradicts the tendency of the effective area (using BK and Poly), and for 54 % when not considering dummy nodes. In other words, for a specific graph the estimated area might be decreased for GLP compared to EaGa but the effective area is increased for GLP (or vice versa). This clearly indicates that an estimation can be misleading. Besides, coordinate assignment and edge routing can have a non-negligible impact on the aspect ratio and compactness of the final drawing.

Performance of the Heuristic Results for final drawings using the presented heuristic are included in Table 3.1 and are comparable to 1-30-GLP, i. e. the heuristic performs better than EaGa with respect to the desired metrics. Table 3.2a and Table 3.2b underline this result and also show that the improvement step of the heuristic clearly improves on all measured metrics. On the downside, the heuristic clearly yields more reversed edges.

Execution Times The IP model was solved using CPLEX 12.6 and executed on a server with an Intel Xeon E5540 CPU and 24 GB memory. The execution times for GLP-IP vary between 476ms for a graph with 19 nodes and 541s for a graph with 58 nodes and exponentially increase with the graph's node count. This is impracticable for interactive tools from practice that rely on responsive automatic layout, but is fast enough to collect optimal results of medium sized graphs for research purposes.

The execution time of the heuristic is compared to EaGa and was measured on a laptop with an Intel i7 2GHz CPU and 8GB memory. The reported time includes only the first two steps of the layer-based approach. It turns out that the execution time of the heuristic is on average 2.3 times longer than EaGa. This seems reasonable, as it involves two executions of the NS layering method. For the tested graphs, the construction and

improvement steps of the heuristic hardly contribute to its overall execution time. The effective execution time ranges between 0.1ms and 10.0ms for EaGa and 0.3ms and 19.7ms for the heuristic. The heuristic is thus fast enough to be used in interactive tools.

The algorithm was also run five times for five randomly generated graphs with 1000 nodes and 1500 edges. EaGa required an average of 374ms, the heuristic 666ms with about 4ms for layering construction and 2ms for improvement. This shows that the time contribution of the latter two is negligible even for larger graphs.

3.2.4 Targeting a Specific Drawing Area

The previous section showed that GLP allows to reduce the number of dummy nodes by increasing the number of reversed edges. Adjusting the weights ω_{rev} and ω_{len}, the aspect ratio of the resulting drawing can be influenced to a certain extent. This section investigates in further detail whether GLP is able to create drawings that are particularly suited for a prescribed drawing area. The measures used so far are not sufficient for this. Therefore, the max scale values of the final drawings created for the ATTar graphs as discussed in Section 3.2.3 are measured based on the width and height in pixels for four different exemplary reference frames with aspect ratios of 0.5, 1.0, 2.0, and 4.0. The results can be seen in Figure 3.10. Remember that no comparison can be made across target aspect ratios but only between the results of different layering methods for the same target aspect ratio.

For a target aspect ratio of 4.0 EaGa shows the best max scale values, which is not surprising since the set of test graphs was selected based on EaGa producing a drawing with an aspect ratio above 2.0. Also, the GLP configurations aim at producing horizontally narrower drawings. For aspect ratios smaller than 4.0, all variants of GLP produce better max scale values than EaGa. Also, the selected weights gradually change the max scale value, which is coherent with the observations for width, area, and aspect ratio in Section 3.2.3. However, the configuration with the smallest weight on reversed edges, 1-10-GLP, gives the best max scale values for three out of the four reference frames. While this is not immediately surprising since

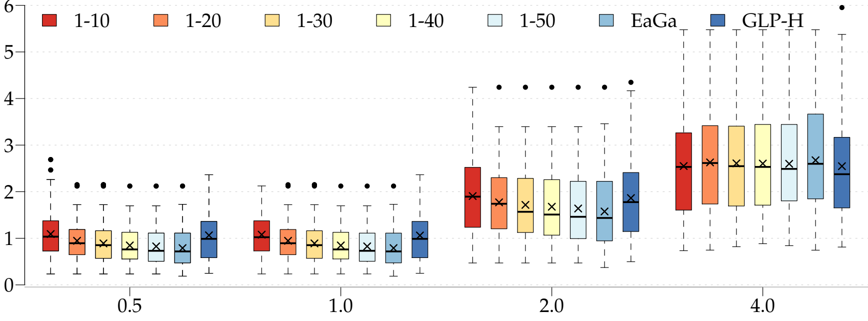


Figure 3.10. Computed max scale values (y axis) for four different target aspect ratios (x axis), using the results for the tested ATTar graphs and the GLP variants as discussed in Section 3.2.3.

1-10-GLP's drawings are the ones with the smallest area, it prevents a static mapping which weights should be used for which reference frames. Additionally, it leads to the question whether significantly larger max scale values are possible, and if so, what trade-offs have to be made to achieve them. The remainder of this section therefore investigates the following points:

1. How do GLP's drawings relate to drawings with optimal max scale values? How much better are the optimal ones?
2. How many edges have to be reversed to achieve optimal values?

To obtain optimal max scale values, GLP must be extended to incorporate the max scale measure into its objective:

$$\begin{aligned}\text{Minimize} \qquad & \omega_{\text{len}} \sum_{(v,w)\in E} \left|L(w) - L(v)\right| \\ & + \omega_{\text{rev}} \left|\{(v,w) \in E : L(v) > L(w)\}\right| \\ & + \omega_{\text{ms}} \max\left\{\frac{w(L)}{r_w}, \frac{h(L)}{r_h}\right\},\end{aligned}$$

where $\mathcal{R} = (r_w, r_h)$ denotes the desired reference frame, $w(L)$ the estimated width of the layering, and $h(L)$ the estimated height of the layering, which

in this case includes dummy nodes.

To solve this modified version, which is referred to as GLP-MS, the IP formulation presented in Section 3.2.1 is not particularly suited: The max scale measure requires reasonable estimates of the width and height of the drawing. This is only possible if the contributions of dummy nodes to the layer heights are considered. Thus, a more sophisticated MIP model (CGL) serves as a basis, which has been presented by Jabrayilov et al. [JMM+16] for a variation of GLP. The authors presume a top-down layout direction. To be consistent with the other parts of this thesis, their explanations are altered to a left-to-right direction in what follows.

The main difference between CGL and GLP is that Jabrayilov et al. set a fixed bound W on the width and add the overall height H of the layering to the objective. They make use of three types of binary variables. First, $y_{v,k}$ equals 1 if and only if $L(v) < k$ for all $v \in V$ and $1 \leqslant k \leqslant H$. The reverse variables $y_{k,v}$ (equal to 1 if and only if $k < L(v)$) are part of the model for ease of understanding but can be eliminated when implementing the model. Second, $r_{u,v}$ equals 1 if and only if the edge (u, v) is reversed. Third, $z_{uv,k}$ equals 1 if and only if the edge (u, v) produces a dummy node in layer k for all $2 \leqslant k \leqslant W - 1$.

To use CGL for GLP-MS, W is set to $|V|$, thus removing W from the input. Two new input variables are introduced to denote the reference frame, r_w and r_h. The height term of Jabrayilov et al.'s objective, $\omega_{\text{wid}} \cdot H$, is replaced by $\omega_{\text{ms}} \cdot MS$, where MS represents the inverse max scale value to be minimized and ω_{wid} and ω_{ms} are weighting constants. Further, the constraints of the original model that are connected to H are removed, while the model is extended by the following constraints to properly incorporate MS. Two dummy nodes s and e are added to the graph's nodes, which are used to mark the very first layer and the very last layer. A dummy edge (s, e) is added to the graph's edges. With this, the following constraints are added to bound MS (the full model and the computation of the height $h(k)$ of a layer k can be found in the next section):

$$y_{k,s} = 0 \quad \text{for all } 1 \leqslant k \leqslant W \tag{A}$$

$$y_{k,v} \leqslant y_{k,e} \quad \text{for all } 1 \leqslant k < W, v \in V \backslash \{e\} \tag{B}$$

$$\frac{h(k)-1}{r_h} \leqslant MS \quad \text{for all } 1 \leqslant k \leqslant W \tag{C}$$

$$\frac{1+\sum_{1\leqslant k\leqslant W} y_{k,e}}{r_w} \leqslant MS. \tag{D}$$

Constraints (A) and (B) assure that no original node is placed before and after the s node and the e node, respectively. Consequently, a dummy node is introduced in every intermediate layer. Since the number of dummy nodes is subject to minimization, s is placed in the first layer and e is placed in the last layer. Constraints (C) restrict MS by the height of the layers. The dummy nodes originating from (s, e) are discarded by the -1. Finally, (D) restricts MS by the layering's width, i. e. it counts the number of layers that are smaller than e's layer and adds the layer itself. It is possible to model the overall width and height of the layering like this, since every node and every dummy node is assumed to have a width and height of one.

Full GLP-MS MIP Model

For completeness, the full model is included below, for further details see the original paper [JMM+16]. Given is a graph $G = (V, E)$, which includes a dummy edge $(s, e) \in E$ as discussed above. Let $W = |V|$ and let r_w and r_h denote the input parameters specifying the drawing area.

$$\text{Minimize } \left(\omega_{len} \sum_{(u,v)\in E} \sum_{k=2}^{W-1} z_{uv,k}\right) + \left(\omega_{rev} \sum_{(u,v)\in E} r_{u,v}\right) + \omega_{\text{ms}}\, MS, \text{ s.t.}$$

$$y_{v,1} = 0 \quad \text{for all } v \in V \tag{E}$$

$$y_{W,v} = 0 \quad \text{for all } v \in V \tag{F}$$

$$y_{k,v} + y_{v,k+1} = 1 \quad \text{for all } v \in V, 1 \leqslant k \leqslant W-1 \tag{G}$$

$$y_{k+1,v} - y_{k,v} \leqslant 0 \quad \text{for all } v \in V, 1 \leqslant k \leqslant W-2 \tag{H}$$

$$-y_{u,k} - y_{k,v} - r_{u,v} \leqslant -1 \quad \text{for all } (u,v) \in E, 1 \leqslant k \leqslant W \tag{I}$$

$$-y_{k,u} - y_{v,k} + r_{u,v} \leqslant 0 \quad \text{for all } (u,v) \in E, 1 \leqslant k \leqslant W \tag{J}$$

$$y_{k,u} + y_{v,k} - z_{uv,k} \leqslant 1 \quad \text{for all } (u,v) \in E, 2 \leqslant k \leqslant W-1 \tag{K}$$

$$y_{k,v} + y_{u,k} - z_{uv,k} \leqslant 1 \quad \text{for all } (u,v) \in E, 2 \leqslant k \leqslant W-1 \tag{L}$$

$$
\begin{aligned}
& y_{k,s} && = 0 && \text{for all } 1 \leqslant k \leqslant W && \text{(M)} \\
& y_{k,v} && \leqslant y_{k,e} && \text{for all } 1 \leqslant k < W, v \in V \setminus \{e\} && \text{(N)} \\
& \frac{\left(\sum_{u \in V}(1 - y_{u,k} - y_{k,u})\right) - 1}{r_h} && \leqslant MS && \text{for all } k \in \{1, W\} && \text{(O)} \\
& \frac{\left(\sum_{u \in V}(1 - y_{u,k} - y_{k,u}) + \sum_{(u,v) \in E} z_{uv,k}\right) - 1}{r_h} && \leqslant MS && \text{for all } 2 \leqslant k \leqslant W - 1 && \text{(P)} \\
& \frac{1 + \sum_{1 \leqslant k \leqslant W} y_{k,e}}{r_w} && \leqslant MS && && \text{(Q)} \\
& y_{v,k},\ y_{k,v} && \in \{0, 1\} && \text{for all } v \in V, 1 \leqslant k \leqslant W && \\
& r_{u,v} && \in [0, 1] && \text{for all } (u,v) \in E && \\
& z_{uv,k} && \in [0, 1] && \text{for all } (u,v) \in E, 2 \leqslant k \leqslant W - 1 && \\
& MS && \in \mathbb{R}_{\geqslant 0} && &&
\end{aligned}
$$

Results

The model was executed using Gurobi 7[4] on a server with an Intel Xeon E5540 CPU and 24 GB memory. Within the set time limit of two hours, 354 of 438 executions of the ATTar graphs finished. That is, for each of the 146 graphs the model ran for three different aspect ratios. For 112 of the graphs all three executions finished. Divided by target aspect ratio 0.5, 1.0, and 2.0, execution took on average 710, 1.938, and 4.926 seconds, respectively.

The results discussed in the following are for the subset of the 112 graphs. The nodes of the ATTar graphs have no specified width and height. Both are set to 20 when measuring pixels and, as said before, have a unit width and height for GLP-MS. Thus, a dummy node contributes the same height to a layer as a regular node.

The weights ω_{len}, ω_{rev}, and ω_{ms} are selected such that the max scale measure is prioritized over both other terms and a reversed edge is penalized

[4] `http://www.gurobi.com/`

stronger than long edges:

$$\omega_{\text{len}} = 1, \quad \omega_{\text{rev}} = \omega_{\text{len}} \ |E| \ |V|, \text{ and } \quad \omega_{\text{ms}} = \omega_{\text{rev}}(1 + |E|) \max\{r_w, r_h\}.$$

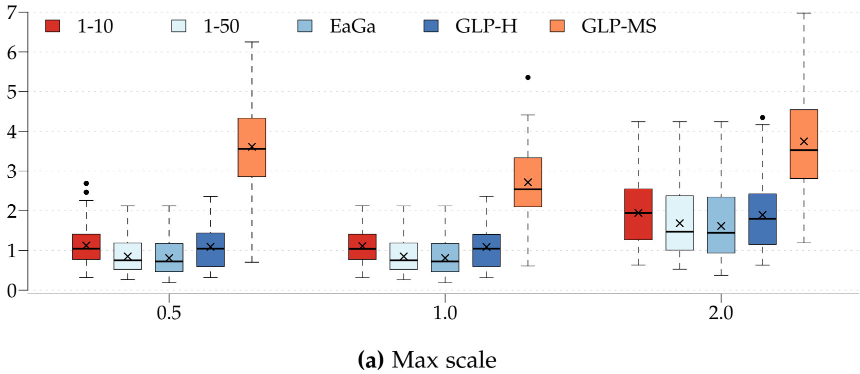
Note that care has to be taken when using large weights like this with MIP solvers. A solution is considered optimal by a solver if the relative gap between the lower and upper bound on the objective is lower than a specified parameter. Therefore, since the objective value is dominated by the max scale part, it can happen that the term measuring the edge length is neglected if the parameter value is too large. Gurobi's default 10^{-4} may cause issues with the graphs we tested, which is why we set it to 10^{-10}.

Remember that the results presented in Figure 3.10 suggest that while the differently weighted GLP is able to improve the max scale values when aiming for a certain drawing area no significant improvements can be made when only a small number of edges is reversed. This feels natural, as by reversing a small number of edges the overall character of the graph is not changed. A very "long" graph will never be suited for the opposite kind of drawing area.

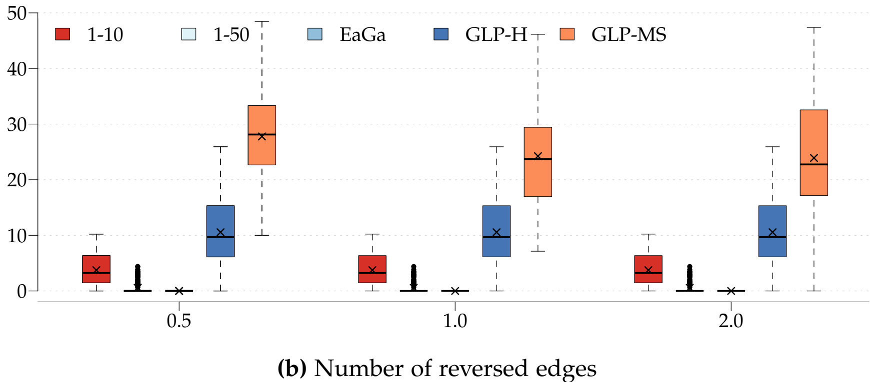
The results of applying GLP-MS to the ATTar graphs are shown in Figure 3.11. They clearly indicate that it is possible to achieve significantly larger max scale values for notably different drawing areas if enough edges may be reversed. For all three tested target aspect ratios, 0.5, 1.0, and 2.0, the max scale values created with GLP-MS are on average at least two times larger. Answering Question (1.) (p. 91), this indicates that GLP's results are still improvable when it comes to a specific drawing area. Nevertheless, answering Question (2.), it must be said that GLP-MS reverses about a quarter of the edges on average.

3.2.5 Discussion

To conclude, GLP is a configurable method for the layering step that, compared to other state-of-the-art methods, shows on average improved performance on compactness. That is, the number of dummy nodes is reduced significantly for most graphs and can never increase. While the number of dummy nodes only allows for an estimation of the area, the effective area of the final drawing, measured in pixels, is reduced as well. Further-



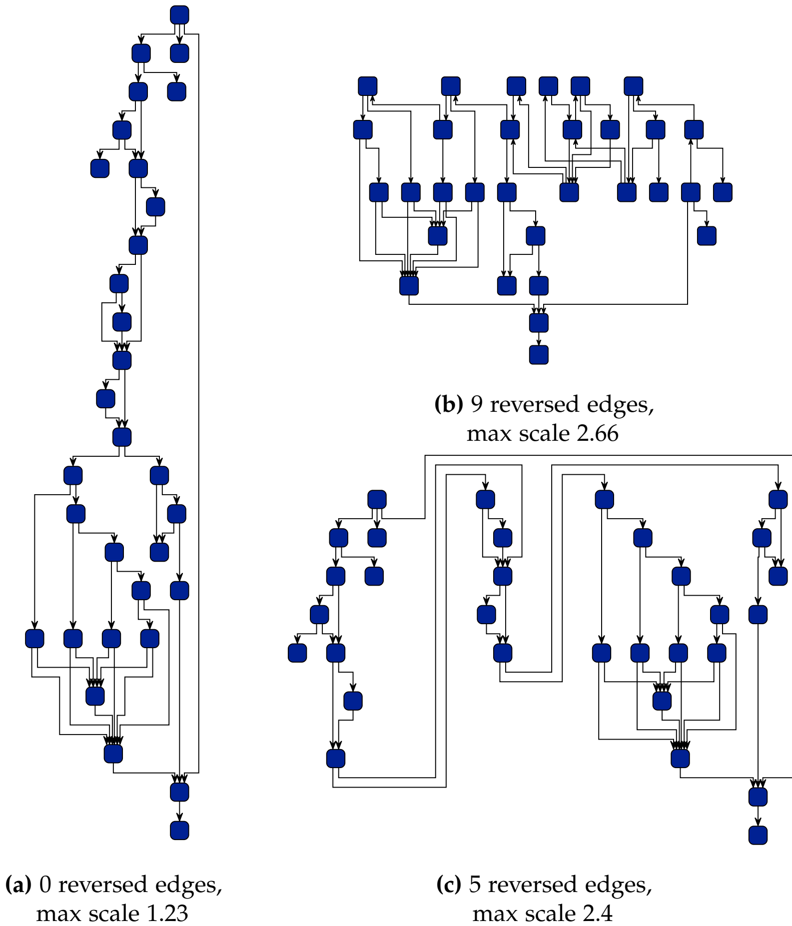
(a) Max scale



(b) Number of reversed edges

Figure 3.11. Results of applying GLP-MS with three different target aspect ratios (x axis) to 112 of the ATTar graphs. The number of reversed edges in (b) is in percent.

more, graph instances for which current methods yield unfavorable aspect ratios can easily be improved. The performance of the presented heuristic is largely comparable to the optimal solutions delivered by the IP approach with the exception of the number of reversed edges. For use cases where the directionality is not that important, the compactness can be improved even further as demonstrated using GLP-MS. Nevertheless, for uses case



(a) 0 reversed edges, max scale 1.23

(b) 9 reversed edges, max scale 2.66

(c) 5 reversed edges, max scale 2.4

Figure 3.12. Different drawings of the g.31.37 graph from the North graphs collection [DGL+97a]. Note that the layout direction of this example is top-to-bottom. (a) is drawn with EaGa, (b) with 2-3-GLP. (c) is created by bluntly cutting the layering of (a) at two points and placing the chunks next to each other. The stated max scale values are for $\mathcal{R} = (1920, 1080)$, an up-to-date resolution of a wide screen. Of the three drawings (b), is the most compact one, however, it is not evident that it is the same graph as in (a).

where directionally should in fact be preserved, a different approach may lend itself better. A higher number of reversed edges seems to be inevitable, however, this may be acceptable if the reversed edges are clearly distinguishable from the regular edges. This would, for instance, be the case if the reversed edges are spatially close within the drawing. Consider Figure 3.12 as a further illustration. While the traditional drawing (a) is unsuited for a wide screen, drawing (c) better matches such a screen's aspect ratio, and the well-defined sections where edges point upwards allow to follow the overall flow of the graph easily. This is not true for drawing (b), on the other hand, which nevertheless is the most compact one. The approach pursued by (c) is investigated in further detail in the next section.

3.3 Wrapping

This section follows up on the idea of the previous section to cut a given layering into multiple *chunks*, instead of altering the definition of a layering as GLP does. From now on this procedure is referred to as *wrapping*. As seen before, one reason for unfortunately large aspect ratios is the fact that all edges have to point into the same direction, another reason is that certain graph types occurring in practice encourage drawings with unfortunate aspect ratios by nature. An example can be seen in Figure 3.13a. It shows a sequentialized SCG (cf. Section 1.4.2), where each node represents an execution step of a program. The larger the program represented by the SCG, the longer the resulting drawing will become. As a consequence, at some point either only a small part of the drawing can be displayed on a computer screen or the graph's drawing has to be scaled down to illegibility. A solution to this problem is the proposed wrapping: split the graph's layering into multiple chunks and draw the chunks side by side, as demonstrated in Figure 3.13b. While the edges that connect one chunk with another chunk point backwards, the chunks are well-separated and the overall flow of the diagram remains clearly visible. The points where the layering is split are referred to as *cuts*:

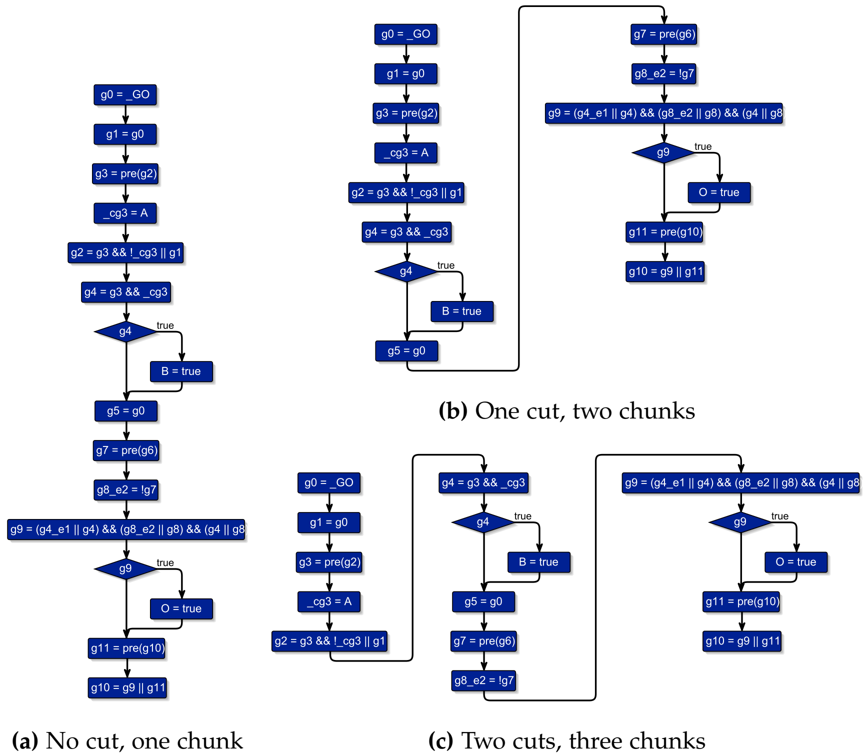
Definition 3.1 (Cut index)**.** Given a graph $G = (V, E)$ and a corresponding layering $L = (L_1, \ldots, L_m)$, a *cut index* c is an index out of $\{2, \ldots, m\}$ where $m \geqslant 2$ is assumed, otherwise splitting a layering is not required. A cut index c may be *valid*. The exact definition of valid depends on the use case and will be defined more precisely later on. If c is not valid, it is *forbidden*.

An ordered set of monotonically increasing valid cut indices $C = (c_1, \ldots, c_k)$ partitions a layering L into $k + 1$ chunks:

$$S_1, \ldots, S_{k+1} = (L_1, \ldots, L_{c_1-1}), (L_{c_1}, \ldots), \ldots, (\ldots, L_{c_k-1}), (L_{c_k}, \ldots, L_m).$$

Let $m' = \max_{1 \leqslant i \leqslant k+1} |S_i|$. A new layering $L' = (L'_1, \ldots, L'_{m'})$ can be constructed by subsequently adding the nodes of each S_i to the layers of L', each time starting at L'_1.

To compute reasonable cut indices, it is necessary to be aware of a graph's dimensions. The previous section on GLP used estimates on the



(a) No cut, one chunk

(b) One cut, two chunks

(c) Two cuts, three chunks

Figure 3.13. A sequentialized SCG drawn with different numbers of cuts. The aspect ratios of the drawings vary significantly, each well-suited for a certain drawing area.

width and the height of a layer that assumed nodes to have a unit dimension. This time, the method shall be more precise: One can conservatively estimate the width and height of each individual layer L_i of a layering L as follows:

$$w(L_i) = \max_{v \in L_i} (w(v) + s), \qquad h(L_i) = \sum_{v \in L_i} (h(v) + s).$$

The constant s is a prescribed spacing value to be preserved between pairs of nodes. To be more precise, one would have to sum up one s less. The overall width and height of a given layering L can then be estimated as:

$$w(L) = \sum_{L_i \in L} w(L_i), \qquad h(L) = \max_{L_i \in L} h(L_i).$$

While this is more accurate than what was used in the context of GLP, the quality of the estimations still depends on the complexity of the used graph instances. In particular, the definitions neglect two elements that may have a significant impact on the dimensions of a drawing: (a) north/south port dummies that are introduced later, just before the third step of the layer-based approach, and contribute further height to a layer and (b) edge routes that can increase the distance between pairs of layers, increasing the overall width. Furthermore, remember that the goal is to draw the computed chunks one below the other with backward wrapping edges in-between. Generally, the subsequent steps of the layer-based approach will ensure that those edges are drawn straight in a final drawing. As a consequence, the nodes within the new layering's layers are not as close to each other as assumed by the height estimates defined above. To address this, further width and height estimates are defined as follows that consider the computed chunks. The dimensions of the individual chunks $S_1, \ldots, S_{k+1}$ are:

$$w(S_i) = \sum_{L_j \in S_i} w(L_j), \qquad h(S_i) = \max_{L_j \in S_i} h(L_j).$$

The chunk-based estimates of the new layering L' then are:

$$w_S(L') = \max_{1 \leqslant i \leqslant k+1} w(S_i), \qquad h_S(L') = \sum_{1 \leqslant i \leqslant k+1} h(S_i).$$

Note, however, that this time $w_S(L')$ intuitively feels inferior to $w(L')$:

$w_S(L')$ may underestimate the width of the individual layers since the different chunks share the same set of layers. Thus, the combination of $w(L')$ and $h_S(L')$ would be expected to be the most accurate one. For ease of presentation and since an informal experiment with the test graphs used here did not show significant differences in the results, the following explanations use the chunk-based estimates only. Nevertheless, it may be worthwhile to explore the impact of different estimates in greater detail in the future.

With these notations at hand, the following problem can be defined:

Problem Wrapping Layered Graphs (WLGP)

Given a graph $G = (V, E)$, a corresponding layering L, and a prescribed drawing area $\mathcal{R} = (r_w, r_h)$, the problem is to find a set of valid cut indices C such that for the induced layering L'

1. the prescribed drawing area is used as good as possible, and
2. the number of edges that point into the "wrong" direction is kept small.

The first point is assessed based on the max scale value as defined in Section 1.3. Since an optimization problem used below is a minimization problem, the inverse definition of max scale is used:

$$\text{Minimize} \quad \max\left\{\frac{w_S(L')}{r_w}, \frac{h_S(L')}{r_h}\right\}.$$

Further note that the second point of the problem does not relate to the number of cuts but to the number of edges that span a certain cut index.

Now, from a technical point of view, the WLGP can be solved in two different ways. The first solution, discussed in the next section, is easier to implement but only allows a single edge to wrap backwards per cut. The second solution requires more intricate adaptations of the layer-based approach but works for an arbitrary number of wrapped edges and is discussed in Section 3.3.2.

3.3.1 Single-Edge

The first solution of WLGP is restricted to wrapping a single edge per cut, thus referred to as WLGPse in what follows. Valid cuts are therefore in-between pairs of layers that are connected by exactly one edge:

Definition 3.2 (Spanning edge)**.** Given a graph $G = (V, E)$ and a layering $L = (L_1, \ldots, L_m)$, an edge $e = (u, v) \in E$ is *spanning* an index i with $1 < i \leqslant m$ if $u \in L_j, j < i$ and $v \in L_k, k \geqslant i$.

Definition 3.3 (Valid cut index in WLGPse)**.** A cut index c is valid for WLGPse if $|\{e \in E : e \text{ spans } c\}| \leqslant 1$.

The wrapping can then be realized using a single processing element after the crossing minimization step which reconstructs the layering according to any computed cut indices. The order of the nodes within the layers can be derived straightforwardly and has largely been determined by the crossing minimization. Wherever a backward wrapping edge is to be inserted, the processor has to create a chain of long edge dummies.

The remainder of this section is structured as follows. First, an optimization problem and two straightforward and fast heuristics to solve WLGPse are presented. Afterwards, the three methods are evaluated based on a set of graphs from practice. The results of the optimization problem serve as a reference for the results of the heuristics.

Optimization Problem

Note that while the computed cut indices are optimal for the way the widths and heights are estimated, the indices are not necessarily the best ones with respect to a final drawing as the dimensions of a final drawing additionally depend on the subsequent coordinate assignment step and edge routing step.

Input and Parameters Let $G = (V, E)$ be a graph with original layering $L = (L_1, \ldots, L_m)$. Let $w(v)$ and $h(v)$ denote the width and height of a node $v \in V$.
Let M be the maximum number of allowed cut indices. An obvious upper bound is $m - 1$.
$a(c)$ indicate if cutting at index c is allowed ($a(c) = 1$) or forbidden ($a(c) = 0$) and is defined for $2 \leqslant c \leqslant m$.
$a_{\mathcal{R}}$ is the aspect ratio of the desired reference frame $\mathcal{R}$. r_w and r_h are computed such that $a_{\mathcal{R}} = \frac{r_w}{r_h}$ and can reasonably be approximated using *continued fractions*, for instance.

Decision Variables $cuts(i)$, $1 \leqslant i \leqslant M$, represent the cuts that should be applied in ascending order and take values in $\{2, \ldots, m\}$. The cuts split the original layering into $M + 1$ ranges:

$$r_1, \ldots, r_{M+1} = \left(L_1, \ldots, L_{cuts(1)-1}\right), \ldots, \left(L_{cuts(M)}, L_m\right).$$

Note that it is allowed for a cut index to occur more than once. It yields an empty range which is simply discarded when constructing the new layering.
$max_r(j)$, $1 \leqslant j \leqslant M + 1$, holds the maximum height of any layer in the range r_j.
Likewise, $sum_r(j)$ holds the sum of layer widths in the range r_j.
d_w and d_h represent the width and height if the assigned cuts were applied and thus realize the $w_S(L')$ and $h_S(L')$: d_w is the maximum of all sum_r variables and d_h is the sum of all max_r variables. Further, let $b2i$ denote a function mapping false to 0 and true to 1.

Objective

$$\text{Minimize} \qquad \max\left\{\frac{d_w}{r_w}, \frac{d_h}{r_h}\right\}$$

Constraints

$$cuts(i) \leqslant cuts(i+1) \qquad \text{for all } 1 \leqslant i < M \qquad \text{(A)}$$

$$
\begin{aligned}
b2i(a(i)=0) &\leqslant b2i(cuts(j) \neq i) && \text{for all } 2 \leqslant i \leqslant m, 1 \leqslant j \leqslant M && \text{(B)}\\
max_r(1) &= \max_{1 \leqslant i < cuts(1)} h(L_i) && && \text{(C)}\\
max_r(j) &= \max_{cuts(j-1) \leqslant i < cuts(j)} h(L_i) && \text{for all } 2 \leqslant j \leqslant M && \text{(D)}\\
max_r(M+1) &= \max_{cuts(M) \leqslant i \leqslant m} h(L_i) && && \text{(E)}\\
sum_r(1) &= \sum_{1 \leqslant i < cuts(1)} w(L_i) && && \text{(F)}\\
sum_r(j) &= \sum_{cuts(j-1) \leqslant i < cuts(j)} w(L_i) && \text{for all } 2 \leqslant j \leqslant M && \text{(G)}\\
sum_r(M+1) &= \sum_{cuts(M) \leqslant i \leqslant m} w(L_i) && && \text{(H)}\\
d_w &= \max_{1 \leqslant i \leqslant M+1} sum_r(i) && && \text{(I)}\\
d_h &= \sum_{1 \leqslant i \leqslant M+1} max_r(i) && && \text{(J)}
\end{aligned}
$$

Constraints (A) assert that the assigned cuts are sorted ascendingly. (B) prevent forbidden cuts. (C)–(E) bind the variables that hold the maximum height of a range. Likewise, (F)–(H) bind the sum of the width variables. Finally, (I) and (J) bind the variables that estimate the overall width and height of the layering induced by the assigned cut indices.

Heuristics

Next, two fast and simple heuristics to determine cut indices are presented. The first one tries to come close to a desired aspect ratio of the final drawing, hence it is referred to as aspect ratio-driven (ARD). The second one tries, at the cost of further work, to maximize the max scale value, hence it is referred to as max scale-driven (MSD).

Aspect Ratio-Driven (ARD) Let a graph G, a layering L, and a prescribed drawing area $\mathcal{R} = (r_w, r_h)$ be given. The desired aspect ratio of the new

layering L' is $a = {}^{r_w}/_{r_h}$, and the heuristic should compute chunks in a way such that $a \approx {}^{w_S(L')}/_{h_S(L')}$. The number of chunks z the old layering L has to be split into in order to achieve a can be estimated as follows, where the heuristic assumes that $w_S(L') \approx {}^{w(L)}/_z$ and $h_S(L') \approx z \cdot h(L)$.

$$z = round\left(\sqrt{\frac{w(L)}{a \cdot h(L)}}\right).$$

A corresponding set of potential cut indices is

$$C = \left\{\left\lceil \frac{|L|}{z+1} \cdot i \right\rceil + 1,\ \ 1 \leqslant i \leqslant z\right\}.$$

z can be bound from above since there cannot be more chunks than layers: $z = \min\{z, |L| - 1\}$. Since ${}^{|L|}/_{z+1} \geqslant 1$, no two elements of C are equal.

Max Scale-Driven (MSD) As explained thoroughly in Section 1.2, aiming at a specific aspect ratio does not necessarily result in a drawing that uses a given drawing area to its full potential. The aforementioned ARD heuristic simply splits the graph such that the resulting layering roughly matches the aspect ratio of the given drawing area. The MSD heuristic takes a different approach by directly addressing the max scale measure.

To better understand its idea, remember the definition of max scale: $\min\{{}^{r_w}/_w, {}^{r_h}/_h\}$, where r_w and r_h are the width and height of a prescribed drawing area, and w and h are the width and height of a produced drawing. Clearly, to get a large max scale value, small values of w and h are advantageous. Following this, MSD's goal is to choose z cut indices in a manner such that the prescribed drawing area's aspect ratio is matched roughly and such that either w or h is minimized.

Let a graph G and layering L be given. The tentative width $w_S(L')$ of the layering L' to be constructed is the maximum of sums (each sum represents the width of a single chunk). Given z, a set of cut indices that minimizes this maximum can be calculated efficiently. A list cw of length $m = |L|$ stores at each index i the cumulative width of all layers at smaller indices than i. Thus, $cw(m)$ holds the overall width. Every chunk should consist of layers whose widths roughly sum up to ${}^{cw(m)}/_{z+1}$. The algorithm then

finds the cut indices by iterating cw from start to end and collecting each index at which the desired sum is exceeded. Since MSD must know z to do so, it uses the z calculated by ARD as an initial guess but is able to freely increment or decrement it since its measure to evaluate the set of computed cut indices is independent of z. During the upcoming evaluations this is denoted by MSD-k, where k indicates the amount of freedom. For instance, $k = 1$ and $z = 3$ would allow MSD to try layerings with two, three, and four chunks.

The tentative height of L' has not been considered so far: improvidently minimizing the sum of maximums (where the maximums are the heights of the individual layers) is very likely to result in unbalanced cut indices; a "good" set of cut indices, with respect to the minimal height, would put the narrowest nodes in chunks by their own and put as many nodes as possible in the same rows as the tallest nodes. Nevertheless, it should be investigated further in the future.

Validification Remember that it is not allowed to cut at every index. As a consequence, the forbidden indices in C have to be altered. In what follows, this process is referred to as *validification*.

Two interchangeable validification strategies are evaluated here. The first strategy is outlined in Algorithm 3.3 and increments every forbidden cut index until it is valid. To preserve the original distances between consecutive cut indices, the remaining cut indices are increased accordingly. This can result in indices larger than the number of original layers. Such indices are simply discarded. The method is referred to as GREEDY. The second strategy proceeds in a more sophisticated fashion. It is outlined in Algorithm 3.4 and seeks for each forbidden index the closest valid index position. For this, it iterates all possible index positions from 2 to $|L|$ and keeps track of the last valid index. When a forbidden cut index is encountered, the iteration is continued until either a valid (larger) index is found or the distance to the next valid index is larger than the distance to the last valid index. If both distances are the same, the smaller index is picked. The intention behind this method is to distribute the cut indices more evenly and to avoid having to discard indices. It is referred to as LOOKBACK.

Algorithm 3.3: Greedy Validification

Input: Graph G with layering L and cut indices C sorted ascendingly
Output: Valid cut indices $\hat{C}$

```
Ĉ ← empty list
offset ← 0
for c ∈ C do
    c ← c + offset
    while c ⩽ |L| and c not valid do
        Increment c and offset
    if c > |L| then Stop
    Add c to Ĉ
return Ĉ
```

Algorithm 3.4: Lookback Validification

Input: Graph G with layering L and ascendingly sorted list $(c_1, \dots, c_n)$ of desired cuts
Output: Valid cut indices $\hat{C}$

```
Let I = (i_1 = −∞, i_2, ..., i_{m−1}, i_m = +∞) be an ascendingly sorted list,
  where the i_ℓ, 1 < ℓ < m, are all possible indices that would be valid cuts
Ĉ ← empty list
k ← 1, ℓ ← 1
offset ← 0
while k ⩽ n and ℓ < m do
    d ← c_k + offset
    while i_{ℓ+1} < d do Increment ℓ
    if d − i_ℓ < i_{ℓ+1} − d then s ← 0 else s ← 1
    Add i_{ℓ+s} to Ĉ
    offset ← offset + i_{ℓ+s} − d
    do Increment k
     while k ⩽ n and c_k + offset ⩽ i_{ℓ+s}
    ℓ ← ℓ + 1 + s
return Ĉ
```

Evaluation

The two presented heuristics, ARD and MSD, were evaluated using 135 sequentialized SCGs that originate from unit tests (cf. Section 1.9) with 10 to 227 nodes, 41.1 on average, and 11 to 253 edges, 45.5 on average. SCGs are drawn top-to-bottom, thus wrapping is expected to be advantageous for larger aspect ratios, and all of the graphs permit valid cut indices. For every graph the two heuristics and the optimization problem were used to compute five different drawings with a particular drawing area in mind. For the optimization problem, denoted by OPT, the number of allowed cuts M was selected such that for every graph at least one cut remained unused. A timeout was set to two hours but never reached. Four questions were to be answered:

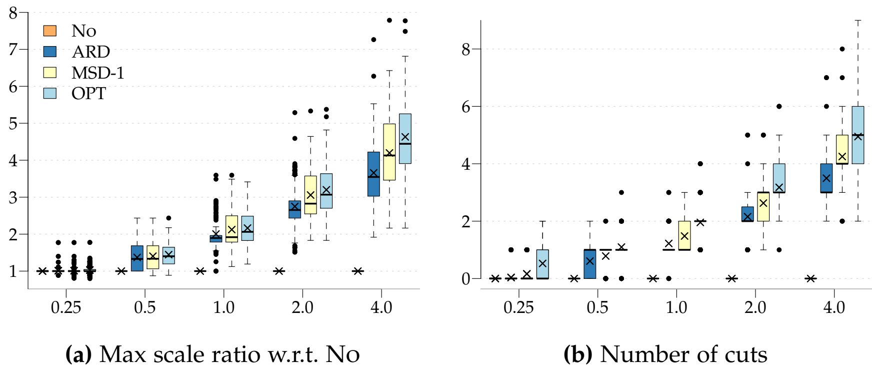
Q1 How do the max scale values of drawings created with ARD, MSD, and OPT compare to drawings where no wrapping is applied?

Q2 Which validification method performs better?

Q3 How close are the estimations of width and height to the width and height of the final drawing?

Q4 What is a reasonable value of k for MSD-k?

Regarding the general quality of the heuristics' results (Q1), consider Figure 3.14, which shows max scale ratios relative to no wrapping. It can be seen that both heuristics are advantageous for target aspect ratios from 1.0 to 4.0, with MSD-1 producing drawings that can be scaled more than four times larger on average for 4.0 than when no wrapping is applied. Also, starting with a target aspect ratio of 1.0, neither heuristic results in worse max scale values than no wrapping. For 0.25 the heuristics compute a single cut for 5 (ARD) and 22 (MSD) graphs without any significant improvement in max scale, indicating that the traditional layerings already fit the drawing area best; something one would expect. For the tested graphs, MSD-1 produces on average slightly larger max scale values than ARD, which stem from an on average larger number of cuts. Looking at OPT's results it can be seen that there is some potential for improvement for target



(a) Max scale ratio w.r.t. No

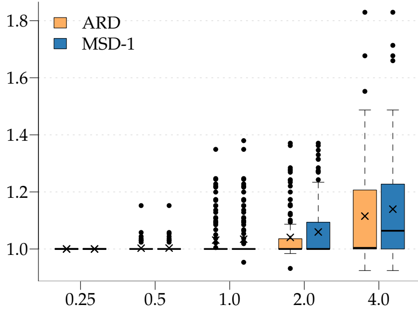
(b) Number of cuts

Figure 3.14. Comparing results for final drawings of SCGs either created with a wrapping heuristic, ARD or MSD-1, the optimization problem OPT, or without wrapping (No). (a) and (b) show different measures (y axis) plotted against varying target aspect ratios (x axis). Where wrapping is required (for larger aspect ratios due to the top-to-bottom layout direction), the heuristics significantly improve the max scale values.

aspect ratios of 2.0 and 4.0, where OPT uses a larger number of cuts but yields better max scale values than the heuristics on average. Remember that OPT does not necessarily give the best result for final drawing areas since it is only optimal with respect to the computed estimates and the set bound on M, something that can be seen for a target aspect ratio of 1.0, for instance, where ARD and MSD yield larger max scale values for a couple of graph instances.

Regarding Q2, Figure 3.15 indicates that the Lookback method is superior. Applied after both ARD and MSD, it consistently produces equal or better max scale values on average for the final drawings, particularly for larger aspect ratios. Following this observation, the remaining plots are produced with the Lookback method.

When using max scale as quality measure, it is not important that the heuristics work with estimates of width and height that are as close as possible to the final pixels of a drawing but that the relation between the estimates resembles their relation within the final drawing. Thus, to address



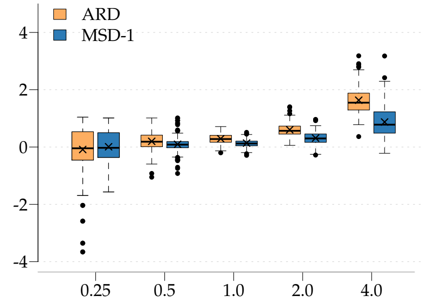


Figure 3.15. Max scale ratios between the two presented validification methods, Greedy and Lookback, measured on final drawings for both heuristics (y axis). Values larger than zero indicate that Lookback performs better.

Figure 3.16. Differences between estimated and final normalized aspect ratios (y axis) for ARD and MSD-1, i. e. what aspect ratio a method thinks it creates and what it actually creates. A value of zero means an exact match.

Q3, Figure 3.16 shows normalized aspect ratio differences between the estimated dimensions of a drawing and the dimensions of a final drawing for both heuristics. The tested SCGs usually result in a layering with no more than one original node per layer and thus the width and height estimations are expected to be rather accurate. The figure supports this in that the estimates are quite close to the final results, with MSD-1 being more accurate than ARD. For larger aspect ratios, both heuristics start to overestimate the final drawings' aspect ratios: the estimated height is smaller than and the estimated width is larger than the respective final values. It remains to be explored how much the quality of the estimates impacts the results and whether better estimations can actually produce better results. After all, it is likely that better estimations require further knowledge about the remaining layer-based steps to be executed after wrapping, the coordinate assignment step and the edge routing step, and would thus significantly increase the complexity of an implementation.

Regarding MSD's k (Q4), consider Figure 3.17. It can be seen that for target aspect ratios larger than or equal to 0.5, setting k to 1 instead of 0 produces slightly better max scale values, while further increasing k does

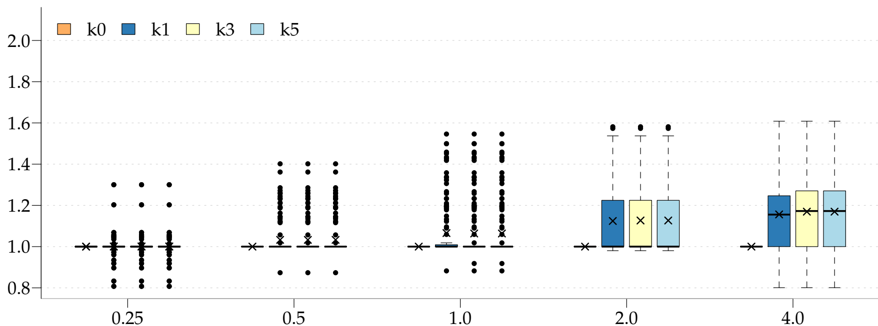


Figure 3.17. Max scale ratio values (y axis) for different ks of MSD w.r.t. $k = 0$.

not significantly improve the results. It can further be observed that an increased k yields a worse max scale value for certain graph instances. This is explained by the fact that a validification method is executed after the heuristic, i. e. MSD-1 finds better cut indices than MSD-0 but the validified indices of MSD-0 allow for final drawings with a larger max scale value.

The next section takes a look at the unconstrained WLGP, which in turn is allowed to cut multiple edges per index.

3.3.2 Multi-Edge

The wrapping presented in the previous section addresses a very restricted scenario in which only one backward wrapping edge per cut is allowed. This restriction allows good estimations on the width and height of the final drawing and allows an easy implementation. However, the desire to leverage the space of a given presentation medium to its full potential remains as well for graphs for which it is not always possible to wrap only a single edge per cut. Flexibly altering the aspect ratio of final drawings for general graphs has already been discussed in Section 3.2 as the Generalized Layering Problem (GLP). There, a layering is computed from scratch, the input graph may be cyclic, and arbitrary edges may be reversed. Consequently, a layering computed by GLP can look quite different compared to a layering computed with a traditional layering method that requires an acyclic graph as input.

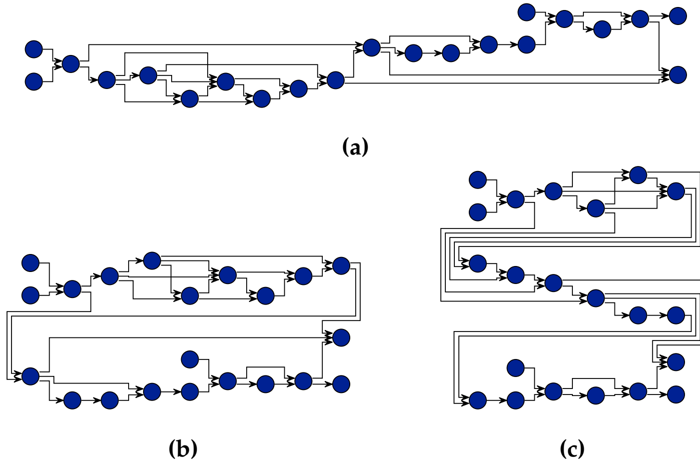


Figure 3.18. Three drawings of the same graph, each time with a different number of cut edges.

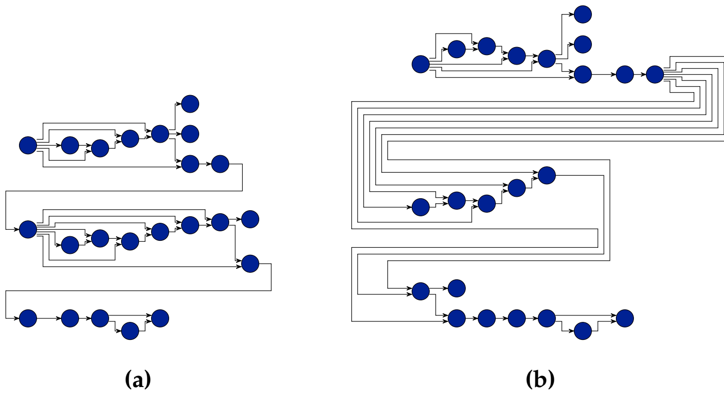


Figure 3.19. Selecting unfortunate cut indices can result in poor and cluttered drawings. In (b) the two cut indices of (a) were increased and decreased by one, respectively.

Algorithm 3.5: Edge-Aware Cut Improvement

Input: Graph G with layering L, cut indices C sorted ascendingly, number of spanning edges *spans*, cost function *cost*, and a distance function δ
Output: Improved cut indices $\hat{C}$

$\hat{C} \leftarrow$ empty list
do $|C|$ times
  Calculate all costs
  Find $idx \in \{2, \ldots, |L|\}$, s.t. $cost(idx)$ is minimal
  Add idx to $\hat{C}$
  Remove c from C for which $\delta(idx)$ is minimal
return $\hat{C}$

The idea behind the unconstrained WLGP is therefore to better preserve the recognizability and to better preserve the overall directionality when looking at a layering before and after modification. For this, an initial layering is created with a traditional layering method and is cut into chunks afterwards. Figure 3.18 illustrates the idea to preserve the recognizability of a graph. It is discernible that (a), (b), and (c) are different drawings of the same graph. As opposed to this, the drawings shown in Figure 3.7 (p. 77), produced with GLP, do not immediately look alike.

When allowing to cut multiple edges, care has to be taken: The space between a pair of layers can be populated by many edges, especially for denser graphs. Cutting many edges results in a large number of backward wrapping edges, which in turn results in poor and cluttered final drawings. See Figure 3.19 for an example. Good cuts are therefore between pairs of layers that have a small number of edges in-between. The next section introduces a method that uses initial cut indices determined by ARD or MSD and afterwards applies an improvement strategy: slightly alter the indices in order to reduce the total number of cut index spanning edges.

Edge-Aware Cut Improvement

Algorithm 3.5 presents a greedy method (ECI) to improve a given set of cuts C with respect to the number of cut index spanning edges. Let I denote

the set of possible indices, i.e. $I = \{2, \dots, |L|\}$, and let $spans(i)$ denote the number of edges spanning index $i \in I$. The method uses a cost function *cost* to rate the quality of an index as a cut. Here, the following function is used:

$$cost(i) = spans(i) + \delta(i)^e,$$

where

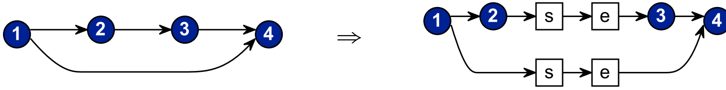
$$\delta(i) = \min_{c \in C} \; |i - c|$$

is the minimal distance of an index i to any of the cut indices of C. The preservation of the existing cuts can be weighted larger by increasing e, where the fact that e is an exponent already strongly limits their freedom. To give an example, for $e = 2$ an index with a distance of two is deemed better than a given cut index that would save more than four edge reversals. Using the cost function, the method calculates the cost of each index and greedily picks the index with lowest cost. Afterwards, the cut index in C that is closest to the picked index is removed, thus resulting in different cost values for the next index to pick. The algorithm runs in $O(|C| \cdot |L|)$ time.

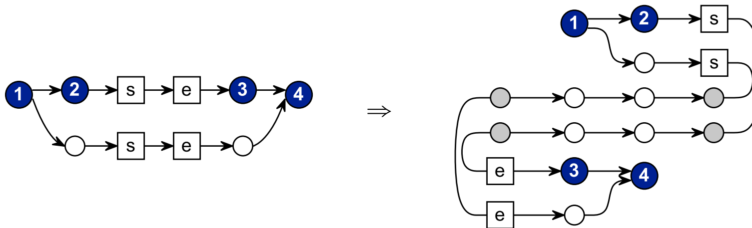
Technical Integration

This section gives technical details on how the multi-edge wrapping procedure can be integrated into the layer-based approach without changing an existing implementation. It consists of three ameliorations, which are illustrated in Figure 3.20.

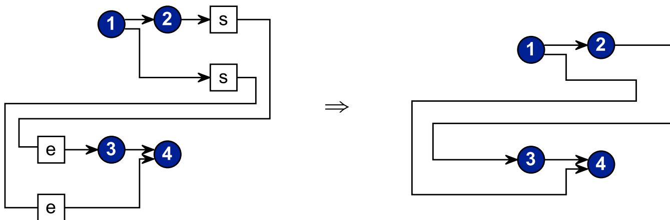
First, immediately after the layering step, before the layering is turned into a proper layering, dummy nodes are inserted wherever a cut is planned. More precisely, for a planned cut between two layers i and j, two new layers are inserted and every edge that either connects or spans i and j is split by adding two *breaking point* dummies in the new layers. The dummies are called `s` and `e` for start and end, and while one can surely come up with an implementation that manages without them, they are helpful as clear markers in both documentation and code. This initial step is illustrated in Figure 3.20a. Afterwards, the existing implementation takes care of making the layering proper by adding dummy nodes for long edges, and it executes the crossing minimization step. There, the breaking point dummies can be understood as long edge dummies and therefore do not actively increase



(a) Dummy nodes for breaking points (s and e) are introduced after the layering step based on calculated cut indices.



(b) After crossing minimization, the breaking point dummies are used to alter the layering. In a left-to-right layout, the chunks defined by the cut indices are placed as rows one below the other. Chains of dummy nodes are inserted for the edges between s and e. Note that the gray circled dummy nodes are connected by *in-layer* edges.



(c) After the edge routing step and after merging the dummy nodes of long edges, the s and e dummy nodes remain. They are removed and the three involved edges are simply joined to form the final edge path.

Figure 3.20. Illustrating the wrapping procedure as part of the layer-based approach. (a), (b), and (c) are three new intermediate steps to be executed at different points during the overall execution of the layer-based approach.

the number of edge crossings.

Second, the new layering can be formed as depicted in Figure 3.20b. The e dummy nodes of every pair of breaking point dummies in a certain layer can be moved to the very first layer of the new layering (in given order). Any successive nodes are moved to the next layer and so forth. The edge that connects the two dummies now runs from the very first layer to one of the later layers of the new layering. Since such a layering would not be proper, dummy nodes must be introduced in the same way as they are introduced for long edges after the layering step. Note that if the chains of long edge dummies are introduced in the reversed in-layer order than the breaking point dummies, no new crossings have to be introduced. Further note that the edges that connect the gray circled dummy nodes to s and e are normally prohibited as they connect nodes that are located in the same layer. ELK Layered already comprises methods that handle such cases (cf. Section 2.6), alternatively, the gray dummy nodes could be dropped and s and e could be connected directly to the adjacent white dummy nodes. This would then require additional edge routing code that makes sure the corresponding edges are properly routed around the s and e nodes. Third, after edge routing and after removing long edge dummies, the scenario depicted in the left drawing of Figure 3.20c is given. The paths of the three edges incident to the s and e nodes can simply be joined to form the final path of the backward edge. At this point certain edges may take superfluous detours, e. g. the edge connecting node 1 and node 4 in the right drawing of Figure 3.20c. While the detour below node 2 is necessary for certain graph instances to prevent edge crossings, it introduces two unnecessary edge bends and increases the edge's overall length in this particular case. Chapter 5 discusses a technique called one-dimensional compaction and shows (p. 188) how it can be used to improve the final edge paths to obtain results as seen in the right drawing of Figure 3.21.

Edge Path Improvements

Besides the small edge detour just mentioned, there are two other types of unfortunate edge paths that stem from the very rigid procedure discussed above:

1. An edge can span multiple cut indices resulting in a back and forth edge path as depicted on the left side of Figure 3.22.

2. A cut edge is first routed to the last layer, then back to the very first layer, and finally to the target node. This can result in unnecessarily long edge paths as depicted on the left side of Figure 3.23.

The next two paragraphs discuss how to tackle these two problems. Example drawings with improved edge paths have already been seen in Figure 3.18.

Edges Spanning Multiple Cuts The first problem stems from the fact that splitting a single edge (u, v) at multiple cut indices results in subsequent pairs of breaking point dummies s and e, which are then connected by long edge dummies. Let d^* denote a chain of zero or more dummy nodes. The sequence of nodes connecting u and v can be written as (following the path between u and v irrespective of the edge directions):

$$u, d_1^*, \mathsf{s}_1, d_2^*, \mathsf{e}_1, d_3^*, \mathsf{s}_2, \ldots, d_{m-2}^*, \mathsf{s}_n, d_{m-1}^*, \mathsf{e}_n, d_m^*, v.$$

As illustrated in Figure 3.22, this sequence can be reduced by re-connecting e_n to the last dummy of d_2^* and dropping all dummy nodes in-between:

$$u, d_1^*, \mathsf{s}_1, d_2^*, \mathsf{e}_n, d_m^*, v.$$

As a consequence, every cut edge is routed back to the initial layer immediately after it is cut. The edge then skips a number of rows vertically, before it is routed to its target node. While this significantly reduces the length of the edges and avoids back and forth edge routes, which feel very unnatural to a human and are tedious to follow, it can happen that new edge crossings are introduced that are otherwise avoided by the intertwining edge routes.

Overly Long Backward Edges The second problem occurs simply due to the methodical procedure of wrapping back the edges by splitting between the two breaking point dummies and inserting a chain of long edge dummies. If the source and target node are not located at the beginning and end of a chunk, this automatically produces overly long edge paths. Figure 3.23 shows an example. Both the s node and the e node are next to a long edge

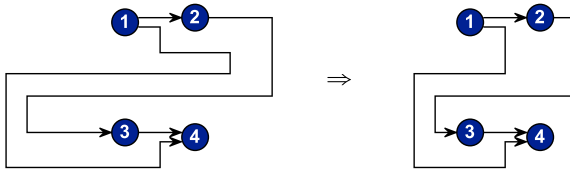


Figure 3.21. Poor edge routes resulting from the very methodical wrapping procedure can be improved using one-dimensional compaction as discussed in Chapter 5.

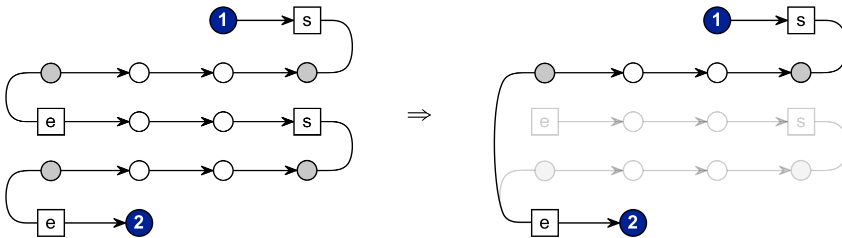


Figure 3.22. Edges spanning multiple cut indices result in back and forth edge paths. In the example, one pair of s and e is superfluous and can be removed.

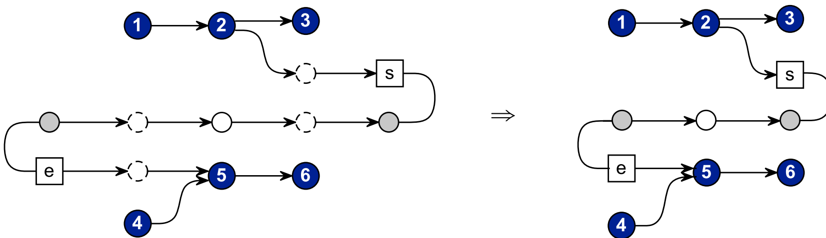


Figure 3.23. The overall edge length can be reduced by replacing superfluous dummy nodes.

dummy (dashed outline) on their left and right side, respectively. Consider only the s node for the moment. If the pair p of the s node and the gray in-layer dummy below it are adjacent within their common layer, and if the same is true for the two neighboring dashed long edge dummies, the two long edge dummies can simply be removed. They are then replaced by p. If one traverses the nodes within the layers from bottom to top, any "nested" s nodes are processed first, making room for other s nodes that were blocked beforehand. The e node in Figure 3.23 can be treated analogously with the exception that this time the layer is traversed in its regular order.

Evaluation

The multi-edge wrapping is evaluated using the ATTar graphs, which have a large aspect ratio when drawn from left to right (cf. Section 1.9).[5] Note that the evaluation of the single-edge wrapping, which was based on SCGs, used a top-to-bottom layout direction and was expected to improve drawings for which traditional layerings yield tall and narrow drawings. For the ATTar graphs, on the other hand, the goal primarily is to improve on wide and flat drawings that are to be fit into drawing areas with a small aspect ratio.

Different configurations of the layout algorithm were executed and are denoted as follows: no wrapping is performed (No), wrapping is performed but neither the cuts are improved nor the edge length is optimized (Wr), wrapping is performed and cuts are improved (Wri), and finally, wrapping is performed and all improvements are executed (Wrii). For all configurations that include wrapping, the score function's exponent is set to 2.0 and MSD-1 is used to obtain initial cuts. During the evaluation of the single-edge wrapping, measures such as backward edges, edge length, and edge crossings are not of major interest since only single edges are routed backward and no new edge crossings can be introduced. This is different this time. As discussed before, cutting many edges results in very cluttered drawings with an increased edge length. Also, optimizing long chains of breaking point dummy nodes can result in additional edge crossings that can make it more difficult to follow specific edge paths. Therefore, in addition to the

[5] g.29.15 was excluded from the results presented in this section since it is significantly denser than the other graphs and worsens the presentation.

max scale measure, the following plots include results on the number of cut edges, the average length per edge, and the average number of crossings per edge. The following questions were to be answered:

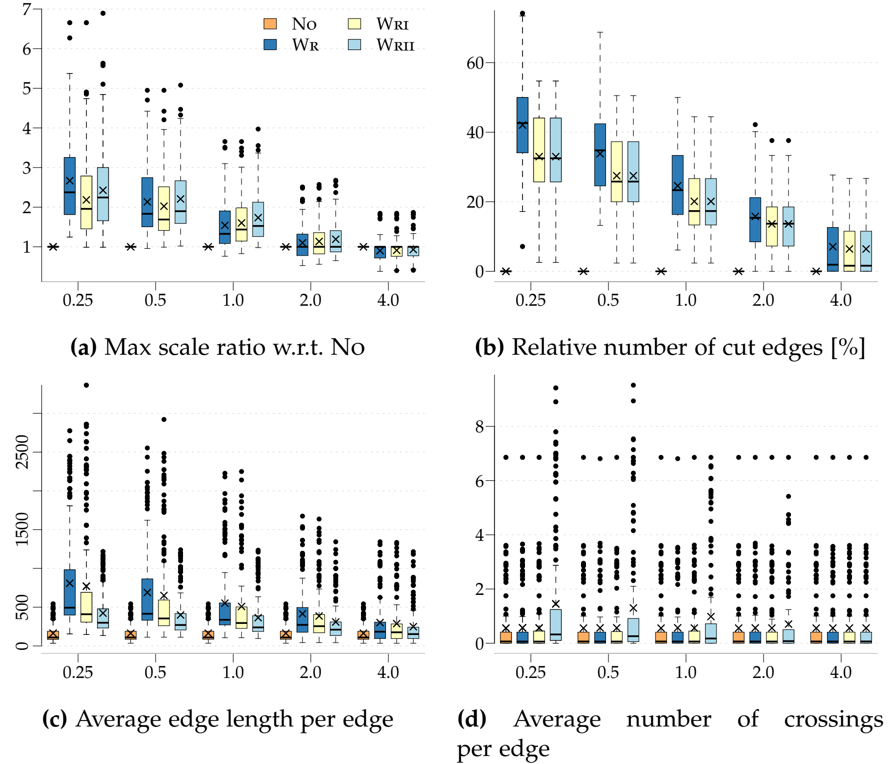
Q1 Does the wrapping improve the max scale measure?

Q2 What is the effect of ECI on max scale and on the number of cut edges? Is the overall edge length reduced by the proposed optimizations?

Q3 The computed cut indices of which of the two heuristic, ARD and MSD, are a better basis for subsequent ECI? Does the score function behave as desired?

Q4 How close are the width and height estimates to the width and height of the final drawing?

Figure 3.24 shows results. When aiming at drawing areas with aspect ratios below or equal to 1.0, executing either of the three wrapping configurations improves the max scale values compared to No (Q1). Drawings created with Wrii can on average be scaled more than twice as large as No for a target aspect ratio of 0.25. For aspect ratios larger than 1.0 no significant improvement can be observed. In fact, for 4.0 No yields the largest max scale values on average; for 72 graphs no wrapping was performed, and for 51 graphs wrapping was performed but resulted in worse max scale values than No. This indicates that one has to be careful when applying the wrapping procedure in ranges where a traditional layering is already close to the prescribed drawing area. This is not surprising since the backward wrapping edges occupy additional space. Moreover, wrapping yields an increased number of cut edges and an increased average edge length, which correlates with the target aspect ratio: the more cuts are required to reach the prescribed drawing area, the more edges must be cut, naturally increasing the total edge length. Figures 3.24b and 3.24c show that the number of cut edges can be decreased using ECI and that the edge length can be improved using the two proposed edge path optimizations (Q2). The increased number of crossings for Wrii, as seen in Figure 3.24d, is expected, and the slight variations visible in the outliers of the other



(a) Max scale ratio w.r.t. No

(b) Relative number of cut edges [%]

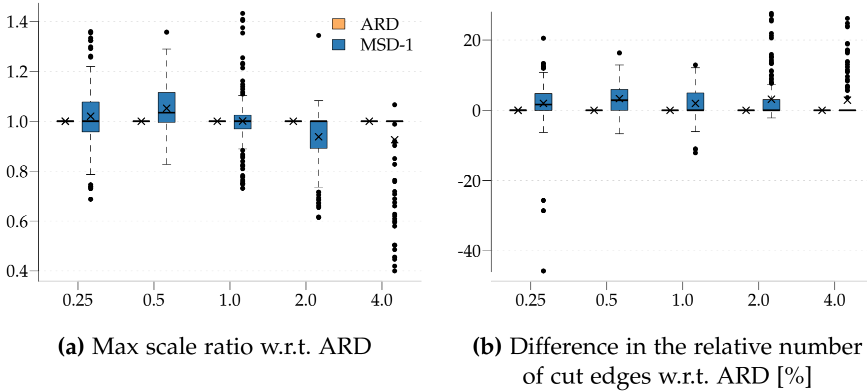
(c) Average edge length per edge

(d) Average number of crossings per edge

Figure 3.24. Results of wrapping ATTar graphs with varying improvements. Target aspect ratios (x axis) are plotted against the measures stated in the captions (y axis).

three methods can be explained by the heuristic nature of the employed layer-sweep strategy to minimize crossings (cf. Section 2.3).

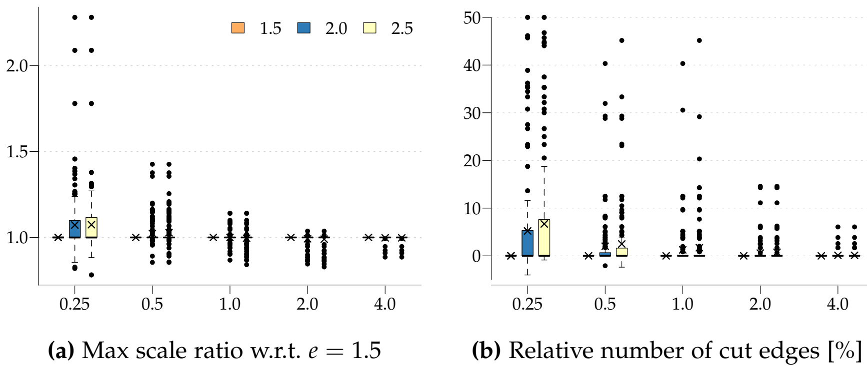
Figure 3.25 shows results of executing WRI with either ARD or MSD-1 to obtain the initial cuts (Q3). No clear winner can be identified. MSD performs on average slightly better for target aspect ratios of 0.25 and 0.5 but yields worse results for target aspect ratios that are closer to a layering's "natural" aspect ratio (2.0 and 4.0). In terms of cut edges, MSD cuts more edges for all target aspect ratios on average.



(a) Max scale ratio w.r.t. ARD

(b) Difference in the relative number of cut edges w.r.t. ARD [%]

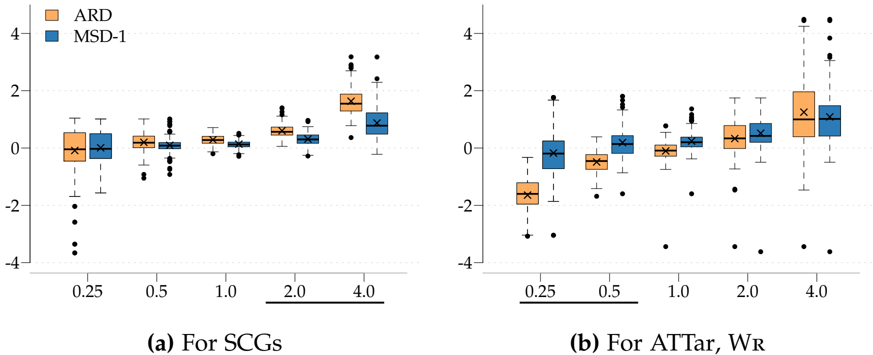
Figure 3.25. Comparing the quality of ARD and MSD-1 as basis for subsequent ECI (without edge length improvement). The measures stated in the captions (y axis) are plotted for various target aspect ratios (x axis).



(a) Max scale ratio w.r.t. $e = 1.5$

(b) Relative number of cut edges [%]

Figure 3.26. Influence of the score function's exponent e on the max scale measure and on the number of cut edges (y axes).



(a) For SCGs

(b) For ATTar, WR

Figure 3.27. Differences between estimated and final normalized aspect ratios (y axis) for ARD and MSD-1. The plot in (a) has already been shown during the evaluation of single-edge wrapping (Figure 3.16). To improve comprehensibility, the aspect ratios for which the wrapping procedure is expected to create a larger number of chunks are underlined. They are different in (a) and (b) as different layout directions are used.

Regarding the behavior of the score function, Figure 3.26 illustrates the impact of the function's exponent e. As one would expect, a lower value of e results in less cut edges but can result in worse max scale values. Only for a target aspect ratio of 0.25 a significant improvement in max scale can be observed when setting e to 2.0 or 2.5 instead of 1.5, however, at the cost of significantly increasing the number of cut edges.

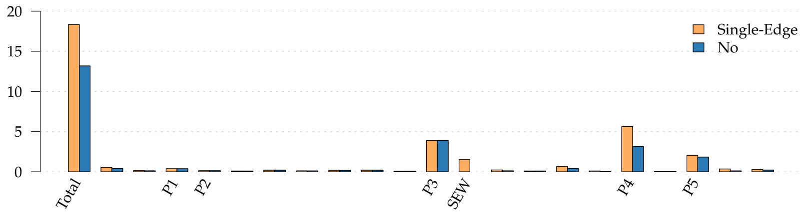
Regarding width and height estimations (Q4), Figure 3.27 indicates that, similar to single-edge wrapping, MSD-1 produces more accurate estimates. This time, however, the heuristics start to underestimate the aspect ratios of the final drawing for the relevant target aspect ratios of 0.25 and 0.5. This can presumably be explained by the fact that compared to the SCGs the ATTar graphs contain more edges that have to be routed between pairs of layers and thus contribute additional width. This is also the reason for the outliers in the bottom right of (b). The corresponding graph (g.46.3) has 46 nodes and 168 edges and requires, due to its edge count, significantly more width between adjacent layers than the other ATTar graphs.

3.3.3 Discussion

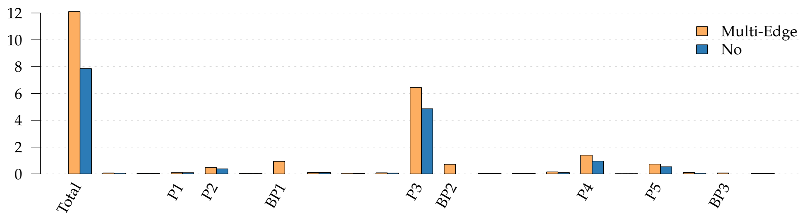
The previous sections presented a methodical procedure that cuts a traditional layering into multiple chunks and places them side by side with the goal to maximize the scale factor with which a drawing can be displayed in a prescribed drawing area. It works best with sparse and "elongated" graphs and graphs that regularly have "bottlenecks". The method can easily be explained to and understood by a user. Apart from the cut points a drawing looks similar to a traditional drawing without any cuts applied. Furthermore, the inherent flow of directed graphs is well-preserved by the wrapping process; the "later" the node is situated within the flow, the "later" the chunk the node ends up in. All of the cut edges that are now pointing backwards are routed in well-defined areas of the drawing, in-between pairs of chunks, and can therefore easily be identified and followed by a user.

Concerning execution time, which has not been discussed so far, the wrapping process adds a significant number of dummy nodes (`s` and `e`, and long edge dummies) to the graph, of which mainly the long edge dummies created after crossing minimization generate additional work. It thus depends on the complexity of the subsequent coordinate assignment and edge routing steps how much the overall execution time increases. To get a rough idea of the overhead, consider Figure 3.28, which show results of executing the two wrapping variants for one graph each on a laptop with an Intel i7 2GHz CPU and 8GB memory: the sequentialized SCG `scg_seq.503.00` with 503 nodes and 574 edges, and the North graph `g.95.1` with 95 nodes and 132 edges. For each graph, ELK Layered[6] was run 20 times, and the fastest run was plotted. It can be seen that the overhead keeps within limits: the single-edge execution (SE) took about 18ms instead of 13ms, the multi-edge execution (ME) took about 12ms instead of 8ms. For the SE case, coordinate assignment took 5.6ms instead of 3.1ms. This is as to be expected since the used coordinate assignment algorithm runs in time linear to the number of nodes at that point, which, with increasing numbers of cuts, almost equals twice the number of original nodes. For the ME

[6] Using a variation of Brandes and Köpf's coordinate assignment algorithm [BK02] and orthogonal edge routes. When wrapping, MSD-1 is used, Wrii for the multi-edge case, and a drawing area with an aspect ratio of 0.25 is targeted.



(a) scg_seq.503.00, 11 cuts applied

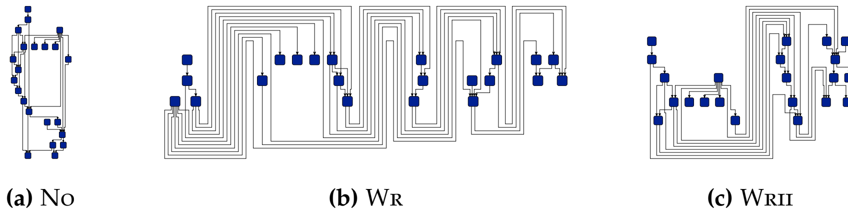


(b) g.95.1, 8 cuts applied

Figure 3.28. Execution time profiles of ELK Layered for (a) single-edge wrapping and (b) multi-edge wrapping compared to no wrapping (No). The barplots show the execution time in ms (y axis) of each layout processor (x axis) and the combined execution time in the very left pair of bars. Only the interesting layout processors are labeled: P1–P5 are the five steps of the layer-based approach, SEW is the sole processor for single-edge wrapping, and BP1–BP3 are the three processors for multi-edge wrapping. The total number of layout processors in (a) and (b) is different since they are assembled dynamically based on the graph's requirements [SSH14].

case, the crossing minimization contributes a significant amount to the total execution time and itself increases from 4.8ms to 6.4ms due to the added s and e dummies. The lower the execution time of an individual processor was, the larger was the variance between executions. This is also the reason for the slight increase in the execution time of the layer assignment step during ME, which works on the same graph in either case.

Alternatively, one could compute node coordinates and edge routes



(a) No **(b)** Wr **(c)** WrII

Figure 3.29. Different drawings of ATTar graph g.22.76, targeting a drawing area with an aspect ratio of 4.0 in cases (b) and (c). The drawings are scaled to a common height.

for each chunk separately, entirely avoiding the long edge dummies, and stitch the chunks together afterwards. However, this alters the layer-based approach's line of action in a way that may not be realizable in every existing implementation.

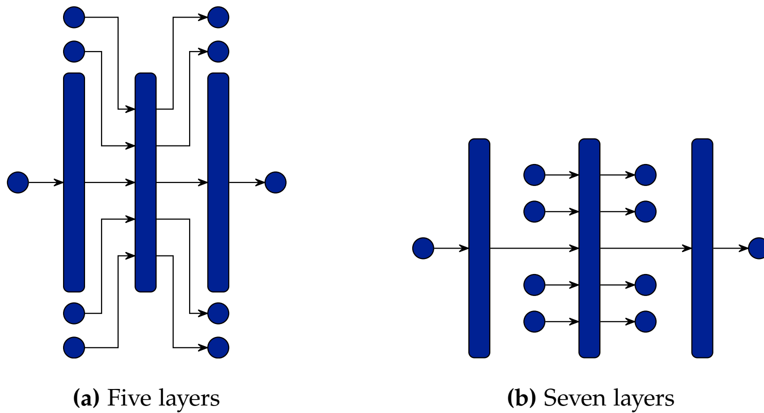
As mentioned, the method becomes less effective when many edges have to be cut, e. g. as it is the case for denser graphs. A problematic example can be seen in Figure 3.29. While the cut improvement significantly enhances the appearance of drawing (b), resulting in drawing (c), neither of the two is necessarily a clear improvement over the traditional layering (a). This problem can likely be addressed with alternative presentation styles, for instance: A set of backward edges in-between a pair of chunks could be *bundled* if it does not involve any edge crossings, since the order of the edges leaving at the first chunk is exactly the same order with which the edges enter the second chunk. It is therefore not necessary to follow the edge paths in detail. Likewise, the edge paths could be omitted completely and the connection between the bottom and the top part of the cut edges could be realized using labeling numbers.

3.4 High-Degree Nodes

As discussed during the introduction of the layer-based approach (cf. Chapter 2), traditional layer assignment methods focus on minimizing the number of layers or on minimizing the overall edge length. The latter implicitly keeps the number of layers low as well. In certain dataflow diagram types it is common that larger nodes collect and process a large number of signals originating from small nodes. Having few layers is not particularly suited for such a scenario. Large and small nodes end up in common layers and possibly result in tall diagrams with unfortunate edge routes, see Figure 3.30a for an example. This problem matches the stated challenge P-LA4 (see p. 46), and its importance in practice is further illustrated in Figure 3.31.

In what follows, the large nodes are called *high-degree nodes* and the small nodes connected to high-degree nodes are called *leaf nodes*. A simple solution to improve an existing layering is to move the leaf nodes to newly introduced layers which are adjacent to the high-degree node's layer. An example can be seen in Figure 3.30b. To address further scenarios, one can allow to move whole *trees* connected to the high-degree node instead of just the leaf nodes.

A procedure that can be incorporated as a post-processor after the layer assignment step to do this is described in Algorithms 3.6 and 3.7. A given layering L is iterated layer by layer and high-degree nodes are identified, that is, nodes with a degree larger than a threshold value t_{hdn}. For every high-degree node of a specific layer, attached trees of tree-height at most t_{th} are collected and separated into left trees (located in layers of lower index) and right trees (located in layers of higher index). Then, new layers are introduced on both sides of the currently handled layer, based on the highest tree on the corresponding side. No new layers are introduced on a side if no tree exists on that side. Afterwards, the previously identified trees are moved to the newly created layers. Note that the trees connected to high-degree nodes sharing the same layer are placed in the same set of newly created layers. All other nodes, just as the high-degree nodes themselves, remain in their original layers. The identification of trees can be realized as a recursive function isTreeRoot, as outlined in Algorithm 3.7. For ease of



(a) Five layers

(b) Seven layers

Figure 3.30. Illustration how introducing additional layers can improve the readability of a diagram.

description, the two functions a and d denote an ancestor function and a descendant function, which are either incoming or outgoing edges based on the side (left or right) for which trees are to be identified. Remember that the graph at this point is acyclic and that all edges point rightwards within the layering. Self loops can simply be ignored during the whole procedure. Hierarchical port dummies (cf. Section 2.6, p. 54), however, must potentially be placed in the very first or very last layer. Let v denote a hierarchical port dummy. To adhere to the layering constraint, one can either refrain from handling the whole subtree of v or just ignore v when moving the subtree. Further note that in the proposed solution a subtree's nodes are moved to the new layers such that adjacent nodes are in adjacent layers, i. e. the subtree is layered with the minimum number of layers and minimal edge length; it does not matter how the nodes of the subtree were placed in the initial layering, but it is also imaginable to preserve the previous layering.

Execution Time The algorithm is linear in the number of nodes, edges, and newly introduced layers. The number of new layers is bound by the number of nodes. As part of the outer loop, the nodes of every layer

Algorithm 3.6: Handling of High-Degree Nodes

```
Input: Graph G = (V, E) with layering L, threshold values t_hdn and t_th
for l ∈ L do
    H ← { v ∈ l | d(v) ≥ t_hdn }
    T_l ← {}, T_r ← {}                              // trees to be moved
    max_l ← 0, max_r ← 0
    for v ∈ H do
        (m_l, ϑ_l m_r, ϑ_r) ← findTrees(v, t_th)
        max_l ← max {max_l, m_l}
        max_r ← max {max_r, m_r}
        Add ϑ_l to T_l
        Add ϑ_r to T_r
    ll ← Insert max_l layers before l
    for root ∈ T_l do Move root's tree to ll
    lr ← Insert max_r layers after l
    for root ∈ T_r do Move root's tree to lr
```

are visited exactly once. Then, for each identified high-degree node, the isTreeRoot function is called for every connected node, identifying subtrees. As such, every edge of the graph is visited at most once. Finally, new layers are added and every node that is part of an identified subtree is moved to its new layer. The degree of a node does not change during this procedure and can therefore be queried in constant time. The same is true for the size of the set of incoming or outgoing edges of a node. Therefore, the running time is $O(|V| + |E| + |V| + |V|)$ or $O(|V| + |E|)$.

Remarks A contact from industry who works with Simulink diagrams on a daily basis reports that specifically handling high-degree nodes "helps a lot", and is "a must-have feature". Still, at least two points should be examined further in the future: First, currently, separate new layers are introduced for each of two high-degree nodes that are in adjacent layers of the original layering. While it feels natural to clearly separate the trees, it would be possible to share the set of new layers in-between their corresponding original layers, with the goal to reduce the overall width. Second,

Algorithm 3.7: Identifying Trees

Input: High-degree node n, tree height threshold t_{th}

$\vartheta_l \leftarrow \{\}$; $\vartheta_r \leftarrow \{\}$ `// sets of identified trees`
$m_l \leftarrow 0$; $m_r \leftarrow 0$ `// maximal tree heights`

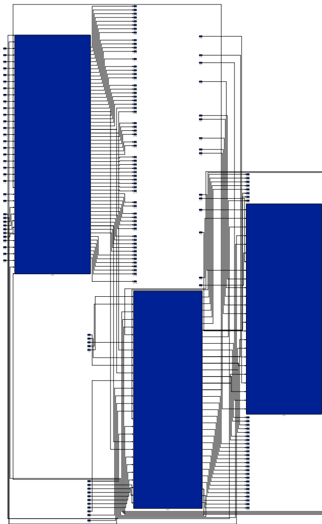
function isTreeRoot(v, $\mathcal{a}$, $\mathcal{d}$)
- **if** v is high-degree node **then return** -1
- **if** $|\mathcal{a}(v)| > 1$ **then return** -1
- **if** $|\mathcal{d}(v)| = 0$ **then return** 1
- $h \leftarrow 0$
- **for** $\{v, w\} \in \mathcal{d}(v)$ **do**
  - $sh \leftarrow$ isTreeRoot(w, $\mathcal{a}$, $\mathcal{d}$)
  - **if** $sh = -1$ **then return** -1
  - $h \leftarrow \max\{h, sh\}$
  - **if** $h \geqslant t_{th}$ **then return** -1
- **return** $h + 1$

for $w \in E_i(v)$ **do** `// collect western trees`
- $h \leftarrow$ isTreeRoot(w, E_o, E_i)
- **if** $h > 0$ **then**
  - $m_l \leftarrow \max\{m_l, h\}$
  - Add w to ϑ_l

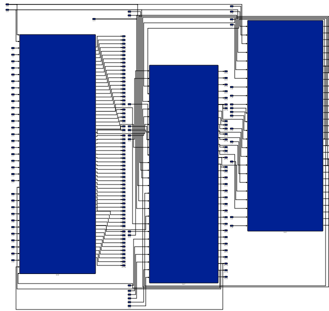
for $w \in E_o(v)$ **do** `// collect eastern trees`
- $h \leftarrow$ isTreeRoot(w, E_i, E_o)
- **if** $h > 0$ **then**
  - $m_r \leftarrow \max\{m_r, h\}$
  - Add w to ϑ_r

return $(m_l, \vartheta_l, m_r, \vartheta_r)$

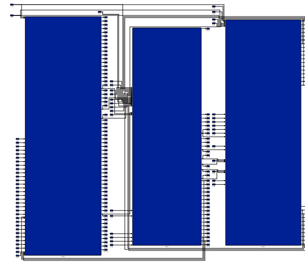
in the proposed solution the two threshold parameters on the degree of a node and on the maximal height of a tree have to be set by a user. Finding appropriate values is a tedious task, and simply setting large values for t_{th} and low values for t_{hdn} can result in rather wide drawings. It should be possible to identify reasonable values on a per diagram basis. Also, it may be reasonable to use different thresholds for different nodes and subtrees.



(a) Traditional layering



(b) Paying attention to high-degree nodes



(c) Flexible port positions and node sizes

Figure 3.31. Example derived from a Simulink diagram from practice. In (a) the problem addressed in this section can be seen. It stems from the nature of traditional layerings in conjunction with high-degree nodes. (b) shows a drawing of the same diagram in which the three high-degree nodes have been handled specifically. If one additionally applies the methods presented in Section 4.1, the drawing seen in (c) is possible.

Chapter 4

The Coordinate Assignment Step

The third step of the layer-based approach, the coordinate assignment step, assigns y coordinates to all nodes the graph is composed of at this point. That is, it assigns y coordinates to the nodes of the input graph as well as to the introduced dummy nodes. As mentioned during the initial introduction of the step (see p. 49), often pursued goals are the following [BK02; HN13]:

▻ short edges,

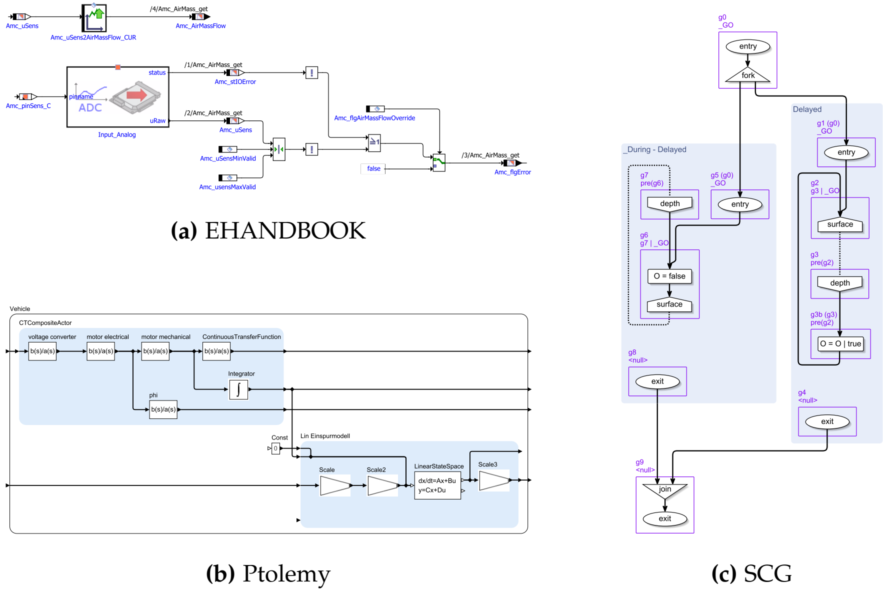
▻ balanced neighbors,

▻ straight long edges.

Desired drawings can be seen in Figure 4.1. A common idea in the literature is to tackle these goals with optimization problems that minimize the y coordinate difference between the source points and the end points of the edges. Building on existing methods, this chapter addresses the challenges arising from the existence of ports and the resulting special dummy nodes introduced for inverted ports and north/south ports, as well as from the orthogonally routed edges. To explain the challenges in further detail:

Ports Nodes have a height, and multiple edges can connect to the same node at different ports. Ports may be free to move alongside a node's border, as long as they maintain their relative order.

Inverted and North/South Port Dummies Both types of dummy nodes do not fit the classic concept of the layer-based approach in that the inverted port dummies are connected to their parent node via in-layer edges and in that the north/south port dummies do not have any connection to their parent node at all.



(a) EHANDBOOK

(b) Ptolemy

(c) SCG

Figure 4.1. Exemplary drawings using the coordinate assignment methods presented in Section 4.2.

Orthogonal Edges Wherever edges are routed orthogonally, edge bendpoints must be introduced between adjacent layers when the edge's source y coordinate differs from the edge's target y coordinate. Minimizing the number of bendpoints may become an additional goal that may take precedence over balanced node positions. This applies in particular to long edges as illustrated in Figure 4.2.

The next two sections present extensions to the approaches of Gansner et al. [GKN+93] and Brandes and Köpf [BK02]. The first one already supports fixed port positions and is flexibly extensible, the second one produces good results in the general case and runs fast [HN13]. In their original versions, neither of the two are particularly suited for orthogonally routed edges. Both sections use common notations with respect to the underlying port-based graph $G = (V, P, E, \pi)$. At this point of the layer-based approach,

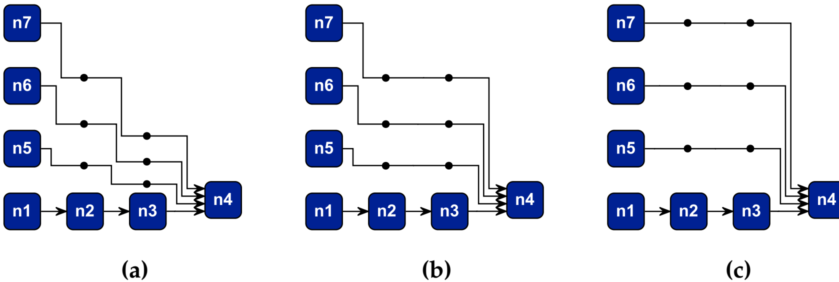


Figure 4.2. Illustrating the staircase effect of long edges. Small black circles represent dummy nodes. With regard to the total edge length, all three drawings are equivalent. However, they differ in the number of edge bendpoints.

the graph has only short edges, may contain several types of dummy nodes, has a layering $L = (L_1, \ldots, L_n)$, and the nodes within each layer have a particular order as computed by the crossing minimization step: $L_i = (v^i_1, \ldots, v^i_m)$, where $m = |L_i|$. The position of a node v^i_j in layer i is denoted by $pos(v^i_j) = j$. An edge is said to be *straight* if it is parallel to the x axis.

4.1 Network Simplex Approach

This section presents extensions to the coordinate assignment method presented by Gansner et al. [GKN+93], tackling challenges P-CA1, P-CA2, and P-CA3 as stated on page 50. The section is structured as follows. First, it outlines the basics of the method and shows how to incorporate inverted ports and north/south ports. It then explains how flexible port positions and flexible node sizes can be realized during coordinate assignment (P-CA2, P-CA3). Finally, it details how straight edges can take precedence over balanced node positions (P-CA1).

Gansner et al.'s method is based on the observation that just as the layer assignment step, the coordinate assignment step seeks for short edges, only that it does so in a different dimension. They therefore formulate the coordinate assignment problem as a layering problem and use their methods

for the Directed Layering Problem (DLP) (see Section 3.2), namely their network simplex algorithm, to solve it. Nevertheless, there are two essential differences between computing a layering and a coordinate assignment that will become clearer during the explanations of the remainder of this section: First, the positioning of a pair of nodes relative to each other is not prescribed this time, which it was by the partial order represented by the directed acyclic input graph during the layering step. And second, nodes connected by an edge are allowed to end up in the same "layer"; they are even desired to, since this would result in a straight edge within the final drawing.

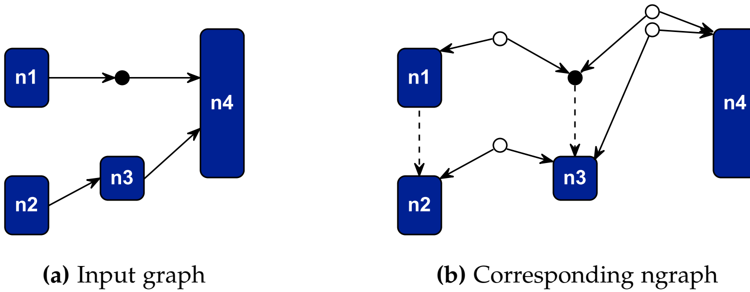
The Original Approach

The optimization problem that is pursued by Gansner et al. for coordinate assignment mapped to a port-based graph is the following. It minimizes the total sum of y coordinate differences between the source point and target points of the graph's edges. When edges are routed orthogonally, it can be simply viewed as minimizing the overall vertical edge length.

$$\text{Min.} \sum_{(p,q)\in E} \Omega_{(p,q)}\ \omega_{(p,q)}\ \left|\Big(y(\pi(p)) + y(p)\Big) - \Big(y(\pi(q)) + y(q)\Big)\right| \qquad \text{(A)}$$
$$\text{s.t.} \quad y(v_i) + h(v_i) + s_{v_i v_j} \leqslant y(v_j)$$
$$\forall v_i, v_j \in l : i + 1 = j,\ \ \forall l = (v_1, \ldots, v_{|l|}) \in L,$$

where each edge has an individual weight ω that can be specified by a user and encodes how important the edge is in terms of shortness and straightness. Further, $s_{v_i v_j}$ is a prescribed separation between pairs of nodes. As mentioned before, and as depicted in Figure 4.2, long edges are to be as straight as possible to avoid "staircases". For this purpose the Ω weights are used and set as follows: 1 – if both nodes of an edge are regular nodes, 2 – if one of the two nodes is a dummy node, and 8 – if both nodes are dummy nodes.

With this optimization problem in mind, Gansner et al. construct an auxiliary graph NG from G, such that an optimal "layer assignment" for NG represents an optimal coordinate assignment on G and vice versa. NG



(a) Input graph

(b) Corresponding ngraph

Figure 4.3. The auxiliary graph (b) for the input graph (a) as used by Gansner et al. to solve the coordinate assignment step using a layering algorithm. The small black circle represents a dummy node introduced after the layering step. In (b), the dashed edges represent separation preserving edges, and the white circles together with two solid edges each are surrogates for the edges of the input graph.

is called *ngraph* in the remainder of this section since it represents the graph that is fed to the network simplex algorithm. An exemplary ngraph can be seen in Figure 4.3.

The ngraph is a tuple $NG = (V_{NG}, E_{NG})$ of nodes and edges. Each directed edge $e = (u, v) \in E_{NG}$ connects a pair of nodes $u, v \in V_{NG}$ and carries real weights $\omega_e \geqslant 0$ and natural separation values $\delta_e \geqslant 0$. Nodes are dimensionless. Recall the definition of the DLP (p. **??**), which seeks for a mapping $\lambda : V_{NG} \to \mathbb{N}$:

$$\text{Min.} \sum_{(u,v) \in E_{NG}} \omega_{(u,v)} \big(\lambda(v) - \lambda(u)\big)$$
$$\text{s.t.} \ \big(\lambda(v) - \lambda(u)\big) \geqslant \delta_{(u,v)} \quad \forall (u, v) \in E_{NG}.$$

Note that contrary to the initial discussion of the DLP in Section 3.2, the δ values are allowed to be zero this time.

Next, the transformation of the port-based graph G to the ngraph NG is discussed. For ease of explanation, assume for the moment that no in-layer edges exist and that the edges connect to a node at the node's most northern point, i. e. as if for all ports p it holds that $y(p) = 0$. The next sections explain how to handle in-layer edges and relative port positions.

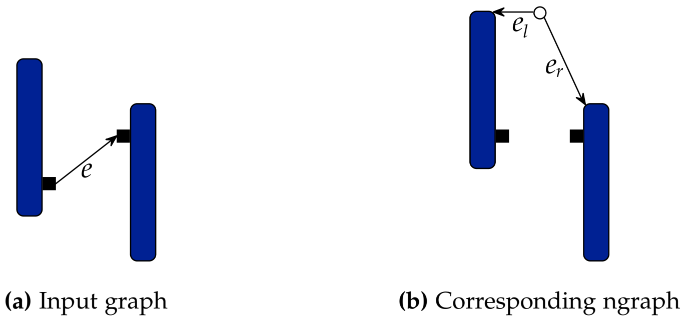
The nodes V of G are nodes of V_{NG}. Each edge (p, q) of E yields an additional dummy node d_{pq} in V_{NG}. The dummy node is connected to the two original nodes of the edge: $e_l = (d_{pq}, \pi(p))$ and $e_r = (d_{pq}, \pi(q))$ are added to E_{NG}. Essentially, this models the absolute function in (A). The weights are set to $\omega_{e_l} = \omega_{e_r} = \Omega_{(p,q)}\omega_{(p,q)}$ and the minimum edge lengths to $\delta_{e_l} = \delta_{e_r} = 0$. To preserve the order of the nodes within the layers as computed by the crossing minimization step and to adhere to spacing constraints, *separating edges* are introduced: for each pair of nodes u, v where u is the immediate predecessor of v within their common layer, an edge $e_{uv} = (u, v)$ is added to E_{NG} with $\omega_{e_{uv}} = 0$, i. e. it has no influence on the cost function, and with $\delta_{e_{uv}} = h(u) + s_{uv}$, where s_{uv} is the minimum vertical separation between nodes u and v. Note how the height of a node is encoded in the separating edge. After execution of the network simplex algorithm, the y coordinate of a node v, $y(v)$, is set to $\lambda(v)$.

Compared to the size of the graph that is passed to the network simplex algorithm during the layer assignment step, the ngraph can be significantly larger. This is for two reasons: due to the auxiliary elements discussed above and due to long edge dummies created after the layer assignment step that are now part of V. To be precise, the ngraph has $|V| + |E|$ nodes and $2|E| + |V| - |L|$ edges.

4.1.1 Ports

Ports whose positions are fixed relative to their parent nodes are addressed by Gansner et al. with altered separation values for the e_l and e_r edges. The idea is that one of the two edges is as long as the offset difference of the two involved ports, which is illustrated in Figure 4.4. Given an edge $e = (p, q) \in E$, the two separation values are set as follows: $\delta_{e_l} = \max\{0, y(q) - y(p)\}$ and $\delta_{e_r} = \max\{0, y(p) - y(q)\}$.

The explanations up to this point summarize the ideas presented by Gansner et al. adjusted to a port-based graph. Something that has not been considered by Gansner et al. are north/south ports and inverted ports, both of which result in further dummy nodes being introduced (cf. Section 2.6). To obtain acceptable final drawings, these dummy nodes require special treatment during coordinate assignment as explained next.



(a) Input graph

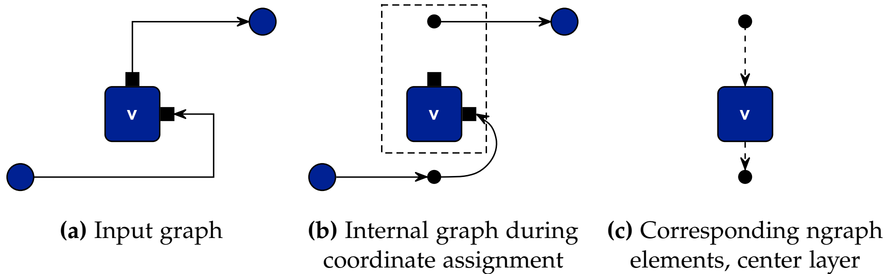
(b) Corresponding ngraph

Figure 4.4. Handling fixed port positions during coordinate assignment. The placement of the original nodes in (b) represents an optimal placement and would thus yield a straight edge in the final drawing.

North/South Ports Let d_p denote a dummy node that has been introduced for a north/south port p. d_p and $\pi(p)$ are located in the same layer, although not necessarily adjacent, and they are not connected by an edge. When there is no connection between the two in the optimization problem, the distance between d_p and $\pi(p)$ can become arbitrarily large; even if the lengths of other edges are unaffected. This unnecessarily elongates the edge connected to the north/south port in the final drawing.

To improve matters, an edge $e_p = (d_p, \pi(p))$ is added to E_{NG} if p is a north port, and $e_p = (\pi(p), d_p)$ is added if p is a south port (cf. upper part of Figures 4.5b and 4.5c). The δ_{e_p} is set to zero and ω_{e_p} is set to a user-configurable weight. A small weight $\omega_{e_p} \ll 1$ favors the shortness of edges connecting to the western or eastern side of a node, which should be the desired behavior in most cases. If d_p and $\pi(p)$ are adjacent within their layer, the e_p edge is redundant to the already introduced separating edge. In such a case it is enough to increase the weight of the separating edge, which is zero by default.

Inverted Ports Let d_p denote the dummy node that has been introduced for an inverted port p. d_p and $\pi(p)$ are located in the same layer and are connected by an in-layer edge. The relative order of the two nodes has



(a) Input graph

(b) Internal graph during coordinate assignment

(c) Corresponding ngraph elements, center layer

Figure 4.5. Handling north/south ports and inverted ports during coordinate assignment. The dashed box in (b) indicates that apart from long edge dummy nodes no other nodes must be placed in-between the north/south port dummy and v. Doing so would result in those nodes being crossed by an edge in the final drawing.

been determined during crossing minimization. The following description assumes d_p has been placed above $\pi(p)$, the other case works analog.

Let $e = \{p, q\}$ denote the edge incident to p. To keep the "inverted" part of e's route short, an edge $e_p = (d_p, \pi(p))$ is added to E_{NG} (cf. lower part of Figures 4.5b and 4.5c). The minimum separation is set to zero, and the weight is set to ω_e. If d_p and $\pi(p)$ are adjacent in their layer, the same simplification, reusing the separating edge, can be made as before for north/south ports.

4.1.2 Flexible Port Positions and Node Sizes

The handling of ports as discussed in the previous section requires the ports' positions to be fixed relative to their respective parent nodes. If this is not already the case within the input graph, it can be achieved by continuously rising the port constraint levels during the first three steps of the layer-based approach and by computing reasonable port positions prior the coordinate assignment step as described by Schulze et al. [SSH14]. A node's ports could, for instance, be evenly distributed on each side of the node.

However, to increase the number of straight edges incident to a node (P-CA2, P-CA3) and to further reduce the edges' lengths, one could let the coordinate assignment algorithm decide the relative positions of the

ports, and potentially adjust the heights of the nodes. Either behavior may be prohibited for certain nodes, e. g. due to the diagram's syntax. Thus, it must be possible to configure the behavior per node. The nodes for which one of the two options is allowed are referred to as *flexible* in what follows. Both options can be incorporated into Gansner et al.'s approach by adding further elements to the ngraph as described next.

Similar behavior has been proposed by Klauske in the context of laying out Simulink diagrams, who further alters the objective of the optimization problem and adds additional constraints [Kla12].

Flexible Port Positions The idea is to model each node $v \in G$ that allows flexible port positions by two dummy nodes d_n and d_s plus one dummy node for each west and east port and add all of them to V_{NG}. d_n represents v's northern border and d_s represents v's southern border. An example is shown in Figure 4.6. The separating edges incident to the predecessor node and the successor node in v's layer must connect to d_n and d_s, respectively. An edge $e_h = (d_n, d_s)$ is added to control the node's height. Since the network simplex method only supports to specify minimum edge lengths, not maximum edge lengths, the minimum separation is set to v's height and the weight is set to infinity (in practice to a sufficiently large value) since node heights are non-flexible at this point: $\delta_{e_h} = h(v)$, $\omega_{e_h} = \infty$.

Let $(p_1, \ldots, p_n)$ denote the west ports of v in the order from the northern border to the southern border, and let d_{p_i} denote the dummy node added to V_{NG} for p_i. Edges $(d_n, d_{p_1}), (d_{p_1}, d_{p_2}), \ldots, (d_{p_n}, d_s)$ are added to E_{NG}. The edges ensure that the computed port order is preserved and correspond to the separating edges for regular nodes. The minimum separation values are set to prescribed separation values either between a border of the node and the first or last port, or between pairs of ports. The weights of the edges are set to zero. The east ports of v are handled in the same way.

Representing flexible nodes in this way further increases the size of the ngraph: Let $V_f \subseteq V$ denote the set of flexible nodes, and let $P_f = \{p \in P : \pi(p) \in V_f \wedge \text{side}(p) \in \{\text{east}, \text{west}\}\}$ be the set of west and east ports of these nodes. $|V_f| + |P_f|$ additional dummy nodes are added to V_{NG} and $3|V_f| + |P_f|$ additional edges are added to E_{NG}.

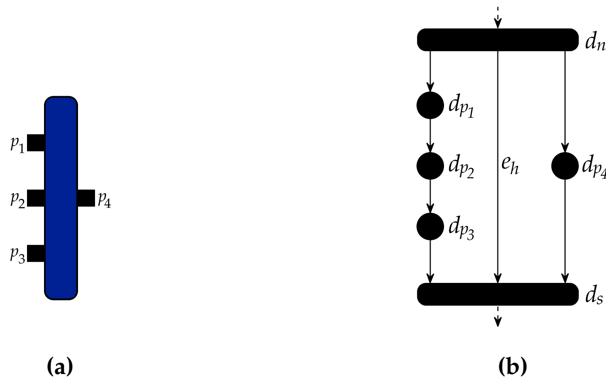


Figure 4.6. A flexible node with four ports (a), and the corresponding representation in the ngraph (b).

Flexible Node Heights To allow flexible node heights, the weight of the e_h edge is reduced. Instead of setting it to infinity, it is set to a value that is smaller than the weight of regular edges. A minimum height can be guaranteed by setting δ_{e_h} to the desired value; it can be set to zero if no minimum height is required.

4.1.3 Straightening Edges

As mentioned in the introduction of this chapter, routing edges in an orthogonal fashion may give rise to the desire to additionally reduce the number of edge bendpoints. For the coordinate assignment step this means that the y coordinate of the source of an edge must equal the y coordinate of the edge's target.

The method described so far allows many symmetrical solutions wherever it is not possible to reduce the y coordinate difference to zero, see Figure 4.7 for an illustration. The placements of (a) and (b) are equivalent from the point of view of the optimization problem that simply minimizes the sum of y coordinate differences. Solution (b) has fewer edge bends, however, as would have a solution in which alternatively the edge `(n3, n4)` is straightened. Klauske addresses this issue by statically altering the

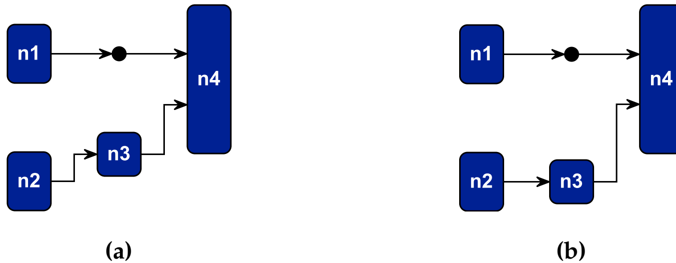


Figure 4.7. When minimizing total edge length, the solutions (a) and (b) are both optimal; the overall edge length cannot be reduced further, since the spacing between n1 and n2 is prescribed. However, (b) has less edge bendpoints when edges are routed orthogonally.

Ω weights of (A), before solving the optimization problem [Kla12, Section 3.3.1]. In the example, solution (b) would be favored by setting $\Omega_{(\mathtt{n2},\,\mathtt{n3})} > \Omega_{(\mathtt{n3},\mathtt{n4})}$. This has the drawback that only one of the two possible solutions that have fewer bends can be obtained. In a scenario in which n3 is blocked by a different node from below, for instance, no bend reduction could be achieved. This section therefore proposes an alternative approach: Altered weights are only used if they do not enforce one-sided decisions. Otherwise, a heuristic post-processing step is used that seeks to alter an ngraph with computed positions in a way that reduces the number of bends without worsening the objective.

Recall the desire to draw long edges as straight as possible to avoid a "staircase" effect. From the perspective of the coordinate assignment step, long edges are sequences of nodes, where the first and last node are regular nodes, all inner nodes are long edge dummies, and each consecutive pair of nodes in the sequence is connected by a single short edge. Let $(e_1, \ldots, e_n)$ denote such a sequence of short edges. The goal is to produce as many straight e_is as possible, which Gansner et al. address with the Ω weights. A drawback of setting $\Omega_{e_1} = \Omega_{e_n} < \Omega_{e_i}$, with $1 < i < n$, is that no tie is broken between the straightness of the first and the last short edge, i. e. it allows the scenario as depicted in Figure 4.2b (p. 135). In the example, two bendpoints could be saved for each of the long edges if the whole vertical

edge length is assigned to either the first or the last short edge, as seen in (c). However, the longer the long edge is, the less distracting it is if the first and last short edges are not straight.

Back to the original problem to increase the overall number of straight edges: avoiding staircases not only applies to long edges but also to sequences of regular nodes where pairs of consecutive nodes are connected by a single edge. An example is the node sequence `n2`, `n3`, `n4` in Figure 4.7. The idea to improve such cases is to identify paths within the input graph that can potentially be modified, either before or after executing the network simplex algorithm, such that the number of straight edges increases, without increasing the overall edge length.

Definition 4.1 (IPath)**.** An *improvable path* $\mathcal{P}$, short *ipath*, is a sequence of at least two edges, for which holds:

- $\mathcal{P} = (e_1, \ldots, e_m) = \big((s, p_1), (p_2, p_3), \ldots, (p_n, t)\big)$, $m \geqslant 2$,
- $\pi(p_j) = \pi(p_k)$ for all $j + 1 = k$, $j \in \{2z + 1 : z \in \mathbb{N}_0 \wedge z < \frac{n}{2} - 1\}$,
- $d^-(\pi(p_i)) = d^+(\pi(p_i)) = 1$ for all $1 \leqslant i \leqslant n$.

If the length of the sequence is exactly two, the ipath is called *short ipath*, and if it is larger than two, it is called *long ipath*.

A single depth-first traversal of the graph, starting at the graph's sinks, is sufficient to identify all ipaths. The identified ipaths are then processed as follows.

Long IPath For long ipaths, one can proceed in the same manner as for long edges. Before executing the network simplex algorithm, the Ω weights are altered: for $\mathcal{P} = (e_1, \ldots, e_m)$, set $\Omega_{e_1} = \Omega_{e_m} = 2$ and $\Omega_{e_i} = 8$ with $1 < i < m$. Obviously, this would not improve the situation for short ipaths: the same Ω weight would be assigned to the two involved short edges.

Short IPath Short ipaths are processed after the network simplex algorithm has been executed for the ngraph. A short ipath $\mathcal{P}_2 = ((s, p), (q, t))$ involves three nodes $v_l = \pi(s)$, $v_c = \pi(p) = \pi(q)$, and $v_r = \pi(t)$, all of which are part of V_{NG} and thus a position has been assigned to them. The

three nodes may be referred to as left, center, and right. Given a coordinate assignment represented by λ, four cases can be differentiated based on the relative positions of the three nodes to each other:

Case A: $\lambda(v_l) < \lambda(v_c) < \lambda(v_r)$,

Case B: $\lambda(v_l) > \lambda(v_c) > \lambda(v_r)$,

Case C: $\lambda(v_c) < \lambda(v_l)$ and $\lambda(v_c) < \lambda(v_r)$,

Case D: $\lambda(v_c) > \lambda(v_l)$ and $\lambda(v_c) > \lambda(v_r)$.

Figure 4.8 illustrates the cases and shows how each of them could be improved. In cases A and B it may be possible to alter v_c's position to either $\lambda(v_c) = \lambda(v_r)$ (A1/B1) or to $\lambda(v_c) = \lambda(v_l)$ (A2/B2). Both modifications would save two bends in a final drawing with orthogonal edges and are feasible if v_c is not blocked by another node in the same layer, either from above or from below. Note that it is possible that moving a specific v_c makes room for another center node to be handled that had been blocked before. Figure 4.9a shows an example: moving v_2 downwards allows to move v_1 downwards afterwards. Thus, it makes sense to iterate the nodes of a layer once from top to bottom and then a second time from bottom to top to leverage previous moves. Another question that emerges at this point is which move, upwards or downwards, should be favored if both of them are possible. An option is to select the shorter move, although this does not impact the overall edge length. Alternatively, one could favor similar moves, see Figure 4.9b for an example in which both center nodes have been moved downwards. The dotted line and the gray circle indicate the alternative position for v_1, resulting in a larger separation of the two edges.

Cases C and D cannot occur if λ represents an optimal assignment: in the depicted examples, v_c must be blocked by another node within the same layer (dashed areas), since otherwise either the left edge's length or the right edge's length could have been reduced further. Consequently, it is not possible to straighten an edge in these two cases, and it is not possible that the treatment of cases A and B results in a scenario where either of the cases C and D can be improved afterwards. This is due to the fact that A and B represent scenarios where the position of the v_c is somewhat free

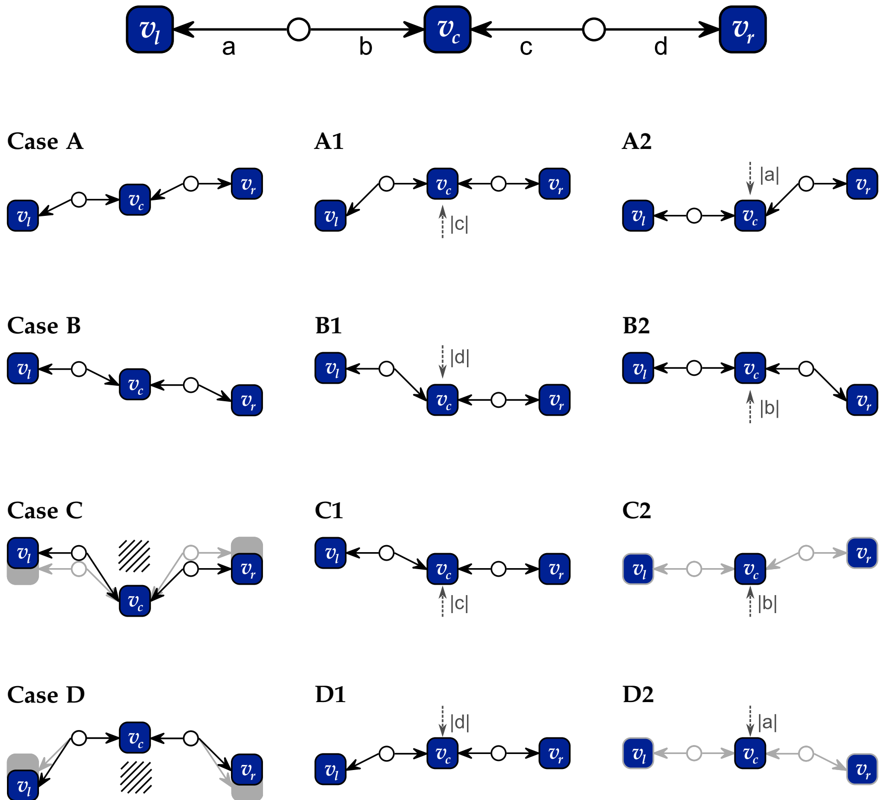


Figure 4.8. The four cases of short ipaths are shown in the left column. The center column and right column show for each case a possibility to move v_c such that two bends are saved in the final drawing. The gray arrows indicate the direction into which v_c has to be moved and are labeled with the necessary distance of the move. For instance, $|a|$ denotes that v_c must be moved by the (vertical) length of edge a. A move is only possible if no other node blocks the center node on the respective side (above or below). The circled nodes illustrate the ngraph's dummy nodes that are introduced for edges of the input graph.

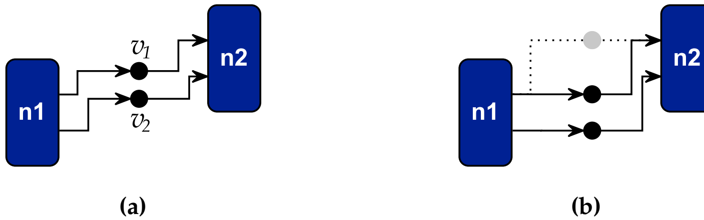


Figure 4.9. Illustrating the effect of different move orders and tie-breaking. (a) Initially v_1 can only be moved upwards and v_2 can only be moved downwards. After moving v_1 upwards, v_2 can both be moved upwards and downwards, or vice versa. (b) Moving both nodes into the same direction, say downwards, creates similar edge paths, while moving one node upwards and the other one downwards clearly separates the two edges (dotted path).

and does not impair the objective. If a case C or D were affected by an A or B, the computed placement would not be optimal. Nevertheless, the cases are included in the figure for future reference, as there may be alternative methods to increase the overall number of straight edges.

4.1.4 Remarks

Amongst other things, the previous sections discussed two ways to potentially increase the number of straight edges: a) allow ports to move on a node's border, and b) additionally allow a node's height to increase. Due to the nature of the layer-based approach, both points only apply to western and eastern ports. Edges incident to northern and southern ports cannot become straight since the (original) nodes connected by the corresponding edge must lay in different layers. Additionally, the coordinate assignment step does not decide on the nodes' x coordinates. To allow straight edges between opposing northern and southern ports, it would thus be necessary to weaken the notion of a layering and to add functionality to detect and handle such cases at multiple points of the algorithm, e. g. layer assignment, crossing minimization, and edge routing; at the cost of a significant increase in complexity. Moreover, both options should be applied with care and should usually not be activated for all nodes of a diagram, for at least

two reasons. For one thing, the syntax of a diagram type may implicitly prescribe the port positions and the dimensions of certain nodes. This is, for instance, the case when the ports' positions are related to certain graphical elements that are part of the node's rendering. Resizing the node, or moving ports, would then break this connection. For another thing, and this time in favor of the two options, diagram types often contain nodes that are predestined for resizing. To give an example, nodes of a dataflow diagram at which the dataflow forks or joins often have a large in- and/or out-degree. Allowing them to change their size may save edge length, bendpoints, and as a consequence diagram area as well (see Figure 3.31c, p. 132).

A point to improve on is the fact that in conjunction with both options, a) and b), the created drawings are unbalanced at times. For instance, the ports of a larger node may conglomerate at the northern part of one of the node's vertical sides, even in case they could be distributed evenly along that particular side without negatively interfering with the remaining drawing. From a practical point of view, this is a significant drawback. The reason is mainly the underlying method, the DLP, which solves a linear optimization problem and therefore cannot encode balance. The issue diminishes when carefully selecting the nodes for which flexibility is allowed. Still, further work is required here. To start, it should be possible to identify sets of nodes within the ngraph, after positions have been computed, that shall remain fixed relative to each other and to apply a balancing post-processing step afterwards. This applies in particular to those nodes which are connected by a straight edge.

Another point to be aware of is that the computed node coordinates are necessarily integral because the underlying method computes a "layering". This can become problematic when ports are present and the ports' positions are real-valued. If the positions of an edge's pair of ports (p, q) are fixed to different non-integer positions relative to the corresponding nodes, it is not possible to get a straight edge after coordinate assignment. This is particularly unpleasant when edges are routed orthogonally, since it results in small, but disruptive, kinks. In practice, however, it is usually acceptable to slightly alter the ports' positions to be integral prior to the coordinate assignment step.

4.2 Brandes and Köpf Approach

This section presents extensions of the coordinate assignment algorithm of Brandes and Köpf [BK02], tackling P-CA1 and P-CA2 (see p. 50). Parts of the discussed extensions have already been proposed by Carstens in his Master's thesis [Car12] and will be clearly marked in the following. The original algorithm assumes that all nodes have the same size and does not take ports into account; thus it straightens at most one incident edge per side per node. Here, both of these restrictions are removed, allowing drawings as the ones seen in Figure 4.1 (p. 134). Other than the previous section, which discussed extensions of Gansner et al.'s method, this section assumes that the coordinate assignment algorithm can neither alter the size of nodes nor the positions of ports. The underlying idea of Brandes and Köpf's algorithm is to create four extremal coordinate assignments and to combine them into a balanced final coordinate assignment. For brevity, a concrete coordinate assignment shall also be referred to as a "layout" throughout this section. Each extremal layout emerges from iterating the graph's layers either rightwards or leftwards, and from iterating the nodes within each layer either downwards or upwards. Within each extremal layout the goal is to minimize the total edge length by aligning nodes with their median neighbors. A neighbor at this is another node that is connected by either an incoming or outgoing edge, depending on the iteration direction.

The interested reader, who may consult the original paper [BK02] for further details, shall be warned that the explanations in this section are for a left-to-right layout direction whereas the paper assumes a top-to-bottom direction. Additionally, most of the following explanations are solely for an iteration direction of rightwards and downwards. The other three combinations work analog.

The Original Approach

This section summarizes the original algorithm of Brandes and Köpf, the discussion of new contributions starts in Section 4.2.1. The algorithm is divided into the following three steps, each of which is briefly outlined below.

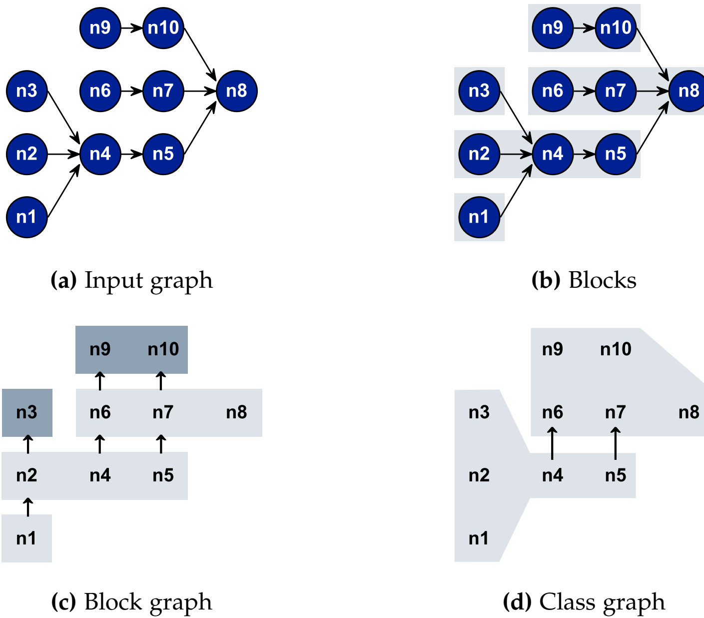


Figure 4.10. Gray boxes in (b) show calculated alignments (blocks) for the graph (a). (c) depicts the block graph with sinks in darker gray, and (d) shows the corresponding class graph.

Alignment Combine the nodes into so-called *blocks*. Different iteration directions may result in different blocks. Edges between the nodes in a block will be drawn straight in a final drawing.

Compaction Move the computed blocks as close to each other as possible and assign explicit y coordinates to the nodes. Depending on the direction, blocks are compacted either upwards or downwards.

Balancing Combine the four extremal layouts resulting from the previous two steps into a final drawing.

Alignment During the alignment step, as many nodes as possible are to be aligned with their median neighbor in the preceding layer. That is, for a node v with k incoming edges $(u_1, v), \ldots, (u_k, v)$ the median neighbor is $u_{(k+1)/2}$. As there is no median if k is even, Brandes and Köpf use one of the two values $m_1 = \lceil \frac{k+1}{2} \rceil$ and $m_2 = \lfloor \frac{k+1}{2} \rfloor$. If k is odd, m_1 equals m_2, and it depends on the iteration direction which value, u_{m_1} or u_{m_2}, is preferred. The idea behind using the median is that for a sequence $x_1, \ldots, x_k$ the sum $\sum_{1 \leqslant i \leqslant k} |x - x_k|$ is minimized if x is the median. An example alignment can be seen between node `n4` and node `n2` in Figure 4.10b.

Not every pair of nodes is alignable: wherever a pair of edges crosses, only one pair of the connected nodes can be aligned at a time. Brandes and Köpf distinguish and handle three types of *conflicts* depending on how many short edges, originating from long edges in the original graph, are involved in a crossing. Their goal is to favor those edges that would result in straight long edges in the final drawing. Since the details of the conflicts and how the alignments are computed are of minor relevance for the presented contributions, they are not discussed further. It shall be sufficient to know that certain edges may be flagged as non-alignable.

Nevertheless, it is necessary to know which nodes were aligned. Series of aligned nodes are referred to as *blocks*, see Figure 4.10b for an illustration.

Definition 4.2 (Block)**.** A *block* is a series of edges $(e_1, \ldots, e_n)$, connecting nodes in consecutive layers. The edges of a block are to be drawn straight in a corresponding drawing. Alternatively, the same block can be written as a series of the involved nodes: $(v_1, \ldots, v_{n+1})$.

A block b has a position $y(b)$, which is initially set to zero. In the original version of the algorithm, the block's position directly stands for the position of the block's nodes.

Compaction For compaction, Brandes and Köpf consider an auxiliary *block graph*, which is visualized in Figure 4.10c. Blocks are the nodes of the block graph and are connected by an edge if two nodes of different blocks are adjacent within their layer: for two blocks b_1 and b_2 an edge (b_1, b_2) is introduced for each $u \in b_1$ and $v \in b_2$ with $L(u) = L(v) \wedge pos(u) - 1 = pos(v)$.

The block graph is further divided into *classes*. Every block that is a sink within the block graph defines its own class. Examples are the two slightly darker blocks in Figure 4.10c. Every other block belongs to the class defined by the left-most sink it can reach within the block graph. Note that left-most is replaced by right-most if iterating the layers leftwards instead of rightwards.

Compaction then works in two steps. First, positions are assigned to blocks using a longest path layering (cf. Chapter 2, p. 57) within each class, which recursively assigns positions relative to the class's sink. Each pair of blocks is separated by a global separation value s_b. Second, the classes as a whole are moved as close to each other as possible, preserving s_b between adjacent blocks of different classes.

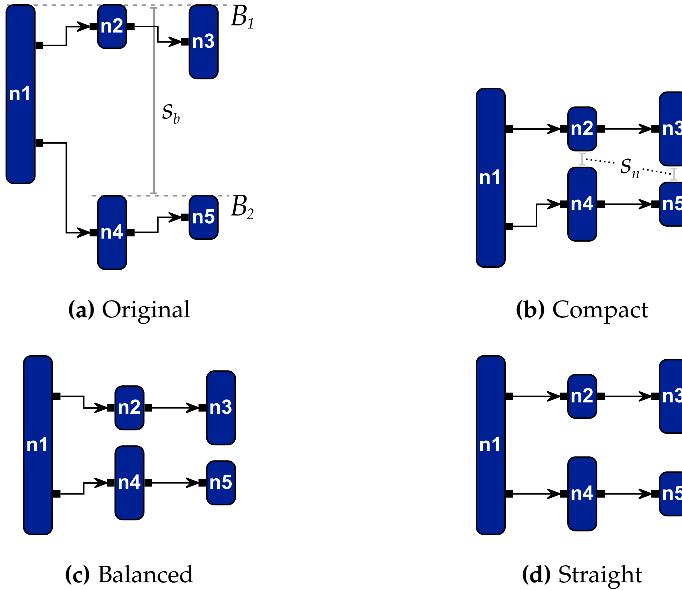
Both steps can be executed in one go by continuously remembering the required separations between classes whenever an adjacent pair of blocks of different classes is encountered during iteration and by applying the separations after all blocks have been placed. Note that iterating the nodes within the layers downwards, as is assumed during the following explanations, implies compacting the blocks upwards.

Balancing As mentioned earlier, the two previous steps, alignment and compaction, are executed four times with different iteration directions. This gives four layouts, two of which are compacted upwards and two of which are compacted downwards. To get a final layout, the four layouts are aligned such that the minimum coordinate of the upwards compacted layouts agrees with the minimum coordinate of the layout of smallest height, and such that the maximum coordinate of the downwards compacted layouts agrees with the maximum coordinate of the layout of smallest height. Afterwards, every node v has four positions $y_1(v) \leqslant \ldots \leqslant y_4(v)$. Brandes and Köpf compute v's final position as follows:

$$y(v) = \frac{y_2(v) + y_3(v)}{2} \quad . \tag{A}$$

This resembles what they call *average median* and is generally defined as

$$\frac{1}{2}\left(x_{\lfloor (k+1)/2 \rfloor} + x_{\lceil (k+1)/2 \rceil}\right) , \tag{B}$$



(a) Original

(b) Compact

(c) Balanced

(d) Straight

Figure 4.11. Illustration of the additional challenges for the Brandes and Köpf algorithm imposed by node sizes and ports. In (a) a global separation value s_b is used to space blocks B_1 and B_2, and ports are neglected. (b) shows a compact drawing, where ports are considered and the blocks can flow into one another using a local separation s_n between the nodes. (c) shows that executing the balancing step (as it is part of the original algorithm) in conjunction with orthogonally drawn edges results in edge bendpoints that are avoidable as illustrated in (d).

for k real values $x_1 \leqslant \ldots \leqslant x_k$. They proved that the average median preserves the order of the nodes within each layer and preserves prescribed separations (see Lemma 4.3, p. 158). Moreover, Brandes and Köpf mention that an advantage of the average median, as opposed to simple averaging, is that it creates straight edges whenever a pair of nodes is aligned in two of the extremal layouts, while being positioned asymmetrically in the remaining two layouts.

Remarks As mentioned earlier, the original approach does not cater for varying node sizes and ports. Ports pose two problems that are illustrated in Figure 4.11a. First, even though all nodes of the blocks B_1 and B_2 are neatly top-aligned, not a single edge is drawn straight in the depicted drawing. And straight edges are arguably the more important aesthetics criterion in this case as opposed to top-aligning. Second, node n1 has two ports, both of which would allow the connected edge to be drawn straight. Yet, n1 and n4 are part of different blocks that will be separated during the compaction step. Furthermore, different node sizes render the global separation value s_b between blocks impractical. s_b would have to be larger than the tallest node of the graph to avoid overlapping nodes, possibly leaving a lot of whitespace. Figures 4.11b to 4.11d show drawings that are more desirable, which use a local separation value s_n between pairs of adjacent nodes. The following sections address all of these issues.

The introduction of this chapter mentions that balanced node positions are not always desired when edges are routed orthogonally. For this reason, Carstens suggests to consider the balancing step to be optional and to select one of the four extremal layouts as the final layout instead, based on a certain decision criterion [Car12]. Two reasonable criteria are the total height of the drawing and the total number of bendpoints.

4.2.1 Ports and Node Sizes

This section explains the necessary extensions of the original algorithm to account for ports and varying node heights. The original algorithm assigns the same y coordinate to all nodes within a block. This automatically yields straight edges if all nodes have the same size and the same attachment points for edges. With ports this is not true anymore. Therefore, an *inner shift* is computed for every node, which represents the necessary offset of a node within its block so that all edges of the block are drawn straight. Figure 4.12 illustrates the effect of the inner shift. Additionally, to address varying node heights, the compaction step is modified to consider node heights when calculating y coordinates for blocks. The inner shift and blocks of non-zero height have already been proposed by Carstens [Car12], however, not that every node's individual height is considered during compaction.

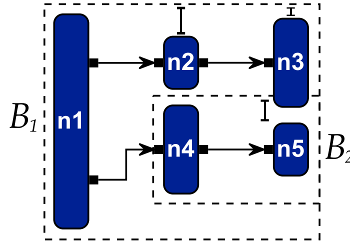


Figure 4.12. Illustration of the inner shift and the interweaving of blocks. Nodes `n2`, `n3`, and `n5` have an inner shift value different from zero. The inner shift defines the offset of a node within its block and is depicted by the spacing markers. Furthermore, block B_2 reaches into the bounding box of B_1, resulting in a compact coordinate assignment.

The following explanations describe the extensions in a declarative fashion. This is different from the paper of Brandes and Köpf and the conference paper presenting the extensions [BK02; RSC+15], which employ an algorithmic presentation that may prove valuable when actually implementing the method. The different presentation has been chosen to allow an incremental explanation of the individual ideas without having to worry about the concrete control flow.

Inner Shift Given a set of blocks $\mathcal{B}$ as computed by the alignment method of the original algorithm, an inner shift $\sigma(v)$ is computed for every node $v \in V$. Let $b = ((p_1, q_1), \dots, (p_n, q_n))$ denote a block. The inner shift is then computed for $1 \leqslant i \leqslant n$ as follows:

$$\sigma(\pi(p_1)) = 0 \tag{C}$$

$$\sigma(\pi(q_i)) = \sigma(\pi(p_i)) + y(p_i) - y(q_i) \tag{D}$$

The computation can yield negative inner shift values. To agree on the top-most position of the blocks, the values are offset such that the smallest inner shift value of a block is zero. At this point the height of a block b is final:

$$h(b) = \max_{v \in b} \big(\sigma(v) + h(v)\big) \tag{E}$$

This height was used by Carstens to determine block positions and to compute the total height of each extremal layout, which in turn may be used as decision criterion for the final layout in case the final balancing step is omitted. Here, this height will not be used, since it is not fine-grained enough to achieve the best compaction.

Block Compaction Given an inner shift for the nodes of each block, compaction of the blocks can be performed using the block graph. Contrary to the original method, the inner shift and the heights of the nodes are considered when computing the y coordinate of a block. Note that the individual height of every node is used and not the overall height of a block. This allows blocks to "flow" into each other, as seen in Figure 4.12.

Further notations are necessary before the computation of a block's position can be explained properly: Remember that the block graph $BG = (\mathcal{B}, BE)$ is a tuple of the computed blocks $\mathcal{B}$ and the edges $BE \subseteq \mathcal{B} \times \mathcal{B}$ that originate from pairs of nodes u and v that are adjacent within a layer of the original graph's layering. Let g denote a mapping that maps an edge $e = (b_1, b_2) \in BE$ to the pair of nodes of the underlying graph: $g(e) = \langle u, v \rangle$ where $u \in b_1$ and $v \in b_2$. The $\langle\rangle$ brackets are used for the pair to avoid confusion with the notation of an edge. Let $\mathcal{C}$ denote the set of classes. The block graph BG can be restricted to a certain class $c \in \mathcal{C}$, which is written as $BG_{|c}$. In such a case, the set of blocks of the restricted graph is $\mathcal{B}_{|c} = c$, and the set of edges is $BE_{|c} = \{(b_1, b_2) \in BE : b_1 \in c \wedge b_2 \in c\}$.

The *adjacent blocks* of a block b that belong to the same class c are connected via an outgoing edge of b: $adj_{|c}(b) = \{b^* : (b, b^*) \in BE_{|c}\}$. The *neighbor pairs*, which essentially dictate a block's position, are pairs of nodes of the original graph: $np_{|c}(b) = \{\langle u, v \rangle = g((b, b^*)) : (b, b^*) \in BE_{|c}\}$. Further, let $b(v)$ denote the block of node v, and let $c(v)$ denote the class of node v.

As in the original algorithm, the blocks of each class's restricted block graph can be positioned recursively in the fashion of a longest path layering; only that the computation of each block's y coordinate is more intricate. For each $c \in \mathcal{C}$, a block $b \in c$ can be positioned as soon as all $b^* \in adj_{|c}(b)$ have been positioned. Since the block graph is acyclic by construction, one can start with the graph's sinks and iteratively assign positions until all blocks

have been positioned. The position of a block b then is:

$$y(b) = \max_{\langle u,v\rangle \in np_{|c}(b)} \left[\Big(y(b(v)) + \sigma(v) + h(v) + s_n \Big) - \sigma(u) \right]. \tag{F}$$

Class Compaction The previous step compacted the blocks within each class as much as possible, with the sink block of each class being positioned at zero. The positions of blocks belonging to different classes may thus either still overlap or offer further compaction potential. Therefore, now the classes themselves must be moved as close to each other as possible without overlapping. For that a *class graph* can be considered: $CG = (\mathcal{C}, CE)$, where CE is a subset of the block graph's edges: $CE = \{(b_1, b_2) \in BE : c(b_1) \neq c(b_2)\}$. Corresponding to the block graph, *adjacent classes* and *neighbor pairs* are defined for the class graph. Given a class c, the adjacent classes are $adj(c) = \{c(b_2) : (b_1, b_2) \in CE \wedge c(b_1) = c\}$, and the neighbor pairs are $np(c) = \{\langle u, v\rangle = g((b_1, b_2)) : (b_1, b_2) \in CE \wedge c(b_1) = c\}$.

Given these definitions, a position is computed for every class in the same fashion as was done for the block graph. The position is then used to offset the classes' nodes. Let $c \in \mathcal{C}$ be a class for which all $c^* \in adj(c)$ have been positioned. $c's$ position is:

$$y(c) = \max_{\langle u,v\rangle \in np(c)} \left[\Big(y(c(v)) + y(b(v)) + \sigma(v) + h(v) + s_n \Big) - \sigma(u) \right]. \tag{G}$$

The final block positions are computed as follows, resulting in a maximally compact and overlap-free layout:

$$y_o(b) = y(b) + y(c) \quad \text{for all } c \in C, b \in c. \tag{H}$$

Final Node Coordinates The computed block positions and the inside shifts give the final node coordinates:

$$y(v) = y_o(b) + \sigma(v) \quad \text{for all } b \in \mathcal{B}, v \in b. \tag{I}$$

Balancing Given the four extremal layouts computed with the modified compaction step, the same balancing procedure can be applied as discussed above (p. 152). For this to work, however, the average median must preserve

the order of the nodes within their layers and the prescribed spacing values even if nodes have varying heights. That this is the case can be shown with a small addition to Brandes and Köpf's lemma, which originally reads as follows.

Let v and w be a pair of adjacent nodes within the same layer, i. e. $pos(w) - 1 = pos(v)$. By construction of the different extremal layouts $1 \leqslant i \leqslant n$ (usually $n = 4$), positions are computed for the two nodes such that $y(v)_i + \delta \leqslant y(w)_i$ holds for any spacing value $\delta > 0$. In particular, this spacing value can be a different one for each pair of nodes. The median is computed over a set of ordered values. Therefore, let y' denote a reordering of the y such that $y'(v)_1 \leqslant \ldots \leqslant y'(v)_n$ and $y'(w)_1 \leqslant \ldots \leqslant y'(w)_n$.

Lemma 4.3. *The average median is order and separation preserving. That is,*

$$\frac{1}{2}\left(y'(v)_{\lfloor (k+1)/2 \rfloor} + y'(v)_{\lceil (k+1)/2 \rceil}\right) + \delta \leqslant \frac{1}{2}\left(y'(w)_{\lfloor (k+1)/2 \rfloor} + y'(w)_{\lceil (k+1)/2 \rceil}\right).$$

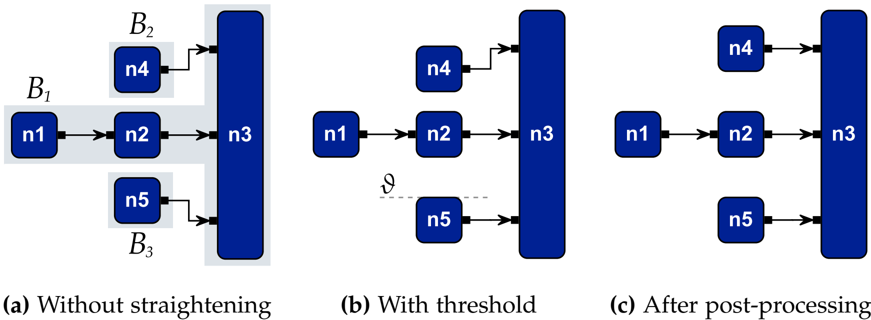
It follows for nodes with heights and the prescribed spacing value s_n:

Corollary 4.4. *The average median is order and separation preserving if the nodes have a height.*

Proof. For the pair of nodes v and w, set $\delta = s_n + h(v)$. □

4.2.2 Straightening Edges

The extensions discussed in the previous section enable the coordinate assignment method of Brandes and Köpf to properly draw diagrams that include ports and nodes with varying heights. This section addresses the desire to favor, at least to a certain extent, straight edges over the most compact layout possible (P-CA2). Figure 4.12 shows an example in which the taller node `n1` allows for more than one incident edge to be drawn straight, at the risk of increasing the drawing's overall height. The original algorithm did not have to address this since nodes were considered to be uniform. There are two parts to the solution proposed here. For one thing, the compaction step is slightly modified, and for another thing, a simple post-processing step is executed after the compaction step.



(a) Without straightening **(b)** With threshold **(c)** After post-processing

Figure 4.13. Illustration of the procedure to straighten additional edges during the compaction step. A threshold value is used to prevent n5 in (b) from being compacted "too far". (c) shows the desired final result, obtained by shifting n4 upwards during a post-processing.

Remember that the extended compaction step, discussed in the previous section, compacts blocks and classes as much as possible. This implies that for a given iteration direction only such edges are possible candidates for additional straightening where one of the involved blocks was moved "too far", for instance, see the edge (n5, n3) between blocks B_3 and B_1 in Figure 4.13a (when compacting upwards). To stick to the notion of "aligning nodes", such an edge is referred to as a *desired alignment* of two nodes belonging to different blocks. The corresponding blocks should thus be placed in a manner that allows the edge to be drawn straight. Figures 4.13b and 4.13c illustrate the two parts of the proposed solution. Without modification, everything is compacted as much as possible as seen in (a). In (b) a threshold value ϑ prevents the block B_3 from moving further upwards, resulting in an additional straight edge. The threshold value is computed based on the edge (n5, n3). (c) shows the desired final result in which the edge (n4, n3) was straightened during a post-processing step.

The post-processing step is necessary because a threshold value can only be determined during the compaction process if one the involved blocks has already been assigned a position. Consider Figure 4.13a and the iteration direction rightwards and downwards, which means compacting the blocks

upwards. The algorithm has to place block B_2 before it can place block B_1. Consequently, when looking at the edge $(\texttt{n4}, \texttt{n3})$, no threshold can be calculated because node n3, or rather its block, has not been placed yet. In such a case, the straightening of the outgoing edge of n4 is delayed until all blocks have been placed.

Desired Alignments There are different possibilities to select desired alignments. A simple solution is to select them greedily on the fly while iterating through the graph during the compaction step. For every edge that is visited, it is checked whether it is a candidate for a desired alignment. If so, and if it can be handled straight away, the aforementioned threshold value is computed and applied as explained below. Otherwise, the edge is recorded for post-processing. Moreover, only those edges are considered that are incident to either the first node of a block or the last node of a block. Note that a block can consist of a single node, in which case both cases are true.

This procedure can be improved at at least three points: (1) be more intelligent in selecting an edge instead of just using the first one that is encountered; (2) check whether edges between nodes that are neither the first nor the last in a block are candidates for desired alignments, there may be multiple of them per block; and (3) when multiple edges incident to a block are candidates to be drawn straight (although not simultaneously), choose the one that allows the most compact layout. Nevertheless, the greedy procedure mentioned above already selects many edges that, to a human, are obvious candidates for straightening.

Threshold A computed threshold shall affect the positioning of a block, as initially defined in (F), as follows. Let b denote a block to be positioned, and let $e = \{p, q\}$ denote an edge that has been selected as desired alignment. It is necessary to work with an undirected edge here since it can be both an incoming and an outgoing edge of b's nodes. Assume that $q's$ node is in b: $\pi(q) \in b$. Furthermore, it is required that the other block $b^* = b(\pi(p))$ has already been placed and that the two blocks belong to the same class: $c(b) = c(b^*)$.

(F) computes the most compact position $y(b)$ for block b. To fulfill the desired alignment, a threshold value is derived from e and the new position for b is computed subject to this threshold:

$$\vartheta(b,e) = \Big(y(b^*) + \sigma(\pi(p)) + y(p)\Big) - \Big(\sigma(\pi(q)) + y(q)\Big) \tag{J}$$

$$y_\vartheta(b) = \max\{\, y(b),\, \vartheta(b,e)\,\} \tag{K}$$

For the next computations of yet unplaced blocks that depend on b's position, $y(b)$ is then considered to take the value of $y_\vartheta(b)$.

The max in (K) is due to compacting upwards, and would be a min when compacting downwards. Where desired, one can limit the maximum difference of the initial block position and the adjusted block position at this point. A reason to do this might be to avoid a layout from becoming too tall in order to draw a single edge straight.

Post-Processing As mentioned above, certain desired alignments must be postponed as it is not always possible to address them immediately. Due to the way the edges are iterated during the compaction step, it is possible to collect postponed desired alignments in a queue and process them one after another afterwards. To illustrate, consider Figure 4.13 again. Imagine a further node `n4'` connected to `n3` and located between `n4` and `n2`. Just as `n4`, it will be delayed. To give both edges a chance to be straightened, it is important to post-process `n4` prior to `n4'`. Using a queue allows to do exactly this. Care has to be taken, however, that blocks that are already part of a desired alignment, e. g. due to a previously applied threshold value, must not be moved.

As soon as all blocks and classes have been positioned, the desired alignments in the queue can be processed one after another, each time checking for the involved edge and block how far the block can be shifted without overlapping other nodes. This way, the processed edge becomes either straight or shortens as much as possible. The required computations to come up with a new block position are the following. Let b be the block to be post-processed and e be the edge that should become straight. Further, let $np(b)$ be the set of neighbor pairs of b that are not restricted to a certain class: $np(b) = \{\langle u,v\rangle = g((b,b^*)) : (b,b^*) \in BE\}$. Two values need to be

known to safely shift a block: ς_s, the extent by which b has to be shifted upwards for e to become straight, and ς_{no}, the maximum extent by which b can be shifted upwards without introducing overlaps.

$$\varsigma_s = \vartheta(b,e) - y_o(b) \tag{L}$$

$$\varsigma_{no} = -y_o(b) + \max_{\langle u,v\rangle \in np(b)} \Big[\Big(y_o(b(v)) + \sigma(v) + h(v) + s_n \Big) - \sigma(u) \Big] \tag{M}$$

$$y_{pp}(b) = y_o(b) - \min\{\, |\varsigma_s|, |\varsigma_{no}| \,\} \tag{N}$$

Note that the threshold in (L) must be newly computed at this point. Since both ς values are negative when compacting upwards, b's position if shifted by the smaller amount of the two. Again, the shift is performed even if the desired alignment cannot be fulfilled since it reduces the length of the edge e. $y_{pp}(b)$ replaces $y_o(b)$ for further computations.

4.2.3 Remarks

The remaining paragraphs of this section discuss several intricacies of the balancing step in conjunction with node heights and edge straightening, and of dummy nodes introduced for north/south ports and inverted ports.

Balancing The introduction of individual node heights has adverse implications for the balancing step as it was defined by Brandes and Köpf. Since blocks are now non-uniform in height and are allowed to asymmetrically interweave, the four extremal layouts are less "symmetrical" in comparison to each other. As a consequence, it does not necessarily hold anymore that straightness prevails for twice aligned nodes (cf. p. 152). An example can be seen in Figure 4.14a. The edge (n1, n3) would be straight after applying the original algorithm, at the cost of moving n2 downwards and increasing the diagram's height.

The situation can get worse when the edge straightening strategy proposed in the previous section is applied. The way desired alignments are chosen within the individual extremal layouts is not necessarily symmetric among the four layouts. This collides with the idea to have four extremal layouts that, taken together, compensate each other's biases. As a conse-

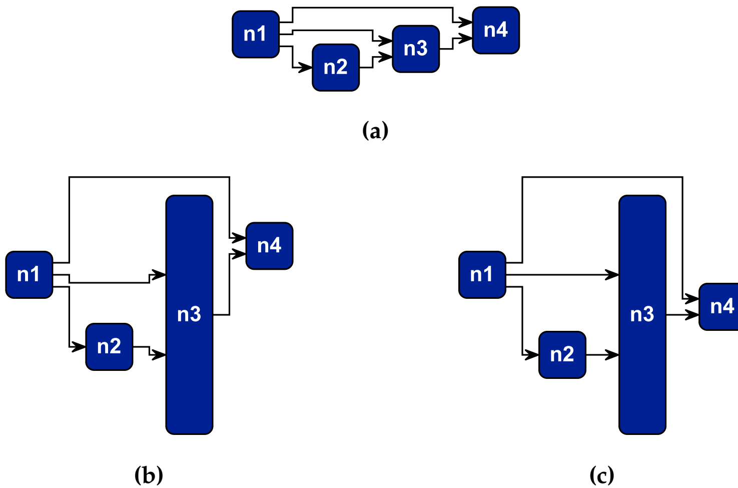
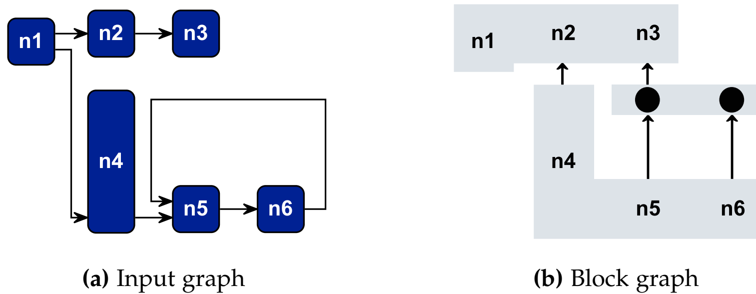


Figure 4.14. Issues with the final balancing step: ports, node sizes, and the procedure to straighten further edges may result in the unfortunately balanced drawings (a) and (b). (c) shows a drawing without balancing applied.

quence, balancing in conjunction with straightening may result in drawings as the one seen in Figure 4.14b. The edge (`n1`, `n3`) makes an unnecessary dip downwards, which would not exist without the additional straightening procedure.

Nevertheless, the effect usually keeps within limits and only occurs when the final balancing step is used. As an illustration, (c) shows a drawing with the maximum number of straight edges, which is one of the four extremal layouts. Still, both mentioned balancing problems should be addressed in the future.

North/South Ports and Inverted Ports Both types of ports result in special dummy nodes being created, with inverted ports additionally yielding in-layer edges and north/south ports yielding partly unconnected dummy nodes. Carstens addresses this situation by simply ignoring both types of dummy nodes in the sense that he regards them as plain long edge



(a) Input graph

(b) Block graph

Figure 4.15. Internally, two inverted port dummy nodes are created for the edge (n6, n5). The dummy nodes are connected to n5 and n6, respectively, by in-layer edges. Ignoring the in-layer edges during the alignment step results in "floating" blocks, as seen in (b), that yield unnecessarily elongated edges.

dummies. Additionally, he omits the in-layer edges connected to inverted port dummies whenever a node's edges are iterated during the algorithm. While this enables the algorithm to work with those particular types of ports, the resulting drawings are unsatisfactory at times. See Figure 4.15 for an example.

Matters can be improved for inverted ports if one allows the post-processing step to select in-layer edges as candidates for straightening. Obviously, it is not possible to straighten an in-layer edge, however, the involved blocks are moved as close to each other as possible, reducing the edge's length. Furthermore, it is possible to handle the dummy nodes of north/south ports in the same way, by temporarily connecting them with the nodes they originate from. This enhancement is part of, and depends on, the straightening step. Thus, it is not available where no straightening is desired.

Implementation Details The descriptions above give an intuition of how the modified compaction step and the straightening method proceed and explain how concrete positions are computed. They give, however, no concrete implementation hints. Pseudo code that details an efficient implementation can be found in the two corresponding papers [BK02; RSC+15]. Brandes

and Köpf show that all three steps can be realized in linear time and require only a small number of arrays that hold pointers to the graph's nodes. Also, it is enough to iterate the graph's nodes once per step within the selected iteration direction to compute all required information. The extensions to the original algorithm as discussed above integrate seamlessly with this and preserve, depending on the post-processing strategy, the overall linear execution time. The inner shift values can be computed in time linear to the number of edges that are part of blocks. The size-aware block compaction only adds constant time operations to the computation of a block's position. The strategy to straighten further edges can be realized in linear time as well: to select an edge, the edges incident to a block are looked at at most once; calculating and comparing threshold values are constant time operations; adding elements to and removing them from a queue can be done in constant time as well; finally, it must be checked during the post-processing step how far blocks can be moved into a certain direction, which looks at each node of the graph at most once.

4.3 Discussion

The previous two sections presented modifications and extensions of the coordinate assignment methods of Gansner et al. and Brandes and Köpf. When compared to each other in their original form, each method has its strengths and weaknesses. The same applies to the modifications discussed here, which target specific problems that may or may not be present for certain diagram types. Thus, there is no definite answer to the question which method, in conjunction with which modifications, is the best. The following evaluation is therefore to be understood as an attempt to verify that the methods behave as expected when applied to diagrams from practice. To make the discussions easier to follow, the coordinate assignment methods and their variants are abbreviated as follows:

- Brandes and Köpf's approach extended with ports and node sizes without balancing, either without additional edge straightening (BK) or with edge straightening (BKS); and
- Gansner et al.'s approach with edge straightening (NS) and with the option to enlarge nodes where reasonable (NSN).

Based on the nature of each of the aforementioned variants, the following four hypotheses are expected to hold:

H1 compared to BK, BKS reduces the number of edge bends at the cost of an increased drawing height;

H2 compared to NS, NSN reduces the number of edge bends at the cost of an increased drawing height;

H3 both BK and BKS produce fewer edge bends than NS; and

H4 regarding execution time, BK is faster than NS, NS faster than NSS.

The variants were applied to two different diagram types: flattened Ptolemy diagrams drawn from left to right and SCGs drawn from top to bottom. See Section 1.4 for an introduction of both diagram types and Section 1.9 for a detailed description of the concrete graph instances. Flexible node sizes were allowed during NSN for nodes that have at least two ports on one of the relevant sides, i. e. on the west or east side for a left-to-right layout.

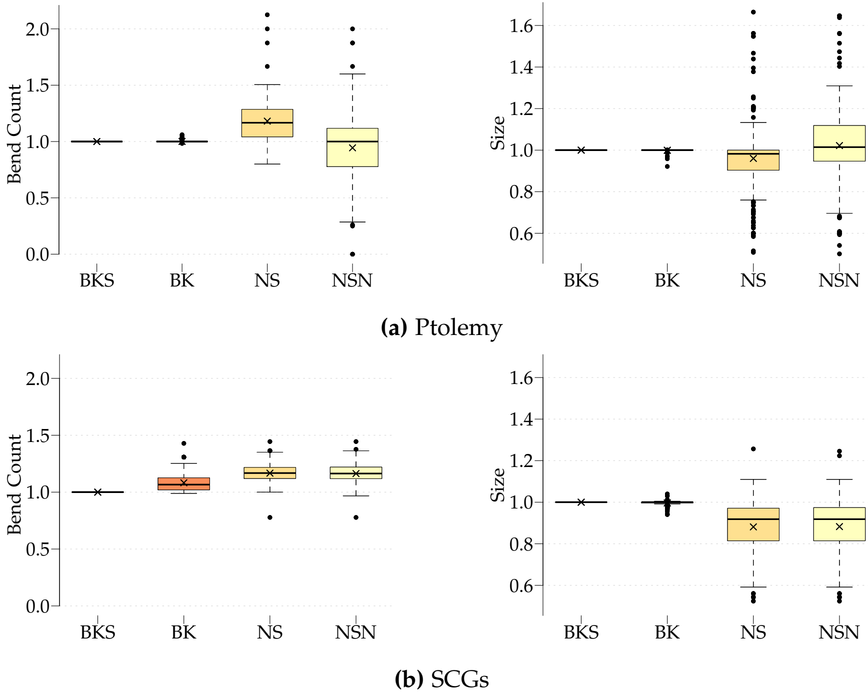


Figure 4.16. Results on the number of edge bends and on the size of the final drawing for the four variations of the discussed coordinate assignment methods. The size is either the width or the height, depending on the layout direction. Values are normalized for each graph instance with respect to the result of BKS, and for each measure the same y axis range is used.

Figure 4.16 shows the results for the number of edge bends and the drawing area size, both of which are normalized with respect to BKS's results. Thus, a bend count value of another method that is larger than 1.0 indicates that BKS produced less bendpoints than the other method. Accordingly, a size value larger than 1.0 means that BKS produced less size.

For both diagram types the number of bends reduces for BKS compared to BK, supporting H1. However, a significant change can only be seen for

the SCGs. The size increases slightly for a small number of Ptolemy diagram instances and about a third of the SCGs. Surprisingly, the straightening reduces the size of 21 SCGs, which can be explained as follows. The SCGs are hierarchical, and it turns out that in the affected cases certain straightening decisions within hierarchical nodes allow a more compact placement on higher hierarchy levels.

NS and NSN, compared to each other, behave as hypothesized in H2 for the Ptolemy diagrams: the number of bends reduces on average if one allows flexible node sizes while the average size slightly increases. The SCGs do not profit from flexible node sizes, which is likely due to the fact that most of the nodes have a single incoming and a single outgoing edge. As hypothesized in H3, the BK variants produce less bends than NS on average and for most of the graph instances. NS yields smaller drawings on average instead. However, there is a non-negligible number of graph instances for which BK's drawing is smaller in turn.

Figure 4.17 shows the execution times of the methods plotted against the number of nodes during coordinate assignment. The node counts are on average 1.47 times larger than the node counts of the input graphs, with a standard deviation of 0.46. The evaluation was performed on a laptop with an Intel i7 2GHz CPU and 8GB memory. It can be seen that for up to 200 nodes all methods finish in under 50ms, which makes them fast enough for interactive applications. BKS remains fast with increasing numbers of nodes: it stays below or close to 10ms for up to 500 nodes. This is different for the other two methods, which require up to 500ms for 500 nodes. H4 can be verified: BKS is significantly faster than NS, which in turn is faster than NSN.

Summarizing the observations, in general and when looking at the right set of graphs, the stated hypotheses hold. Nevertheless, the presented results include many outliers, which are both due to the heterogenous data sets and due to the fact that the methods are sensitive to the concrete graph instance.

To conclude this chapter, extensions and modifications to two coordinate assignment methods have been presented that allow them to be flexibly adjusted to the needs of a broader range of diagram types. The approach of Gansner et al. has been modified to optionally allow flexible port positions

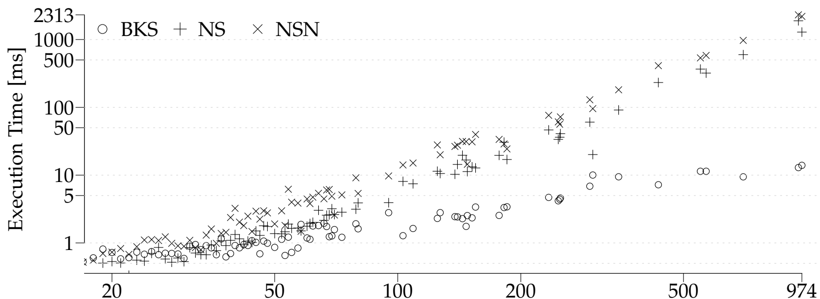


Figure 4.17. Execution time plotted against the number of nodes for the tested Ptolemy graphs. Both axes are logarithmic. For each node count value the maximum execution time of any graph instance with that node count is plotted.

on a node's border and to allow a node to resize in height, both for the case that it increases the total number of straight edges. Furthermore, it has been shown how certain edge paths can be post-processed, again to increase the number of straight edges. Due to the underlying optimization method, execution times increase significantly for larger graphs.

The approach of Brandes and Köpf has been extended to support different node sizes and ports, which makes it usable for a wider range of diagram types, such as dataflow diagrams. Additionally, modifications to further increase the number of straight edges have been presented. At its core, the method first and foremost aims at straight edges, potentially at the cost of a larger drawing size. The method is fast, even for larger graphs.

The presented extensions of both methods leave room for further work with practical relevance, two points in particular:

Balance Improve BK with respect to combining the four extremal layouts without impairing straightness, and improve NS with respect to finding balanced port positions when flexible nodes are involved.

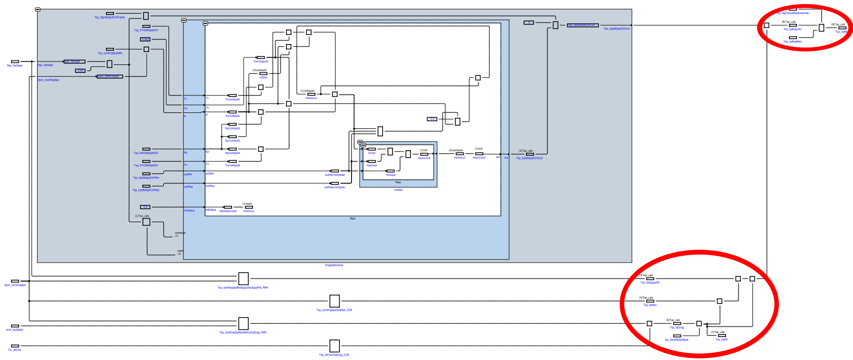
Compactness Address item P-CA4 (p. 50), where the goal is to identify a preferably small set of edges and to intentionally draw those edges non-straight to obtain a more compact drawing.

Chapter 5

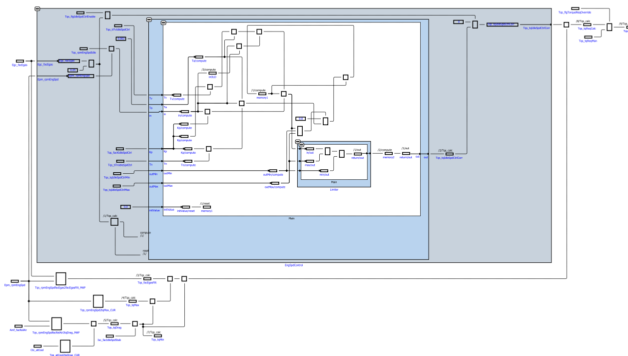
Post-Processing: Width Reduction

The previous chapters were concerned with improving several known methods for various steps of the layer-based approach. This chapter, on the other hand, discusses a new method that aims at reducing the width of the created drawings and is executed as a post-processing step after the five traditional steps. It addresses challenges P-LA1 and P-LA3 as stated in Section 2.2 (p. 46). The underlying idea of the presented method is to use the simple and well-studied technique of *one-dimensional compaction* to pull a drawing's elements together as much as possible. For this to be effective, it must be possible to move the elements of the diagram in a way that is less restrictive than prescribed by the usual rigid layering. Since the compaction ought to be executed as post-processing, it is unproblematic to abandon the layering since the traditional algorithm that requires it has completed. An example can be seen in Figure 5.1. A lot of space is saved in (b) by abandoning the rigid layering and by compacting everything as much as possible within the x dimension. As mentioned in Chapter 2 (p. 43), computing somewhat relaxed layerings has been proposed before in the context of the layer assignment step itself, the used methods impose additional challenges on the subsequent steps, however. Contrary to that, the post-processing step proposed here runs independently.

This chapter is structured as follows. First, the technique of one-dimensional compaction is reviewed. Second, it is explained how the technique can be used in the context of graph layout and the layer-based approach. In particular, Sections 5.2 and 5.3 explain what has to be considered when compacting either dataflow diagrams or state diagrams. The chapter then closes with a small evaluation to assess the method's practicality and a discussion.



(a)



(b)

Figure 5.1. Illustration of applying one-dimensional compaction to a layer-based drawing of a dataflow diagram. (a) An automatically drawn diagram with traditional methods. Circled nodes are pushed to the right due to the rigid layering. (b) The same diagram after post-processing. The diagram's width is reduced by about 16 % and the average edge length is reduced by over 50 %.

5.1 One-Dimensional Compaction

One-dimensional compaction is a well-known technique to minimize the area occupied by a set of objects in the plane. As opposed to the often NP-hard two-dimensional compaction problems, it can be solved efficiently in time $O(n \log n)$, n being the number of objects to compact [Len90]. Lengauer thoroughly discusses one-dimensional compaction in the context of VLSI design and presents methods for several, quite general, variations [Len90]. Here, only those concepts are discussed that are relevant for the remainder of this thesis. The interested reader is referred to the well-written book for further details.

Let $\mathcal{R}$ denote a set of rectangles in $\mathbb{R}^2$. A rectangle $r = (r_x, r_y, r_w, r_h)$ is a quadruple of the rectangle's top left position $(r_x, r_y) \in \mathbb{R}^2$ and its size $(r_w, r_h) \in \mathbb{R}^{+2}$. Let the union ($\cup$) of a pair of rectangles be defined as the set of points $(a, b) \in \mathbb{R}^2$ that are covered by either of the rectangles (hence the resulting set does not necessarily describe a rectangle), and let the intersection ($\cap$) be the set of points that are covered by both of them.

Definition 5.1 (Left of, overlap horizontally)**.** Similar to Lengauer, r is said to be *left of* s with $r, s \in \mathcal{R}$ if $(r_x + r_w < s_x)$, and r and s are said to *overlap horizontally* ($r \prec s$) if

$$(r_y < s_y + s_h) \quad \wedge \quad (s_y < r_y + r_h) \quad \wedge \quad r \text{ left of } s.$$

A set of rectangles $\mathcal{R}$ is *valid* if $\forall r \neq s \in \mathcal{R} : r \cap s = \emptyset$. The one-dimensional compaction problem in the x dimension seeks for a transformation of a valid set of rectangles $\mathcal{R}$ into a valid set of rectangles $\mathcal{R}'$ by changing x coordinates only and by preserving the order $\prec$, such that the width w is minimized, with

$$w = \max_{r' \in \mathcal{R}'} \ (r'_x + r'_w) \ - \ \min_{r' \in \mathcal{R}'} \ r'_x.$$

Note that the case to compact the height can be defined analogously in the y dimension.

Two steps are executed to solve the problem. First, a *constraint graph* is derived from a given set of rectangles. It encodes which rectangles overlap horizontally. Second, the constraint graph is used to position the rectangles in a way that yields minimum width.

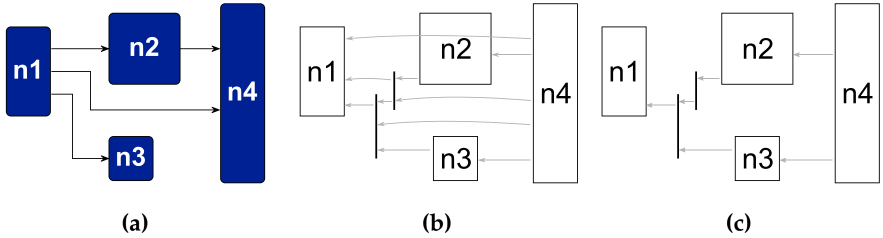


Figure 5.2. A set of rectangles (a) with two possible constraint graphs. The vertical segments of edges are represented by rectangles. The constraint graph in (b) contains redundant constraints, which were removed in (c).

Constraint Graph

The constraint graph $CG = (\mathcal{R}, C)$ is a tuple of a valid set of rectangles $\mathcal{R}$, i. e. the nodes of the graph and a set of constraints C. The constraints $C \subseteq \mathcal{R} \times \mathcal{R}$ form the directed edges of the graph. An edge $c = (s, r)$ with $s, r \in \mathcal{R}$ is added to CG if $r \prec s$. Note that this graph is acyclic by construction.

Naively, the constraint graph can be constructed in $O(n^2)$ time by checking for every pair of rectangles if they overlap horizontally, n being the number of rectangles. However, this way the number of edges is in $O(n^2)$ and several edges may be redundant. Consider Figure 5.2b, where the constraint between the pair of rectangles `n1` and `n4` is transitively guaranteed by the two constraints (`n1`, `n2`) and (`n2`, `n4`). Figure 5.2c shows a constraint graph without redundant constraints.

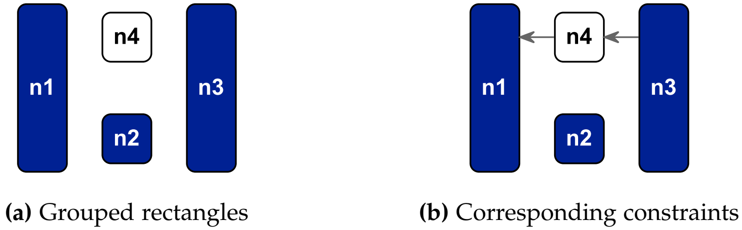
Lengauer shows how the constraint graph can be computed using a scanline technique in time $O(n \log n)$ and with $O(n)$ edges [Len90]. The procedure is illustrated in Algorithm 5.1. Let $Y^- = \{r_y : r \in \mathcal{R}\}$ denote the set of the upper coordinates of all rectangles and let $Y^+ = \{r_y + r_h : r \in \mathcal{R}\}$ denote the set of lower coordinates. For a point $p \in Y^- \cup Y^+$, let $r(p)$ denote the rectangle for which p was added. The scanline processes the points in increasing order from top to bottom. An array (cand) is used to hold constraint candidates. The set S, ordered based on the rectangles' x coordinates, allows to query the (current) predecessor (left(r)) and successor (right(r))

Algorithm 5.1: Constraint Graph Computation

```
Input: ℛ: set of rectangles
Data: cand[r]: constraint candidates indexed by rectangle,
S: sorted set of rectangles
Output: C: set of constraints

points ← Y⁻ ∪ Y⁺
Sort points ascendingly (prioritizing Y⁺)
for p in points do
    if p ∈ Y⁻ then
        r ← r(p)
        Put r into S
        cand[r] ← S.left(r)
        cand[S.right(r)] ← r
    else
        if S.left(r) exists ∧ S.left(r) = cand[r] then
            Add (r, S.left(r)) to C
        if cand[S.right(r)] = r then
            Add (S.right(r), r) to C
        Remove r from S
```

of a rectangle r according to the order. Its implementation must provide insert and delete operations that run in $O(\log n)$ time and constant time operations to access neighbor elements. When the scanline encounters an upper coordinate, it adds the corresponding rectangle to S and updates the constraint candidates. When a lower coordinate is encountered, the scanline "finishes" the corresponding rectangle by removing it from S and by checking the set of candidate constraints, possibly adding them to the constraints of the final graph. By definition of $\prec$, no constraint must be added between a pair of adjacent rectangles r and s if the lower coordinate of one of the rectangles equals the upper coordinate of the other rectangle, e. g. $r_y + r_h = s_y$. This can be achieved by prioritizing Y^+ over Y^- during scanline execution. For the correctness of this procedure and minimality of the resulting constraint graph, see Lengauer [Len90].



(a) Grouped rectangles **(b)** Corresponding constraints

Figure 5.3. Nodes n1, n2, and n3 are in the same group. Cycles within the grouped constraint graph prevent compaction: n1 must be positioned before n4, however, n4 must be positioned prior to n3. Since n1 and n3 belong to the same group, they must be positioned simultaneously.

Grouping

In certain use cases it is required that two or more rectangles keep their relative positioning to each other. An example are a rectangle that represents a vertical edge segment incident to north/south ports and a rectangle that represents the node which the edge is incident to (cf. Figure 5.9b, p. 186, for an illustration). The segment does not overlap horizontally with the node and thus may detach during compaction. To avoid this, rectangles are *grouped*, which is similar to the concept of *grouping constraints* used by Lengauer. Formally, the constraint graph $CG = (\mathcal{R}, C)$ is extended to a *grouped constraint graph* $GCG = (\mathcal{R}, C, G)$ where every rectangle is part of a group. A group $g \in G$ is a non-empty set of rectangles $g \subseteq \mathcal{R}$ where every rectangle has an offset $r_\delta \in \mathbb{R}^2$ to an imaginary origin (g_x, g_y). By convention, the left-most rectangle of a group has an offset of zero. During compaction, the relative positions between grouped rectangles are to be preserved by the algorithm. A grouping is *valid* if no rectangle is in more than one group: $\forall g_i \neq g_j \in G : g_i \cap g_j = \emptyset$. Any constraint $c = (r, s) \in C$ can be neglected if r and s are in the same group. Let $g(r), r \in \mathcal{R}$, stand for r's group. $out(g)$ denotes the outgoing constraints of g, i. e. $\{(r, s) \in C : r, s \in \mathcal{R} \wedge r \in g\}$, and $in(g)$ denotes the incoming constraints. $\delta^+(g) = |out(g)|$ denotes the *out-degree* of group g, i. e. the number of constraints leaving g, $\delta^-(g) = |in(g)|$ denotes the *in-degree*.

It is possible to group nodes in a way that yields a cyclic grouped

constraint graph. Consider Figure 5.3. The filled nodes n1, n2, and n3 form a group g. To fix n4's position, the position of g, and thus the positions of all of g's nodes, must be fixed first because of the constraint (n4, n1). However, n3 requires that n4 has been positioned because of the constraint (n3, n4). Thus, neither of the four nodes can be positioned in this scenario. The depicted scenario corresponds to a directed cycle: $g \in out(g(\texttt{n4}))$ and $g(\texttt{n4}) \in out(g)$. It is important to keep this in mind when using the compaction algorithm that is presented in the next section since it requires the input graph to be acyclic.

Compaction

The minimum width that can be achieved is bound by the longest path of the grouped constraint graph. To obtain a placement with minimum width, Algorithm 5.2 can be executed. Initially, the method sets the tentative positions of all groups to zero. It then selects the sinks of the graph and performs three actions. Let g denote such a sink:

1. Turn g's tentative position into its final position and position g's rectangles based on the specified offsets.
2. Update the tentative positions of groups g^* that are constrained by g to $\max\{g^*_x, \max_{r \in g}(r_x + r_w)\}$.
3. Remove g's incoming edges $in(g)$ from the graph.

The procedure is repeated until all groups have been placed. It takes linear time and is guaranteed to terminate if the input graph is acyclic.

This closes the brief review of one-dimensional compaction. The next section turns to the question how to use it in the context of graph layout.

5.2 Dataflow Diagrams

Working with dataflow diagrams laid out by the layer-based approach, I frequently observed scenarios where wide nodes prevent more compact drawings by pushing other nodes far to the right in order to respect the rigid layering. Figure 5.1 already exemplified such a scenario, and it is the

Algorithm 5.2: Group Compaction

```
Input: GCG = (ℛ, C, G): grouped constraint graph
g_x ← 0   ∀g ∈ G
sinks ← { g ∈ G : δ⁻(g) = 0 }
while sinks not empty do
    g ← sinks poll
    for r ∈ g do
        r_x = g_x + r_δ
    for (s, r) ∈ {(s, r) ∈ C : r ∈ g} do
        g(s)_x = max{g(s)_x, r_x + r_w}
        Remove (s, r) from C
        if δ⁻(g(s)) = 0 then
            Add g(s) to sinks
```

initial motivation for the work presented in this chapter. The method is not limited to dataflow diagrams, though. The subsequent section, Section 5.3, illustrates how the presented compaction technique can be adjusted to work for state diagrams as well. The main difference there is that edges are often drawn using splines instead of in an orthogonal fashion.

First, however, this section shows how to convert a given drawing of a dataflow diagram into a one-dimensional compaction problem. It pays special attention to the peculiarities of dataflow diagrams: orthogonally routed (hyper-)edges, inverted ports, and north/south ports. It furthermore discusses how to sensibly consider the lengths of edges as well as prescribed spacing values that are to be preserved between graph elements.

Representing a Dataflow Diagram with Rectangles

The goal is to represent all relevant elements of a diagram using rectangles. Remember that the one-dimensional compaction technique discussed above requires a set of non-overlapping rectangles. After applying the layer-based approach this is mostly the case, in particular: (1) nodes and edges do not overlap, (2) prescribed spacings between graph elements are respected, and (3) the connection point of an edge on the corresponding node's perimeter

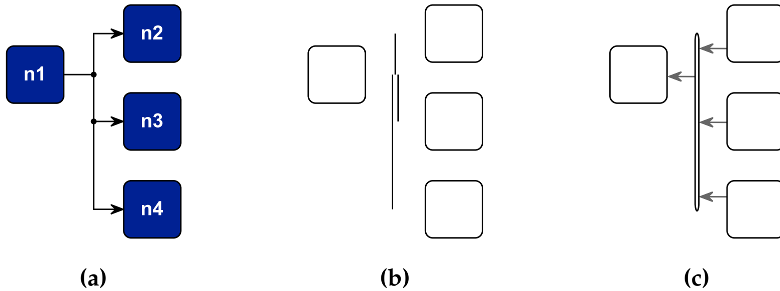


Figure 5.4. Depending on the representation of hyperedges, vertical segments of edges have to be merged. Assume that the hyperedge seen in (a) is represented by three individual edges `(n1, n2)`, `(n1, n3)`, and `(n1, n4)` whose vertical segments are simply drawn on top of each other. To illustrate, the three vertical segments are drawn next to each other in (b). For compaction they are merged into a single rectangle as shown in (c).

is fixed. Note that if overlaps would exist, one could try to remove the overlaps before performing compaction, which is a well-studied task in itself [DMS06; GH09; NNB+16].

As mentioned before, dataflow diagrams can be seen as directed hypergraphs, and as explained in Section 1.3 hypergraphs can be translated into regular port-based graphs. To that end, each hyperedge is represented by a set of regular edges. The translation allows to use traditional layer-based methods without the requirement to specifically address hyperedges. However, it also means that multiple segments of different edges, all representing the same hyperedge, may lay on top of each other – a fact that must be considered in the following and is illustrated in Figure 5.4.

Let $PG = (V, P, E, \pi)$ denote the given port-based graph representing a laid out dataflow diagram. For one-dimensional compaction, the nodes V form the set of rectangles $\mathcal{R}$ where each rectangle represents a node's bounding box. Furthermore, for every vertical segment of an edge $e \in E$, a rectangle with corresponding height and width is added to $\mathcal{R}$. The width corresponds to the edge's thickness within the drawing. The following explanations simply assume a unit width. In case the diagram contains

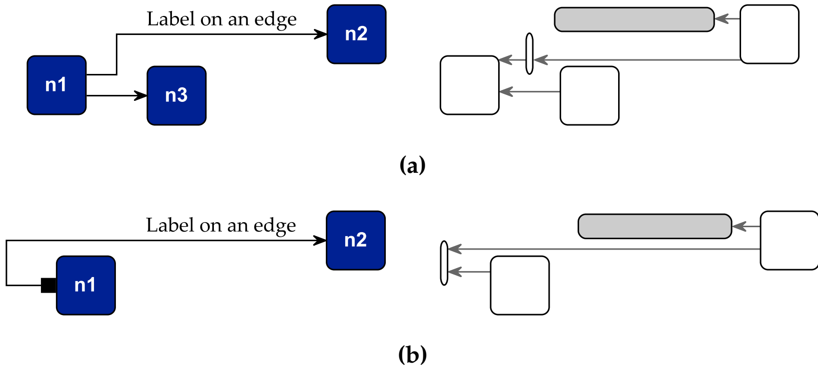


Figure 5.5. Small diagrams (left) and the corresponding constraint graphs (right). Edge labels can be considered during compaction by adding a label's bounding box to the set of rectangles (filled boxes). Care must be taken that a rectangle that represents an edge label stays close to the edge: there is no leftward constraint for either of the depicted edge labels. It is thus possible that the label ends up to the very left of the drawing, completely detached from the corresponding edge. (b) additionally illustrates potential for compaction that occurs in the presence of inverted ports: the edge label can well be moved above `n1`.

hyperedges, some of the rectangles created for the edge segments may overlap and must be merged prior to compaction. This can be done efficiently and is explained in further detail in Section 5.2.2. The horizontal segments of the edges do not have to be considered at this point. However, during the compaction process the length of an edge may change implicitly, and usually short edges are desired. This topic is discussed separately in Section 5.2.1. Because of assumptions (1)–(3) and the potential merging of the vertical segments' rectangles, the set of rectangles $\mathcal{R}$ is valid and a constraint graph can be computed.

Another important element of diagrams are labels. Both nodes and edges can carry labels. How to consider them during compaction depends on the way the labels are modeled during layout. For instance, node labels that are placed inside the bounding box of the node itself do not require additional treatment. If the labels are placed outside of the node itself, the

node's bounding box can be enlarged to comprise the label. Edge labels are more demanding: In ELK Layered, edge labels are often modeled as dummy nodes and added to the graph itself for the time the layout algorithm is executed. This allows to reserve enough space for labels during both the placement of nodes and the routing of edges. Consequently, the labels do not overlap with any other graph element and their bounding box can safely be added to $\mathcal{R}$. Nevertheless, care has to be taken during compaction that a label remains connected to the corresponding edge. The rectangle that represents the label during compaction is not necessarily constrained by other elements, as is illustrated in Figure 5.5. A possible solution is to manually add constraints between the rectangle that represents the label and the adjacent vertical segments of the edge. If the edge's path does not have a vertical segment or only on one side of the label, the label should already be constrained by a node.

The next sections discuss further facets of dataflow diagrams that must be considered to get satisfying results. In particular, solely minimizing the width of a drawing is not sufficient to get aesthetically pleasing drawings. Consider Figure 5.6b, where placing node `n2` further to the right would reduce the length of the incident edge. Consequently, it is important to minimize, or at least consider, the total edge length during width minimization.

5.2.1 Edge Length

One-dimensional compaction itself is not aware of the edges of the input diagram and thus does not consider the length of the individual edges. As just mentioned, this is illustrated in Figure 5.6b. The problem of finding positions for a graph's nodes that minimize the total edge length is known from Chapter 3, where it is referred to as DLP. The problem has also been studied in the context of VLSI design, where the length of electrical wires on a circuit board should be minimized [Len90; HT92].

Next, two solutions are suggested that are simple and specifically tailored to graph layout. Nevertheless, it might be worthwhile to explore and compare further existing algorithms, in particular the ones of Hambrusch et al. that among all solutions with minimum total edge length find the ones with minimum width [HT92].

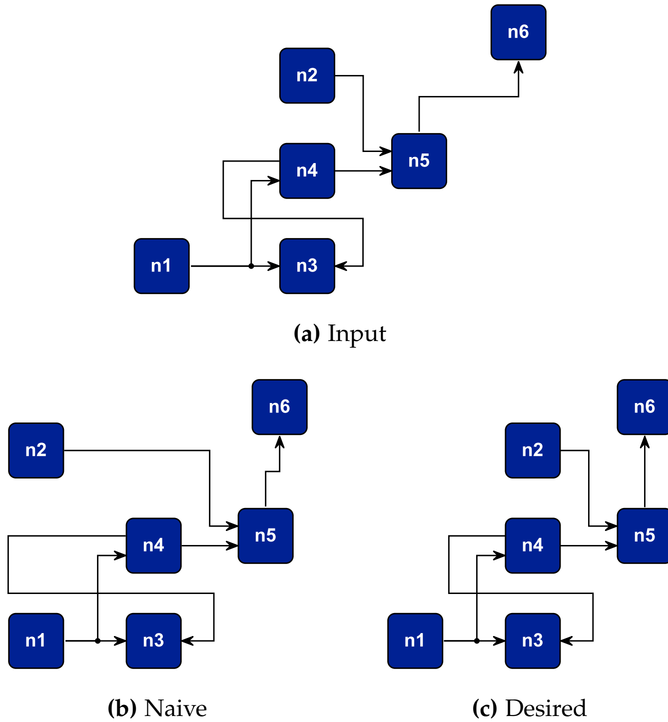


Figure 5.6. Example of applying one-dimensional compaction to the input graph (a). Comparing (b) and (c) illustrates the importance of considering edge length in addition to minimal width (see (n2, n5)) as well as the importance of special handling of certain vertical segments (see (n5, n6) and (n4, n3)).

LR To achieve minimal width with some edge length reduction, compact everything to the left first, using Algorithm 5.2. Then fix the positions of nodes that have no outgoing edge in the original graph, and fix the positions of vertical segments that are not constrained eastwards by a node they are connected to, such as the right vertical segment of edge (n4, n3) in Figure 5.6. Afterwards, execute another compaction pass to the right subject to the just fixed positions. This would yield the desired result as seen in Figure 5.6c

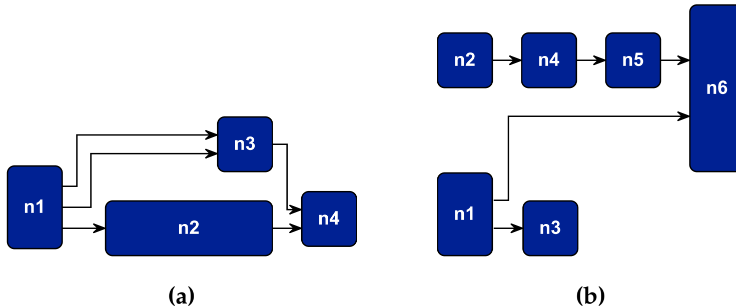


Figure 5.7. Two suboptimal compaction results when considering edge length. In (a) moving n3 to the left would decrease the overall edge length, and in (b) moving the two nodes n1 and n3 to the right would have the same effect.

but may yield sub-optimal results for other diagrams, consider the two examples seen in Figure 5.7, for instance.

EL To minimize the total edge length, one can use Gansner et al.'s NS algorithm for DLP (cf. Chapter 3, p. 57). Note that in this case the compaction is performed by NS, i. e. Algorithm 5.2 is not used.

Before the constraint graph is passed to NS, it is augmented with the edges of the original graph and further auxiliary edges as explained next. Only the original edges should be subject to minimization. Therefore, zero weights are assigned to the edges of the initial constraint graph. Prescribed spacings can be addressed by assigning corresponding minimal edge length values. Unfortunately, it is not always possible to directly add the original edges: consider Figures 5.8a and 5.8b. Adding an edge from n1 to n2 would prevent n1 from being placed below n2 in both cases, since in that case the left-most coordinate of n2 must stay right of the right-most coordinate of n1. To solve this, one can use Gansner et al.'s construction to minimize an edge's length without prescribing the order of the connected node pair (cf. Section 4.1). As a reminder: an auxiliary node is added to the network simplex graph and two edges from it to both of the existing nodes. The minimal lengths of these edges are chosen such that they reflect the offsets

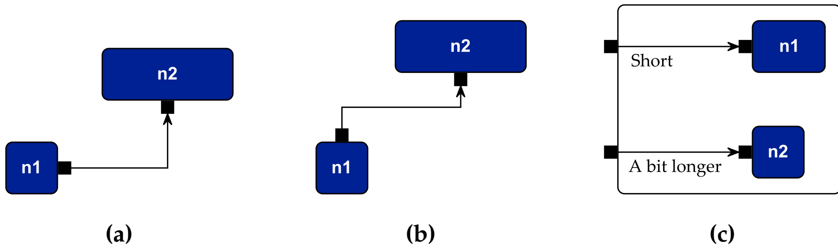


Figure 5.8. When aiming for a compact layout with short edges, several special cases have to be considered. In (a) and (b) node `n1` should be allowed to be placed below `n2`. In (c) the labels of hierarchical ports have different lengths, which results in compaction potential: `n1` can be moved further to the left.

of the edges' attachment points on the nodes' perimeters. This allows edges to become straight that directly connect north and south ports, such as the one seen in Figure 5.8b. The traditional layer-based approach is not able to do this since it must place `n1` and `n2` in different layers. A small drawback of the proposed solution is that it does not necessarily preserve the order of the involved nodes: it can result in a backward pointing edge (`n1`, `n2`) if the structure of the remaining graph encourages placing `n2` to the left of `n1`. In certain use cases this may even be desired, though.

5.2.2 Vertical Segments

Remember that at this point we are only concerned with orthogonally routed edges, which essentially are alternating sequences of horizontal segments and vertical segments. During one-dimensional compaction in the x dimension, only the vertical segments must be considered. As discussed before, the segments can simply be modeled as narrow rectangles. Nevertheless, to obtain acceptable results, care has to be taken in the presence of hyperedges and in the presence of a type of vertical segments that will shortly be defined as *free segments*. For the sake of simplicity, the following explanations and figures assume that edges connect to nodes via ports but refrain from explicitly mentioning or drawing the ports. They further assume that all edges have a single source and a single target.

Hyperedges As mentioned above, hyperedges can be represented by a set of one-to-one edges. In such a case, vertical segments that are part of the same hyperedge may overlap within a drawing. For one thing, this prevents compaction since an overlap-free set of rectangles is required. For another thing, one must ensure that the set of overlapping vertical segments moves synchronously during compaction to not break apart the edge's route. The easiest solution is to simply merge the set of overlapping rectangles into a single rectangle. This can be done in a pre-processing step before constraints are computed. It requires $O(n \log n)$ time, n being the number of vertical segments: Partition the segments based on x coordinates and sort them based on y coordinates. A single execution of a sorting algorithm is enough for this. Afterwards, for each partition, merge the segments that share an extent within the y dimension.

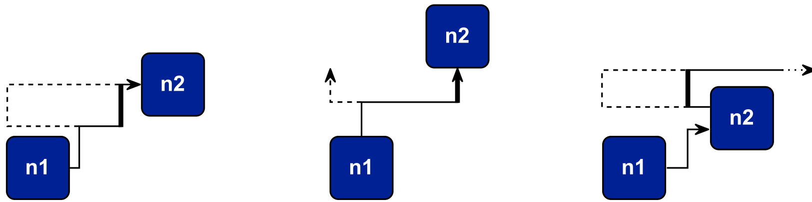
After compaction, only the new x coordinate of a merged vertical segment s has to be transferred to the set of vertical segments s originates from. The y coordinates are not altered during compaction and thus do not have to be transferred back to the input graph.

In the other case, where hyperedges are drawn using a set of overlap-free segments, no special actions must be taken.

Free Segments Informally, a free segment is a vertical segment that has too much leeway. Its position after compaction may result in unnecessarily elongated edges, see Figure 5.9 for three examples. More formally:

Definition 5.2 (Free segment)**.** Let s denote a vertical segment which is represented by a rectangle r_s. Let $e = (u, v)$ be the edge whose route s is part of, and let r_u and r_v denote the rectangles that represent the nodes u and v, respectively. Finally, let S denote the possibly empty set of rectangles representing other vertical segments of e. s is called a *free segment* if its rectangle is not overlapped horizontally by either of the aforementioned rectangles. That is, $\forall r \in S \cup \{r_v, r_u\}$ holds that $r \not\prec r_s$.

Note that free segments may well be constrained by other elements of the diagram. Nevertheless, they are still likely to be moved "too far" to the left (during leftward compaction) resulting in undesired drawings. The three



(a) Touching vertical segments **(b)** North/south ports **(c)** Inverted ports

Figure 5.9. Problems with unconstrained vertical segments (bold). Without special handling, the dashed edge paths would be created. For illustration, the nodes and other vertical segments are not moved.

types of free segments that are illustrated in Figure 5.9 can be handled as described in the next sections. They occur in the context of

(a) "touching" vertical segments,

(b) north/south ports, and

(c) inverted ports.

Touching Vertical Segments The route of an edge can contain multiple vertical segments, some of which can be free segments, see Figure 5.9a. The horizontal order of the vertical segments shall be preserved during compaction for two reasons: (1) to preserve the intention of the layer-based approach to let edges point forward, and (2) to avoid results such as the unnecessarily elongated dashed routes depicted in the figure. Apart from this, it is both allowed and desired that the vertical segments can move as close to each other during compaction as possible. In the best case, they collapse into a single vertical segment, removing a kink from the edge's route.

A possible solution to preserve the order of the segments is to ensure that a constraint is added to the constraint graph whenever a pair of vertical segments originating from the same edge shares a common y coordinate. Remember that this case is not covered by the definition of $\prec$. The necessary

constraints can, for instance, be added manually while transforming the segments of an edge into rectangles. Alternatively, prior to computing the constraints, the rectangles that represent the vertical segments can be enlarged slightly within the y dimension to overlap horizontally. To allow segments to collapse, the compaction algorithm must be aware of whether a pair of vertical segments belongs to the same edge. In such a case it can omit the width of the "left" segment's rectangle when computing the tentative x coordinate of the other segment's rectangle.

One can imagine that a diagram can be compacted to a larger extent if one was allowed to break apart certain vertical segments or to change the order of the vertical segments. Figure 5.9a is an example for the second point: if the dashed edge route is allowed, n2 can be moved above n1, resulting in a maximally compact drawing at the cost of a longer edge path. This is an interesting topic and should possibly be pursued further in the future. Beforehand, however, it should thoroughly be evaluated to what extent users accept the disruption of the otherwise left-to-right flow and whether the methods used here are applicable. For instance, based on the constraint graph used here, it cannot straightforwardly be decided which vertical segments to split (additionally to existing bends), where to split them, and how often to split them.

North/South Ports North/south ports are ports that are located at either the northern border or the southern border of a node. Section 2.6 outlined how to properly incorporate them into the layer-based approach.

A first or last segment of an orthogonally routed edge that is incident to a north/south port is always a free vertical segment. Figure 5.9b shows an example with both a north port (on n1) and a south port (on n2). One-dimensional compaction, as specified above, would allow both vertical segments to detach from the nodes' perimeters. To prevent this, the rectangle r_n that represents a node n and any rectangle r_{vs} that represents a vertical segment incident to a north/south port of n are added to a common group. The offset between the r_n and r_{vs} within the group is set to the vertical segment's attachment point subtracted by the node's x coordinate (assuming a common coordinate system).

This, in conjunction with the explanations of the previous section on

"touching" segments, allows a result that is not possible with the traditional layer-based approach. Edge routes directly connecting a northern port with a southern port can now collapse into a single vertical segment. The effect can be seen for the edge (`n5`, `n6`) in Figure 5.6c. In the traditional algorithm this is not possible because there `n5` and `n6` must be located in different layers.

Inverted Ports Inverted ports are either western ports with outgoing edges or eastern ports with incoming edges. To get a proper drawing, the edges have to be routed around the node they are incident to, and they therefore have to run "against" the flow for some section. This necessarily yields free segments, where it depends on the compaction direction which segments are affected: when compacting to the left, only the edges of western ports are relevant. To be precise, the routes' first vertical segments are free in that case. Accordingly, when compacting to the right, the very last vertical segments of the incoming edges of eastern ports are free. An example of the former can be seen in Figure 5.9c. Just as for touching segments, the free segment should stay close to the corresponding node to keep the overall edge length short.

A possible solution proceeds as follows. A *pull* is assigned to the rectangles that represent the vertical segments when they are initially created. A vertical segment on the western side, such as the one in Figure 5.9c, gets assigned a *right* pull. A vertical segment on the eastern side gets assigned a *left* pull. It then is the responsibility of the used strategy to address edge lengths (cf. Section 5.2.1) to interpret the pull. LR fixes the positions of leftwards pulled vertical segments during its second compaction run, where everything is compacted to the right. EL adds additional auxiliary edges to the network simplex graph that connect leftwards pulled segments and rightwards pulled segments to the corresponding node. The weight of the auxiliary edge can be set to a low value to favor a more compact drawing over short inverted edges, or to a high value to favor the opposite.

In the context of wrapping layered graphs (cf. Section 3.3.2), a use case was mentioned where only the orthogonal edge paths of a drawing should be compacted, or rather optimized, without modifying node positions. Hereto, pulls are assigned to all vertical segments as follows. Following an

edge path from source node s to target node t gives an order of the two nodes and the vertical segments: $s, vs_1, \ldots, vs_n, t$. Let $r_1, \ldots, r_{n+2}$ denote the corresponding rectangles, where r_1 is s's rectangle and r_{n+2} is t's rectangle. For vertical segments that are not free, regardless of the compaction direction, it is usually not important into which direction they are pulled. As long as they keep their relative order, the direction that allows a more compact drawing should be favored. As such, a third type of pull is used: *dc*, which stands for *don't care*. Vertical segments with a dc pull are referred to as dc segments in what follows.

For $2 \leqslant i \leqslant n+1$, assign a pull to vs_{i-1}:

$$pull(vs_{i-1}) = \begin{cases} \text{dc} & \text{if } r_{i-1_x} < r_{i_x} \;\wedge\; r_{i_x} < r_{i+1_x}, \\ \text{left} & \text{if } r_{i-1_x} < r_{i_x} \;\wedge\; r_{i+1_x} < r_{i_x}, \\ \text{right} & \text{if } r_{i-1_x} > r_{i_x} \;\wedge\; r_{i+1_x} > r_{i_x}. \end{cases}$$

With these pulls and fixed node positions, the LR technique is sufficient to improve the edge paths of wrapped graphs as desired. An example was already shown in Figure 3.21 (Chapter 5, p. 119).

Where desired, it is possible to balance the positions of the dc segments: After LR's second compaction pass, which compacts rightwards, the positions of the dc segments are stored. Let them be denoted by vs_x for a dc segment vs. Then, a third compaction pass is executed leftwards again, resulting in new positions vs_x^* for the dc segments. Note that at this point neither the leftwards and rightwards pulled segments alter their position nor do the nodes. Thus, the final position of a dc segment can be set to the mean of its two extremal positions: $\frac{vs_x + vs_x^*}{2}$.

As a final remark of this section, note that simply grouping vertical segments that stem from inverted ports with the corresponding node can result in a cyclic constraint graph. Since the segment does not connect directly to the node's perimeter, scenarios as the one depicted in Figure 5.3 are possible. Additionally, rigidly fixing the relative position between a node and a vertical segment can eliminate compaction potential.

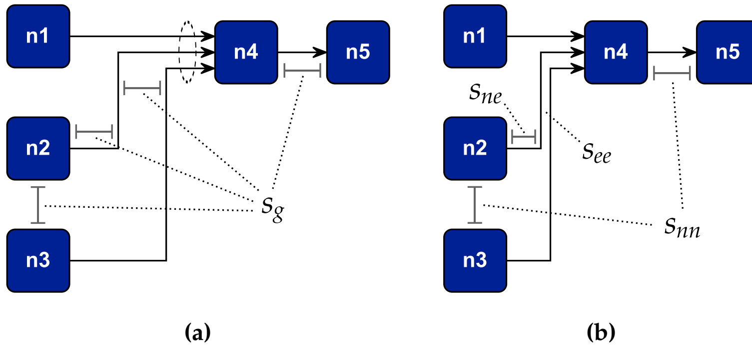


Figure 5.10. Prescribed spacings in diagrams. In (a) a global spacing value s_g must be preserved between elements. In (b) three different spacing values must be considered: s_{nn}, s_{ne}, and s_{ee} between pairs of nodes, pairs of a node and an edge, and pairs of edges, respectively. The dashed circle in (a) marks an area where prescribed spacings must be violated to properly route the edges to the node.

5.2.3 Spacings

Diagram types prescribe spacing values that are to be preserved between elements and that can differ based on element types. Three spacing types are discussed here: (a) a single global spacing value s_g is used to separate any pair of elements from each other, (b) several global spacing values are to be preserved between pairs of elements depending on element types, and (c) every element has an individual spacing value, possibly a different one for each side and possibly a different one in conjunction with every other element. Figure 5.10 illustrates the first two spacing types.

To handle the prescribed spacing values during one-dimensional compaction, the rectangles that represent the elements of a diagram can be enlarged in specific manners. The main difficulty there is to retain a valid set of rectangles. The next paragraphs discuss why this is challenging, even for the simplest type of spacing (a).

Scenario (a): The global spacing s_g can straightforwardly be preserved during compaction by adding $s_g/2$ to each side of the rectangles before the constraint graph is computed without introducing any overlaps. Neverthe-

less, the drawings that should be compacted are not always free of spacing violations as illustrated by the circled area in Figure 5.10a. To properly connect the edges to the common node, the edge paths must come closer to each other than permitted by s_g. And while the depicted horizontal segments are irrelevant during one-dimensional compaction, the problem would be just the same for a set of vertical segments connecting to north/south ports.

Scenario (b): Since spacings are defined between pairs of element types, it is not possible to enlarge the individual rectangles by a fixed value without introducing overlaps. To give an example, let $s_{ne} < s_{nn}/2$. Extending every node-representing rectangle using $s_{nn}/2$ may result in an overlap between a rectangle that represents a node and a rectangle that represents a vertical segment. A solution could be to enlarge rectangles by the largest possible value that does not introduce overlaps, however, this can then result in a compacted diagram in which prescribed spacings are violated.

Scenario (c): As this scenario can be used to model (b), at least the same problems apply as before.

So how can a practical solution look like? Technically, the ELK Layered algorithm provides the facilities to let users specify spacings of type (c). However, to date most implementations of the five steps only properly support spacings of type (b). For this reason and due to the problems explained above, a best effort approach to support the type (b) spacings shown in Figure 5.10b is outlined next. The three spacings are s_{nn}, s_{ne}, and s_{ee}, which prescribe the minimal distance between pairs of nodes, pairs of a node and an edge, and pairs of edges, respectively.

Instead of computing the constraint graph in one pass, it is built incrementally by executing the scanline algorithm multiple times. Note that this may yield more constraints than absolutely necessary. Each time a different subset of the overall set of rectangles is considered and each time the rectangles are enlarged by a different value. Consequently, a rectangle has no unique value by which it is enlarged, which is why the subsequent compaction step operates on the original-sized rectangles. For this to be feasible, an individual "length" c_l is associated with every constraint c, which corresponds to the minimum spacing between the two involved elements and therefore resembles the enlarging of rectangles. Now, when executing the compaction algorithm, the tentative x coordinates of constrained rectan-

gles are computed in a way taking the c_ls into consideration. Further note that during scanline execution constraints are computed looking at overlaps within the y dimension only. Thus, it is enough to enlarge the rectangles within the y dimension.

For the three given spacing values, the scanline algorithm is executed three times: First, only node-representing rectangles are considered and enlarged by $s_{nn}/2$. Second, edge-representing rectangles are enlarged by $s_{ee}/2$ and constraints are generated solely between pairs of edges. These two passes guarantee valid spacings between pairs of nodes and pairs of edge segments. The third pass is executed using all rectangles. This time the selection of the enlarging value is more intricate and three cases must be distinguished based on the value of s_{ne}. Either of the three cases results in a valid input to the scanline algorithm by carefully enlarging rectangles such that no overlaps are introduced. Note that the first two cases are not exclusive. If both conditions hold, either case can be chosen. The third case permits constraints that may result in a compacted diagram with slightly violated prescribed spacings. This is because the rectangles are enlarged by the smallest spacing value in order to avoid overlaps.

$s_{ne} < \frac{s_{nn}}{2}$: Add s_{ne} to all node-representing rectangles.

$s_{ne} < \frac{s_{ee}}{2}$: Add s_{ne} to all edge-representing rectangles.

$s_{ne} > \min\{s_{nn}, s_{ee}\}$: Add $\min\{s_{nn}, s_{ee}\}$ to the node-representing rectangles.

A different way to ensure valid spacings in every scenario is to compute the constraint graph in the naive way mentioned in Section 5.1. While it takes quadratic time and yields a quadratic number of constraints, the pair-wise comparison of elements allows to handle every scenario discussed above. Again, a length has to be associated with every constraint and must be considered by the compaction algorithm.

Violated Spacings As mentioned before, sometimes prescribed spacing values must be violated by the layout algorithm itself, e. g. to bring edges incident to a common side of a node close enough together for them to properly connect to the node. The circled area in Figure 5.10a shows

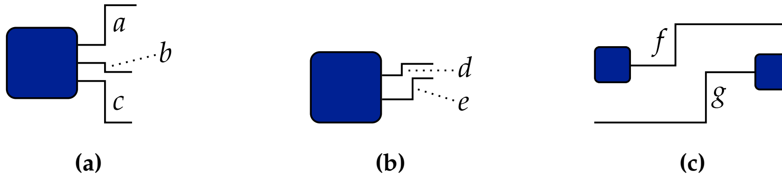


Figure 5.11. Spacing violations are inevitable within the immediate vicinity of a node.

an example. A problem arises in scenarios such as the one depicted in Figure 5.11a: the vertical segments of the three edges incident to the same node share that same x coordinate. Enlarging either of the rectangles that represent the vertical segments a, b, or c would immediately result in vertical overlaps. It is described next how such overlaps can be avoided. The overall set of vertical segments VS can be split into three subsets:

VS_{ns}*:* vertical segments that are incident to north/south ports,

VS_{we}*:* vertical segments that are the first or last segment of edges incident to west and east ports, and

VS_o*:* all other vertical segments.

Now the idea is to *block* the vertical segments of VS_{ns} and VS_{we} in one of three ways: upwards, downwards, or into both directions. The consequence of being blocked is that the corresponding rectangle is not enlarged within the blocked direction during constraint computation. The segments of VS_o do not have to be considered.

Let $vs_{y1} < vs_{y2}$ denote the two endpoints of a vertical segment vs, and let r denote the rectangle of the node vs's edge is incident to. All $vs \in VS_{ne}$ are blocked upwards if they are incident to a south port and are blocked downwards if they are incident to a north port. Additionally, whenever the corresponding node's rectangle is enlarged by a certain value s, all $vs \in VS_{ne}$ must be shrunken by s within the blocked direction. The $vs \in VS_{we}$ are

blocked as follows:

$$block_{we}(vs) = \begin{cases} \text{upwards} & \text{if } vs_{y2} > r_y + r_h, \\ \text{downwards} & \text{if } vs_{y1} < r_y, \\ \text{both} & \text{otherwise.} \end{cases}$$

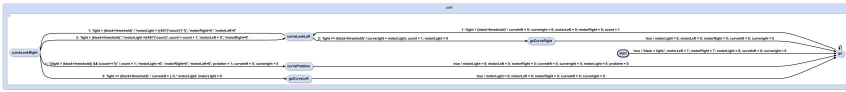
Unfortunately, the blocking is problematic in scenarios as depicted in Figure 5.11b. Both vertical segments d and e are blocked within both directions, and no constraint will be created between the pair of them. As a consequence, the two vertical segments could collapse into a single line during compaction, making it hard to tell apart the two edges. To prevent this, further constraints can be created: The given horizontal order of the vertical segments is the result of an informed decision of the orthogonal edge router. It is chosen such that the number of edge crossings stays low and such that overlapping edge paths are prevented. Therefore, one can simply add constraints between the vertical segments of edges that are incident to the same side of a node that resemble the segments' horizontal order.

Still, this does not solve the problem for two unrelated vertical segments as shown in Figure 5.11c. f is blocked downwards, and g is blocked upwards. It remains open how to solve this.

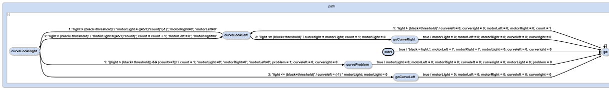
5.3 State Diagrams

The desire to reduce the width of a drawing exists for state diagrams as well. Figure 5.12 shows an example in which first and foremost the diagram's labels, and how they are dealt with during layer-based layout, are responsible for the drawing's significant width.

A crucial difference between dataflow diagrams and state diagrams is that state diagrams' edges are often drawn by using splines. In the previous section, the key idea to handle the orthogonally routed edges of dataflow diagrams during compaction is to use narrow rectangles to represent the vertical segments of the orthogonal edge routes. This is not immediately possible for splines. Therefore, to handle splines, the compaction procedure must either be extended to allow collision-detection between nodes and



(a)



(b)

Figure 5.12. Applying one-dimensional compaction to a layer-based drawing of an SCChart. In (a) dummies for edge labels in combination with the rigid layering result in an unfortunately wide drawing. (b) shows an alternative drawing obtained by applying the compaction method proposed here.

splines and between pairs of splines, which seems like a rather complex undertaking, or the splines' routes must be approximated using rectangles. Depending on the way splines are computed by a layout algorithm, the latter option is viable.

The idea then is to apply one-dimensional compaction to state diagrams in exactly the way as described in the previous section, only that the edges are represented differently: Whenever a section of a spline is parallel to the y axis or curved, the area in which that section would be located is represented by a rectangle. The remaining sections of a spline run parallel to the x axis and can be ignored, just as for orthogonally routed edges. After compaction, a spline's route is simply offset according to the movement of the representing rectangles. For this to work, two things are required:

R-S1: good, overlap-free, and rectangular approximations of the sections of a spline that are to be represented by rectangles, and

R-S2: the ability to transfer the rectangles' new positions to the splines without disturbing the splines' smooth curvature.

It depends on the used type of spline whether this is possible or not. Also, the transformation of spline routes to rectangles is closely connected

to the concrete technical realization of the splines. This thesis is mainly concerned with the ELK Layered algorithm, which computes splines in a fashion that is compatible with the procedure outlined above. The following explanations are therefore based on the way splines are implemented in ELK Layered. Nevertheless, one can incorporate the presented procedure into other algorithms wherever the used techniques allow it.

The convenient feature of ELK Layered's splines is that curved parts are exclusively located in-between layers. Accordingly, whenever the spline passes a layer, it runs parallel to the x axis. This is true both for the case where the spline passes the layer completely from left to right as well as for the case where it passes only a part of the layer until it reaches the node it is incident to. It is thus possible to cover the curved parts with rectangles that are guaranteed to not overlap with the bounding boxes of nodes. Figure 5.13 shows an exemplary spline route covered by two rectangles. The next sections describe the proposed procedure in further detail.

Note that while this way of drawing splines is advantageous for one-dimensional compaction as there are clearly-defined, well-separated areas in which a spline curves, the routing style feels sub-optimal at times. In particular, the rather long sections parallel to the x axis do not really feel "spline-ish". However, this thesis is not concerned with the spline edge routing process itself and thus takes the edge routes as they are.

5.3.1 Splines in ELK Layered

The spline edge routing technique used in ELK Layered was proposed and implemented by Toepffer in his diploma thesis [Toe14]. Its central concepts are to use the dummy nodes that have been introduced for long edges as reference points for the splines and to use techniques known from orthogonal edge routing to order the "vertical" sections in a way that prevents unnecessary edge crossings. *Clamped B-Splines* are used as internal spline representation. They allow an intuitive positioning of the control points and automatically yield smooth curves. Since the spline is clamped, it starts at the first control point and ends in the last control point. A disadvantage is that the spline does not run through any of the inner

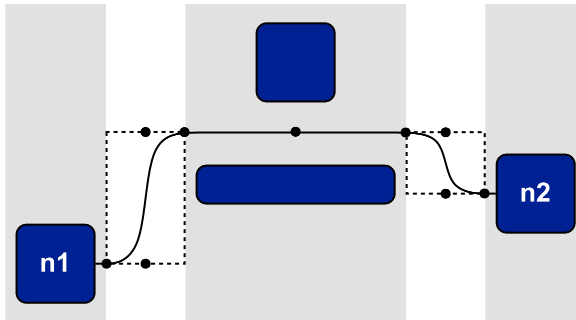


Figure 5.13. Illustration of how ELK Layered routes edges as splines. The small black circles are the control points of the created B-Spline. Layers are outlined as gray boxes. The dashed boxes represent areas in which the spline may not be parallel to the x axis.

control points. Selecting control points in the right way, however, one can ensure that the spline comes very close to a desired point, such as the position of a dummy node [Toe14]. Figure 5.13 shows an example of how the control points are composed for a long edge. For final presentation the B-Splines have to be converted into a representation understood by a rendering framework, many of which use *Bézier splines*. It is easy to transform B-Splines into Bézier splines, e. g. using *Boehm's algorithm* [Boe80].

Remember that during the edge routing step all edges are short because long edges have been split with dummy nodes. Toepffer proceeds as follows to create the overall route for an original edge of the input graph. First, control points are generated *locally*: they are computed for every short edge, where only straight edges and non-straight edges are differentiated. At this point, straight means parallel to the x axis. For the straight edges a single control point is inserted halfway between the source node and the target node. For non-straight edges control points are inserted in the way illustrated in Figure 5.13. The spline route is then finalized during the process of removing long edge dummies: the control points calculated for the individual short edges can simply be joined to form the overall spline.

An important topic in the context of spline edge routing are self loops,

for which ELK Layered computes control points during an intermediate step prior to the edge routing step itself. The space occupied by the self loop is added to corresponding node's margins. As a consequence, self loops can be omitted during one-dimensional compaction since the rectangles created for nodes include the nodes' margins.

5.3.2 Vertical Segments

Thanks to the use of B-Splines and the way the control points are placed, both requirements mentioned above, R-S1 and R-S2, can be fulfilled without much effort.

A single rectangle is created for each curved part of a spline. These curved parts are referred to as vertical segments in the following, and they are equivalent to the vertical segments of orthogonally routed edges apart from having a larger width. Let $p_1, \ldots, p_n$ denote the control points that describe a vertical segment between a pair of layers. The representing rectangle r is then described by:

$$r_x = \min_{1 \leqslant i \leqslant n} p_{i_x}, \qquad r_y = \min_{1 \leqslant i \leqslant n} p_{i_y},$$
$$r_w = \big(\max_{1 \leqslant i \leqslant n} p_{i_x}\big) - r_x, \qquad r_h = \big(\max_{1 \leqslant i \leqslant n} p_{i_y}\big) - r_y.$$

See the dashed boxes in Figure 5.13 for an illustration. It is obvious that vertical segments' rectangles can overlap. And as opposed to the vertical segments of orthogonally routed edges, this time they can overlap in both x dimension and y dimension. To turn them into a valid set of rectangles for one-dimensional compaction, the rectangles can be merged in the same way as explained for orthogonally routed edges (cf. Section 5.2.2). Afterwards, compaction can be performed, and when it comes to adjusting the spline route according to the computed positions, one can proceed as follows. Let r^* be the position of r after compaction. The control points of r are offset by $\delta = r_x - r^*_x$. That is, $p^*_{i_x} = p_{i_x} - \delta$ for all $1 \leqslant i \leqslant n$.

This leaves the control points that have been omitted so far: the ones that are not part of vertical segments. Let $p_s, p_1, \ldots, p_m, p_t$ be a subsequence of such control points. At this, p_s and p_t can be one of the following two cases:

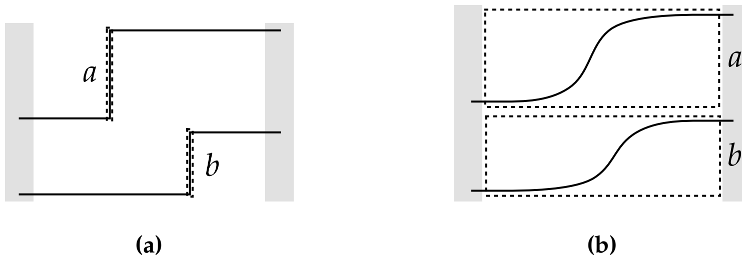


Figure 5.14. The difference between the rectangles (dashed boxes) created for vertical segments that originate from (a) orthogonally routed edges and from (b) spline edges.

- an attachment point on the border of the corresponding node, or
- the first or last control point of a vertical segment.

In either case, the control points have the same y coordinate, and due to the nature of B-Splines, it is sufficient to distribute the inner control points $p_1, \dots, p_m$ equidistantly between p_{s_x} and p_{t_x} in order to roughly preserve the appearance of the spline.

5.3.3 Spacings

Section 5.2.3 discussed how to handle prescribed spacing values by enlarging rectangles. Figure 5.14 illustrates an important difference between orthogonal edge routes and spline edge routes. It depicts a scenario where two edges run between adjacent layers. In (a) the vertical segments cover only a small portion of the width between the two layers. Enlarging the representing rectangles (dashed) does not result in overlaps. In (b), on the other hand, the rectangles span the whole width due to the way the rectangles are constructed for the spline sections. As a consequence, the rectangles cannot safely be enlarged.

One solution would be to merge the rectangles once more after they have been enlarged, and to un-merge the rectangles after the required constraints have been computed. This can, however, result in many rectangles between

a pair of layers being merged, effectively having the same blocking effect as the rigid layering itself. Another solution is to simply refrain from enlarging the rectangles that represent vertical segments, and to accept that some edges may come closer to each other than desired. With splines this is not as bad as with orthogonal edges. The rectangles that represent vertical segments comprise a lot of whitespace that prevents other elements from coming too close to the edges' paths in many cases. If the second solution is pursued, one has to be careful to preserve the horizontal order of the vertical segments: a solution described in Section 5.2.2 relies on the fact that constraints emerge automatically due to vertically overlapping vertical segments. If the rectangles are not enlarged, this is not the case. However, it is easy to determine the required constraints between the vertical segments of a single edge and to simply add them to the constraint graph.

5.3.4 Sloppy Splines

As mentioned above, the way Toepffer selects the control points for the spline routes ensures that no nodes are overlapped but may feel unesthetic at times since they are reminiscent of orthogonal edge routes. This motivated N. B. Wechselberg to explore leaving out a number of control points to increase the splines' curvature.

Unless an edge is straight (parallel to the x axis), Toepffer creates four control points $p_1, \ldots, p_4$ within the area between a pair of layers (cf. Figure 5.13). Wechselberg alters this depending on the type of nodes involved in the way illustrated in Figure 5.15:

Two regular nodes Instead of creating four control points, a single control point is created halfway between the two layers. Its y coordinate depends on the difference between the out- and in-degree of the source and target node, respectively. If the difference is zero, the y coordinate is selected such that it yields a straight edge, otherwise some curvature is added as depicted by the two gray paths in Figure 5.15a.

Regular node and dummy node In the case that one of the two nodes is a dummy node, two of Toepffer's control points are reused: p_3, p_4 if the

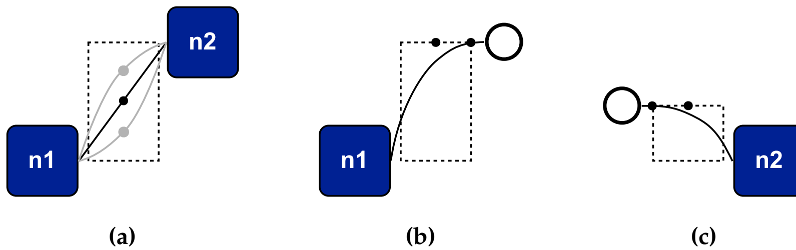


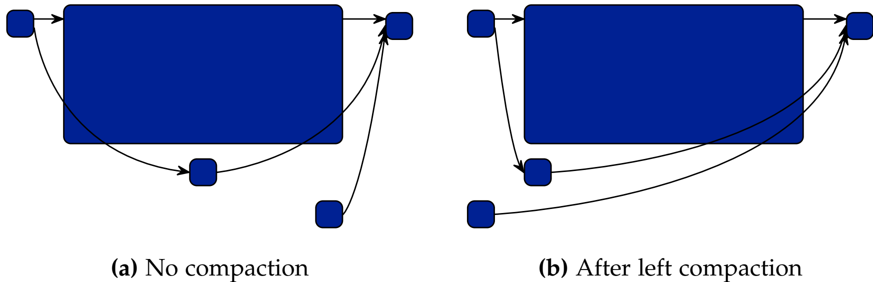
Figure 5.15. Control points for *sloppy* splines as proposed by Wechselberg.

target node is the dummy node (as seen in Figure 5.15b), and p_1, p_2 if the source node is the dummy node (as seen in Figure 5.15c).

Two dummy nodes If both involved nodes are dummy nodes, Wechselberg does not alter the existing behavior, thus either four control points are created or a single control point halfway between the layers if the corresponding edge is straight. Most of the time consecutive dummy nodes are long edge dummies which are preferably placed at the same y coordinate by the previous coordinate assignment step. Therefore, in the majority of the cases a single control point should be added.

From the edge paths seen in Figure 5.15 one can imagine that the resulting spline routes are visually more pleasing than when created with the conservative approach of Toepffer. However, one can also see that the edge paths leave the areas marked by the dashed rectangle. Consequently, the paths may overlap nodes in a final drawing. The more a diagram's nodes vary in size, the likelier it is that overlaps occur, an example can be seen in Figure 5.16.

The proposed compaction procedure can still be used, given that one derives the vertical segments from Toepffer's control points (as sketched in Figure 5.15). Otherwise, the created vertical segments would have no vertical extent and it would not be possible to properly preserve the horizontal order of elements during compaction. Additionally, it should be noted that new overlaps between edge paths and nodes can be introduced due to compaction. However, with sloppy splines these overlaps may already be



(a) No compaction **(b)** After left compaction

Figure 5.16. Sloppy spline routing can result in edges overlapping with nodes. During compaction, edge overlaps may both be removed and introduced.

there, and it is also possible that overlaps are removed due to compaction. To see this, consider Figure 5.16, where in (b) one edge overlap is removed and one edge overlap is introduced by compacting to the left.

5.4 Discussion

The previous sections explained how one-dimensional compaction can be applied to both dataflow diagrams and state diagrams. In Section 6.1.1 the technique will additionally be used to improve the arrangements of disconnected subgraphs in hierarchical graphs.

As an indication for the practicality of the proposed method, consider Table 5.1. It summarizes the results of a series of evaluations based on three sets of diagrams from practice: two sets of dataflow diagrams and one set of state diagrams. The dataflow diagrams are 69 diagrams from the commercial interactive model browsing solution EHANDBOOK[1] and 529 Ptolemy diagrams derived from a subset of the diagrams introduced in Section 1.9. Both diagram types are hierarchical. For this evaluation the hierarchy was removed: sub-diagrams were extracted and evaluated separately. This is feasible since the layout algorithm considers every sub-diagram separately anyway. The state diagrams are 331 SCCharts, which represent student

[1] http://www.etas.com/de/products/ehandbook.php

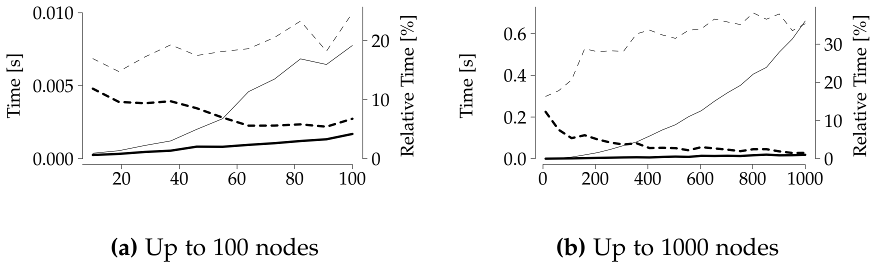
Table 5.1. Results of applying one-dimensional compaction to layer-based drawings. LR stands for left compaction followed by right compaction, and EL stands for compaction aiming for short edges. $\overline{n}$ and $\overline{e}$ denote the average number of nodes and edges. $\overline{w}$ denotes the average width after compaction in percent of the original width, and $\overline{el}$ the average edge length. Standard deviations are given in brackets.

	$\overline{n}$	$\overline{e}$		$\overline{w}$ (%)	$\overline{el}$ (%)
EHANDBOOK	25.4 [15.0]	30.8 [18.3]	LR	83.7 [11.9]	78.2 [15.4]
			EL	85.3 [11.2]	76.6 [16.2]
Ptolemy	15.7 [07.4]	19.6 [11.5]	LR	93.6 [08.2]	88.3 [13.8]
			EL	94.3 [07.4]	87.1 [13.3]
SCCharts	18.9 [14.1]	21.3 [21.2]	LR	91.6 [10.4]	89.0 [10.8]
			EL	91.9 [10.2]	88.4 [11.2]

submissions in the context of practical courses at university (cf. Section 1.9). The splines of the computed layouts are represented by cubic Bézier splines, concatenations of Bézier curves. Each curve is defined by four control points. To measure the splines' lengths, each curve was approximated by twenty points that were connected with straight lines. This time, the hierarchy was not flattened since the individual diagrams would become very small. The stated node counts and edge counts comprise the whole hierarchy, in particular, they include hierarchical nodes.

Results are reported for both compaction strategies mentioned in Section 5.2.1: subsequent left-right compaction with node locking (LR) and total edge length minimization (EL). The average width of the drawings decreased between 16 % and 6 %, the edge lengths decreased between 22 % and 11 %. No significant difference can be observed between the two compaction strategies. Still, since those edges that can obviously be shortened are immediately noticed by users, the strategy that minimizes edge lengths should be favored in most scenarios.

Additionally, it should be noted that the degree of the compaction also depends on the used method for the coordinate assignment step. To give an example, the evaluation above used the approach of Brandes and Köpf (BK) (cf. Section 4.2). Using Gansner et al.'s network simplex approach (cf. Sec-



(a) Up to 100 nodes

(b) Up to 1000 nodes

Figure 5.17. Execution times for the dataflow diagrams and the two compaction strategies plotted against the number of nodes: LR (bold lines) and EL (thin lines). The solid lines show the absolute execution time in seconds (left y axis), and the dashed lines show the relative execution time compared to the overall algorithm (right y axis).

tion 4.2), the width of the SCCharts drawings after applying LR decreases only to 93.3% instead of to the 91.6% seen in Table 5.1. The difference is explained by the fact that the BK approach tends to produce drawings with larger height, providing more space for horizontal compaction.

Concerning execution times, Figure 5.17 shows that both methods finish in well under 10ms for up to 100 nodes, with EL using about a fifth of the overall execution time and LR using about a tenth. For up to 1000 nodes EL's execution time increases significantly, which is expected since the network simplex algorithm is used. Still, it finishes in under 0.6s. Therefore, all setups are fast enough for applications that involve user interaction. The evaluations were performed on an Intel i7 2GHz CPU and 8GB memory laptop using a 64bit JVM.

The decision to handle prescribed spacing values by enlarging the rectangles that represent graph elements turned out to be unfortunate. Many special cases have to be considered, and which rectangles to enlarge at which point in time must carefully be thought through. Even worse, certain spacings cannot be guaranteed. Therefore, it should be examined whether it is possible to consider different types of prescribed spacing values as part of a fast constraint calculation algorithm itself that then replaces the scanline procedure outlined above.

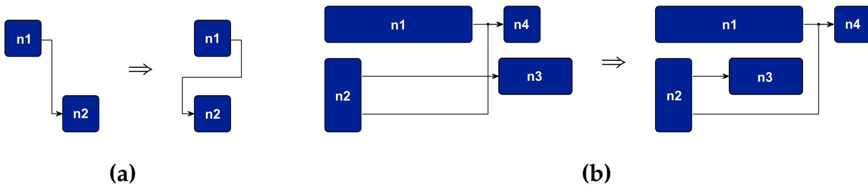


Figure 5.18. Further opportunities to reduce the width of a drawing: (a) splitting a vertical segment and allowing an edge to take a detour, and (b) allowing node n3 to jump over the adjacent vertical segment.

Furthermore, the width of the drawings could be reduced even further if certain assumptions were relaxed. For instance, if edges were allowed to partly run "against" the flow, vertical segments could be split such that the corresponding edge takes a small detour with the effect that the overall diagram becomes narrower, see Figure 5.18a. Chen and Lee discuss this topic in the context of VLSI design, where it is referred to as *jog insertion* [CL97]. Additionally, if the relative order of elements to each other were allowed to change, nodes could for instance jump over vertical segments wherever it saves space. Figure 5.18b shows an example.

Chapter 6

Further Topics of Practical Relevance

This chapter discusses several topics that I believe have not been addressed to their full potential yet, be it in the literature or in practical applications. The selection of topics is mainly based on personal experience, on feedback of colleagues who work with drawings of SCCharts and SCGs on a daily basis, and on requests we get from users of our open-source project ELK. Most of the points that are going to be mentioned have been discussed in the literature before, and the need for improvement often stems, as Spönemann puts it in his doctoral thesis, from both a *discrepancy* between research results and practically accessible implementations [Spö15, p. 1] and from the disregard of the fact that *details matter* for an implementation to be useful in practice [Spö15, Section 7.2]. To some extent, these shortcomings arise from the greatly varying requirements that are placed on layout algorithms by specific use cases, often including algorithmically challenging problems and imprecise definitions of the desired outcome. Furthermore, the tool developers who integrate and fine-tune layout algorithms are often not familiar with the exact domain-specific demands on final drawings; sometimes even the domain experts are not completely certain what the most useful way to lay out a certain diagram is or they do not share the same opinion. As an analogy, consider different coding styles for textual programming languages: some people prefer opening curly braces of code blocks to be placed in a new line, others do not.

Next, every topic is briefly outlined, the first four topics are discussed in further detail afterwards, in the hope of motivating future research.

Hierarchical graphs Many diagram types from practice and their underlying graph representations are hierarchical. Hierarchical graphs can be laid out by (1) specifically designed layout methods that are aware of the hierarchy, or (2) by transforming the problem to laying out a number of simple graphs in a bottom-up fashion, for instance. Both strategies have their strengths and weaknesses and taken by themselves neither addressed every problem that may occur in practice. A challenging problem is how the drawing of a hierarchical graph can be tailored to fit a prescribed drawing area as good as possible.

O-NO avoidance In practice it is often not important to find the drawing that is optimal with regard to a particular metric, but to find a drawing for which it is not obvious to a human how it can be improved. A layout algorithm therefore should avoid arrangements within a drawing that are *obviously not optimal (O-NO)*.

Knowledge-based drawings When humans lay out diagrams manually, they usually base their decisions on some kind of background knowledge. For instance, when arranging nodes that represent physical locations, a human may have the desire to resemble the relative positions of the locations to each other within the drawing. A question is how to realize such behavior with an algorithm while simultaneously fulfilling other relevant aesthetics criteria. Furthermore, in certain use cases a drawing of a diagram already exists, and the task of a layout algorithm may then be to alter the drawing with the goal to "clean it up," while preserving its overall appearance. Various aspects of this have been studied in the literature, however, questions remain especially on the tooling side of things, e. g. how to integrate the methods properly into existing tools and how to let users intuitively guide the layout process.

Layout configuration Layout algorithms usually offer a number of parameters that influence their behavior. While this allows to adjust the algorithm to very specific needs, it is often hard for a user to keep track of the available parameters and how they interact with each other. This leads to the questions how users can be assisted in their task to find the desired parameters and what good default values are that work for the majority of an algorithm's use cases.

Extended graph features In their most basic form, graphs comprise of nodes and edges. A significant number of layout algorithm implementations is limited to these two elements. Many diagram types, however, demand further elements to be considered, the reasonable placement of which is essential to obtain a satisfying drawing. Examples are ports, labels, and hyperedges. While dedicated algorithms exist for certain elements, their results are inadequate at times. An example are edge label placement algorithms that are executed after the actual layout algorithm, with the effect that the required space between a pair of nodes to fit in an edge label without overlapping any node may not be available.

Small graphs vs. large graphs The layout of small graphs, small in terms of node and edge counts, has different requirements than that of larger graphs. Arguably, the drawing of a small graph, which can be perceived at once, must be of higher quality than a drawing of a large graph, for which the flaws are not immediately apparent. Within this context, researchers have, for instance, explored whether different layouts are beneficial when either gaining an overview of a diagram or when trying to understanding the details of a specific part of a diagram [DMS+08] and whether the number of edge crossings is equally important for small and large graphs [KPS14].

6.1 Hierarchical Layout

The layout methods discussed in the previous chapters were concerned with laying out simple graphs. When computing positions for the nodes and edges, they mainly had to consider the adjacency relations between the nodes. The layout of diagrams that are hierarchical is usually more difficult since in addition to the adjacency relations of the underlying graph's nodes, the inclusion relations have to be considered. That is, common drawing conventions require that a node included within another node is to be fully contained within its parent node in a drawing; for the diagrams considered in this thesis this is always a hard requirement.

Examples of hierarchical diagrams have already been seen in the introduction of this thesis, e. g. the Ptolemy diagram in Figure 1.1 (p. 2) and

both diagrams in Figure 1.6 (p. 24). Formally, the underlying graph of a hierarchical diagram has been defined in Section 1.3 as a *hierarchical graph*. It is a graph $G = (V, E)$ that is additionally characterized by an *inclusion tree* $T = (V, I)$ where I is a set of directed *inclusion edges*. The inclusion edges specify which node is to be contained within which other node in a resulting drawing. Figure 1.4 (p. 18) visualizes the concepts of a hierarchical graph.

Several related definitions exist in the literature, at times with subtle differences for identical terms. For instance, Jünger and Mutzel define *compound graphs* in accordance with the definition of hierarchical graphs here [JM04, Sec. 2.6], while Brockenauer and Cornelsen do not require the inclusion information to be a tree [BC01]. The concept of *clustered graphs* can be seen as compound graphs in which the edges may only connect leaves of the inclusion tree [JM04, Sec. 2.6]. Another similar formalism is that of *higraphs*, as given by Harel [Har88].

Thorough overviews of methods to lay out compound graphs, or special cases of them, can be found in the book chapter of Brockenauer and Cornelsen [BC01] and in the diploma thesis of Fuhrmann [Fuh12]. Most of the proposed methods are either based on force-directed ideas [BM99; DGC+05; DGC+09] or modify the traditional layer-based approach [SM91; SM96; San96b; For02; Rai06]. For the special case of clustered graphs, the force-directed idea is again popular [WM96; EH00; HE98; FT04; BAM07; IMM+09], while other strategies have been pursued as well [EF96; BD07].

Apart from the layout methodology itself, there are various strategies to traverse the hierarchical graph's inclusion tree during the layout process:

Top-down Starting at the root node, traverse the inclusion tree in level-order, i. e. breadth-first. This strategy is usually not useful in practice since the dimensions of a hierarchical node are unknown until its children have been arranged.

Bottom-up Traverse the inclusion tree in level-order starting at the deepest level, or alternatively, traverse the tree in post-order. Both traversal strategies ensure that a hierarchical node is only laid out after its children have been laid out.

Mixed The previous two strategies have disadvantages: either the dimensions of inner hierarchical nodes are unknown or the surroundings of hierarchical nodes are unknown. To overcome this, one can traverse the inclusion tree multiple times, e. g. to pass hints to inner hierarchical nodes on how they should be laid out to aid the layout of its surroundings.

Global Process the whole graph at once, e. g. by constructing an auxiliary graph in which the boundaries of hierarchical nodes are represented by dummy nodes.

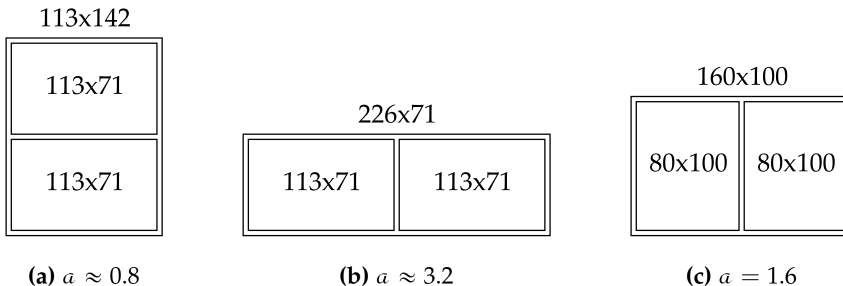
In a survey paper, Sander mentions advantages and disadvantages of the bottom-up and the global strategy that are summarized next [San99]. The bottom-up strategy is easy to implement, allows to configure different layout algorithms for each hierarchical node, and allows to recompute only parts of the layout if an inner node changes: only the direct path from the changed node to the inclusion tree's root. On the downside, hierarchical edges, edges connecting nodes with different parents, cannot be laid out properly. The global strategy, on the other hand, can respect the global situation when positioning nodes and can lay out hierarchical edges. However, it is more complex to implement, it is slower in terms of execution time, it must compute a completely new layout if something changes, and it must use the same layout method for the whole graph.

Thus, when deciding on a strategy to lay out hierarchical graphs, two central questions are (a) how much complexity is acceptable, both in terms of implementation effort and execution time, and (b) whether the graphs to be laid out contain edges that cross hierarchies. In Section 1.3 such hierarchy-crossing edges are defined as *hierarchical edges* and two subtypes are distinguished: *short* and *long*. Already short hierarchical edges require special attention during bottom-up layout. They connect a node to the boundary of its parent node, i. e. the currently processed hierarchical node, or vice versa. At the time the edge route is determined, however, the boundary is neither final nor has it a structural representation within the graph. Klauske et al. describe a procedure to handle short hierarchical edges [KSS+12]. As explained above, handling long hierarchical edges is impossible during bottom-up layout, unless one makes certain arrangements.

For instance, one could transform each hierarchical edge into a set of short hierarchical edges prior to executing a layout algorithm. A drawback of this is that each short hierarchical edge can only be routed locally: it is not clear where the best spot for each short hierarchical edge is to leave its parent's boundary from a global perspective.

The following sections are concerned with the question how to improve layouts in the context of the bottom-up approach. The first topic (Section 6.1.1) pursues similar goals as some of the layering methods presented in Chapter 3, which seek layerings that result in a drawing that best suits a prescribed drawing area. When laying out hierarchical graphs bottom-up, the drawing that results from laying out the simple graph represented by the top-level hierarchical node should fit the prescribed drawing area; it is unclear, however, how the shape of the prescribed drawing areas for the individual hierarchical subgraphs should look like. See Figure 6.1 for an illustration. Therefore, the problem to be solved is to assign a desired drawing area to each hierarchical subgraph such that, taken together, the top-level drawing area is used to its full potential. The second topic (Section 6.1.2) deals with compacting disconnected subgraphs with short hierarchical edges. Assuming that the disconnected subgraphs have been laid out separately, Schulze presented a simple algorithm, called *cell packing* [RSG+16b], to safely position the subgraphs relative to each other such that the short hierarchical edges do not overlap other subgraphs. Here, it is discussed how the resulting drawing can be made more compact.

Before continuing, a short note on the situation in ELK: Where required, the framework lays out hierarchical graphs using the bottom-up strategy, applying a specific layout algorithm to each hierarchical node. Consequently, long hierarchical edges must be omitted during layout. To support these and to generally improve the layout of hierarchical graphs in the context of ELK's layer-based implementation, Fuhrmann implemented a global layout strategy based on an approach by Sander in her diploma thesis [San96b; Fuh12]. However, while working well, the implementation was removed again from the framework after a while since it requires intricate modifications of the otherwise well-separated and flexible implementation of the traditional layer-based approach, conflicting with the clearly defined goal of ELK Layered to keep the code-complexity low and to stay as maintainable



(a) $a \approx 0.8$ **(b)** $a \approx 3.2$ **(c)** $a = 1.6$

Figure 6.1. Illustrating the difficulty of the bottom-up layout strategy to aim at a prescribed drawing area. Assume that the prescribed drawing area has an aspect ratio of 1.6 and that each box in the figure represents a hierarchical node. The two inner hierarchical nodes in (a) and (b) have been laid out with the goal to match a prescribed drawing area with an aspect ratio of 1.6. Say the result is a drawing of 113×71 pixels, thus an area of about $8,000$. It can be seen that neither arrangement of the two nodes comes particularly close to the prescribed drawing area at the top-level (1.6). (c) illustrates that targeting an aspect ratio of 0.8 for the two inner nodes instead allows to perfectly match the prescribed drawing area at the top-level without increasing the occupied area.

as possible [SSH14]. It was later replaced by a strategy that pursues the splitting of long hierarchical edges as outlined above.

6.1.1 Drawing Area Awareness

Diagrams are generally viewed on a medium with a fixed drawing area, e. g. a computer screen or a sheet of paper. As such, a drawing should not only adhere to certain aesthetics criteria, but it should also use the available drawing area to its full potential. Several authors discussed this topic for simple graphs in the context of various layout methodologies, for instance, by adding additional forces to adhere to the drawing area for force-directed approaches [FR91; DH96] and by computing specially tailored layerings for layer-based approaches [HN02a; NRL11]. The matter is different for hierarchical graphs that are laid out in a bottom-up fashion: When the inner-most hierarchical subgraphs are laid out, it is not clear what "local"

drawing area should be targeted at this point in order to use the top-level drawing area to its full potential. Figure 6.1 illustrates the matter.

A possible formalization of the hierarchical problem reads as follows. Let $G = (V, E, I)$ denote a hierarchical graph with inclusion tree $T = (V, I)$, and let $H = \{h \in V : V_{ch}(h) \neq \emptyset\}$ be the set of hierarchical nodes. Further, let $\mathcal{A}$ be an algorithm that applies a layout algorithm to a simple graph represented by a hierarchical node. $\mathcal{A}$ additionally takes a *configuration* that is used to configure the concrete layout algorithm itself and how it should behave. A configuration could read: use an implementation of the layer-based approach and have it lay out the graph left-to-right. The topic of layout configuration is discussed further in Section 6.4.

Let $h_1, \ldots, h_n$ be an enumeration of the hierarchical nodes H in the order as they would be visited by a post-order traversal of the inclusion tree. Let C denote a set of potential configurations, and let $c = (c_1, \ldots, c_n)$ denote a tuple of these configurations where each $c_i \in C$ is the configuration to be used for h_i. A bottom-up layout then is a sequence of layout algorithm invocations:

$$S = \mathcal{A}(h_1, c_1), \ldots, \mathcal{A}(h_n, c_n),$$

and the problem to solve is:

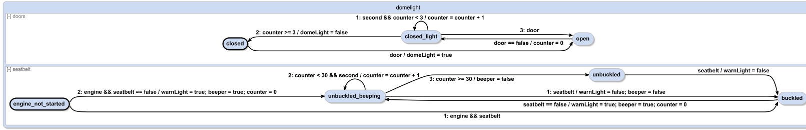
PROBLEM DRAWING AREA-AWARE HIERARCHICAL LAYOUT (DAHLP)

Given a hierarchical graph G and a cost function f, the goal is to find the tuple c that, after executing S, minimizes f measured on the final drawing:

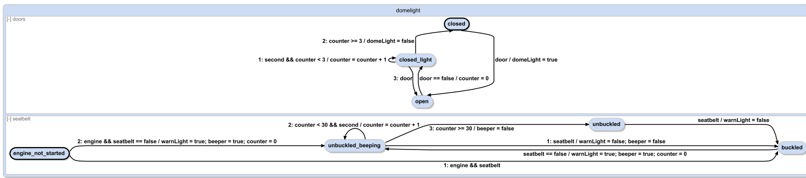
$$\min \; f\big(\mathcal{A}(h_n, c_n)\big).$$

Assuming that every hierarchical node allows the same configurations, there are $|C|^n$ possible configurations of G, which makes it impractical to simply test every configuration. Assuming that a single layout run takes 10ms and that only two different configurations are used, the required time for a hierarchical graph with 20 hierarchical nodes is already $10\text{ms} \cdot 2^{20} = 2.9$ hours.

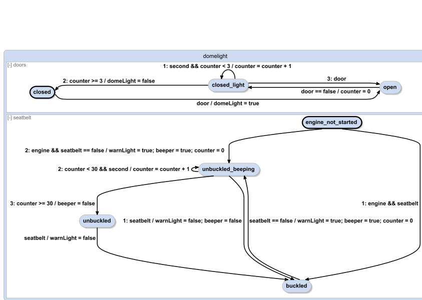
As discussed during the introduction (p. 13), the max scale measure is advantageous when assessing how well a drawing fits a prescribed drawing area. Consequently, its inverse version lends itself for f. Figure 6.2 shows

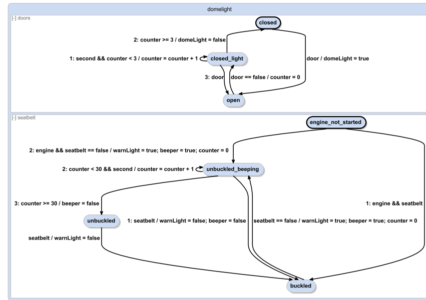


(a) $s = 0.90$



(b) $s = 0.90$





(c) $s = 1.63$ **(d)** $s = 1.37$

Figure 6.2. Four drawings of the same SCChart that differ in the employed layout direction for the two regions. With respect to a reference frame of a standard computer screen, $\mathcal{R}_{(1920,1080)}$, (c) has the highest max scale value s and is thus considered to be the best drawing out of the four.

an example in which two possible configurations are allowed per region of the SCChart: either the layout direction is rightwards or it is downwards. For the given SCChart, the best drawing with respect to max scale, (c), is found when the top region is laid out left-to-right and the bottom region is laid out top-to-bottom. The next section extends this illustration to a small evaluation that emphasizes the necessity to address such problems arising from the hierarchical nature of the graph to obtain satisfying drawings in practice.

Bruteforcing SCCharts

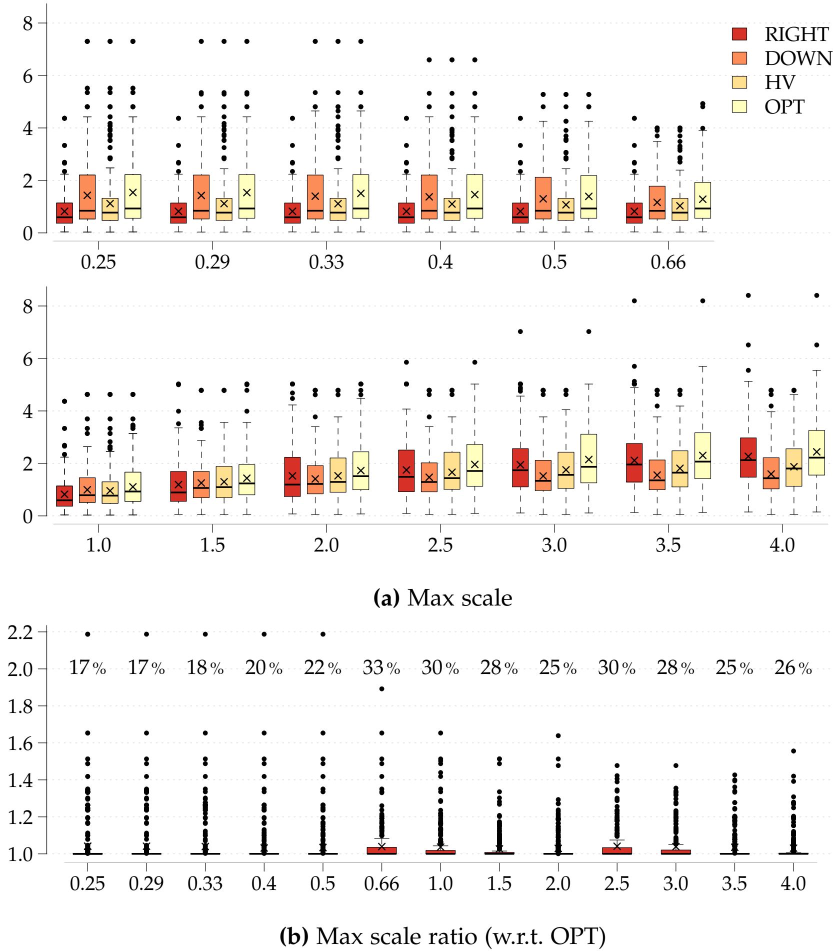
The feasibility of the imagined procedure is examined in the following using the SCCharts as introduced in Section 1.9. However, since computing a layout for every configuration combination is expensive, only those SCCharts were used whose underlying graphs have less than 40 nodes, leaving 289 instances. Remember that SCCharts comprise of alternating hierarchy levels represented by states and regions. To keep matters simple, only the region representing nodes shall allow different configurations and only the layout direction shall be altered. The regions contained in state representing nodes are arranged by a simple packing algorithm with the goal to achieve a certain aspect ratio, which was set to the aspect ratio of the desired top-level drawing area for each state. Clearly, this point offers potential for improvement.

The following four strategies were evaluated. Setting the layout direction of a region refers to setting the layout direction of the hierarchical node that represents the region, and thus the direction with which the simple graph represented by that hierarchical node is to be laid out.

RIGHT Set the layout direction of every region to left-to-right.

DOWN Set the layout direction of every region to top-to-bottom.

HV (horizontal-vertical) Let the levels of the inclusion tree be numbered, where the 0th level contains the root node. The even layers hold regions that contain states to be laid out by a regular layout algorithm. The odd layers in turn hold states that contain regions that are excluded from the



(a) Max scale

(b) Max scale ratio (w.r.t. OPT)

Figure 6.3. Results of laying out SCCharts using different layout directions during bottom-up layout. (a) shows max scale values (y axis) for the four different strategies used and for the thirteen different tested aspect ratios of prescribed drawing areas (x axis). (b) shows the max scale ratio between OPT and the best of the other three. The additional percentage numbers in (b) are the relative graph counts, out of 289, for which none of the other three found a solution as good as OPT.

configuration process for reasons explained above. Now, alternate the layout direction for the regions: set it to left-to-right if a region is in a level l with $l \bmod 4 = 0$ and to top-to-bottom if $l \bmod 4 = 2$.

OPT For the n regions, find the configuration tuple c out of C^n with $C = \{\{\text{direction=left-to-right}\}, \{\text{direction=top-to-bottom}\}\}$ that maximizes the max scale measure.

Each graph was laid out with each of the four just described strategies for thirteen different target drawing areas using ELK Layered as layout algorithm. Since the explicit dimensions of the drawing areas are irrelevant at this point, only their aspect ratio is stated in what follows. The results are summarized in Figure 6.3a. As one would expect, laying out every region downwards comes close to or equal to OPT in many cases when targeting a narrow but high drawing area (0.25–0.66). Accordingly, laying out every region rightwards is advantageous for wide but flat drawing areas (3.5–4.0). The HV strategy performs well for drawing areas with medium aspect ratios (1.5–2.5). However, as shown in Figure 6.3b, the OPT strategy found better drawings in a significant number of cases, between 17 % and 33 % of the diagrams for the tested aspect ratios. The plot in Figure 6.3b shows the max scale ratio between OPT's result and the best result of the other three strategies.

For larger graphs it is to be expected that the gap to optimal results increases, which highlights that new methods are required to compute drawings for hierarchical diagrams that are to be tailored for prescribed drawing areas.

Summary

Compared to a global layout strategy, a bottom-up layout has two significant advantages: different layout algorithms and configurations can be used for each hierarchical node, and the bottom-up layout procedure is easy to implement; even without knowing which concrete layout algorithms are ever going to be executed. However, to create drawings of hierarchical graphs that fit a prescribed drawing area, new methods are required. The relevant points to be addressed in this context are:

Capable basic layout algorithms The algorithms that are used to lay out the individual simple graphs must be able to produce drawings tailored for a specific drawing area.

Guiding traversal algorithm An algorithm must decide what the desired drawing area of each individual hierarchical subgraph is. It may turn out during later layout runs that earlier layouts are disadvantageous, in which case it would make sense to move back and forth within the inclusion tree instead of traversing it in a single post-order pass.

Assess ability to change To decide how the drawing area of certain hierarchical subgraphs should look like, it must be possible to assess what "shapes" a graph's drawing can take on. For instance, the drawing of a path differs significantly when drawn either left-to-right or top-to-bottom, while the direction does not have a significant impact on the shape if the drawings of a graph are "squarish".

6.1.2 Short Hierarchical Edges

When a hierarchical node contains multiple subgraphs that are not connected among each other, see Figure 6.4 for a simple example, the problem arises to place the subgraphs in the plane such that little space is used. Here, it is assumed that each subgraph has been laid out separately beforehand. A possible strategy is to approximate each subgraph by its bounding box and formulate the problem as a rectangle packing problem. However, such problems are often NP-hard [Len90] and rectangles may be poor approximations. Freivalds et al. and Goehlsdorf et al. discuss the relevant related work from a graph drawing perspective and present heuristics for the problem based on a *polyomino* representation, which approximates every subgraph using squares on a grid [FDK02; GKS07].[1] The polyomino approach works well for simple graphs without hierarchical edges. However, Ptolemy diagrams (cf. Section 1.4), for instance, are hierarchical and regularly contain short hierarchical edges that connect different hierarchy levels of the diagram through hierarchical ports. The edge between the `Const` node and its parent

[1] As a side note: the idea of polyominoes was also used by Gansner et al. in the context of *dynamic graph layout* (cf. Section 6.3) aiming at *layout stability* [GHN13].

node ConfigureInputFile in the top right corner of Figure 6.4 is an example of a short hierarchical edge. When placing the subgraphs in the plane, the short hierarchical edges have to be considered specifically. They are not allowed to cross other subgraphs, which cannot be prevented using the previously mentioned methods. Furthermore, subgraphs should be placed such that the overall length of short hierarchical edges is as small as possible. In the area of VLSI design, Lai et al. presented a method based on the *sequence-pair representation* that allows to specify for a module, i. e. a rectangle, that it has to touch one of the four boundaries [LLW+01]. However, this is not sufficient for the previously described problem and, as said, rectangles do not approximate subgraphs well.

The methods discussed in the following are restricted to a scenario in which the side of the hierarchical node a short hierarchical edge connects to is preset. In this setting, Schulze presented the *cell packing* algorithm to compute a placement of subgraphs with short hierarchical edges that is guaranteed to be overlap-free [RSG+16b]. A weakness of the algorithm is that its results are suboptimal in terms of the used area and the total length of the short hierarchical edges. The remainder of this section therefore explains how the placements of the cell packing algorithm can be improved using the one-dimensional compaction technique known from Section 5.1. To better approximate a subgraph, the *rectilinear convex hull* of the subgraph's elements is constructed and is split into a set of rectangles. Both can be done in $O(n \log n)$ time using a scanline method, where n is the number of points used to represent the area covered by a subgraph in the first case, and the number of corners of the rectilinear convex hull in the second case. Before explaining the compaction procedure further, a general formulation of the problem is outlined.

Let $\mathcal{C}$ denote a set of *components*. Each component $c_i \in \mathcal{C}$ is a tuple $c_i = (\mathcal{R}_i, \mathcal{E}_i)$ where $\mathcal{R}_i$ is a non-empty set of *rectangles* and $\mathcal{E}_i$ is a (possibly empty) set of *external extensions*. Rectangles are quadruples (x, y, w, h), with all elements in $\mathbb{R}$. The k-th rectangle of c_i is r_i^k. Assume that all rectangles of the same component touch somewhere alongside their border. The l-th external extension $e_i^l = (d_i^l, \delta_i^l, \epsilon_i^l)$ of a component c_i is a triple of a direction $d_i^l \in \{n, e, s, w\}$, an offset δ_i^l relative to r_i^0, and a width ϵ_i^l. The offset and the width describe an extension clockwise, i. e. for a south extension, the

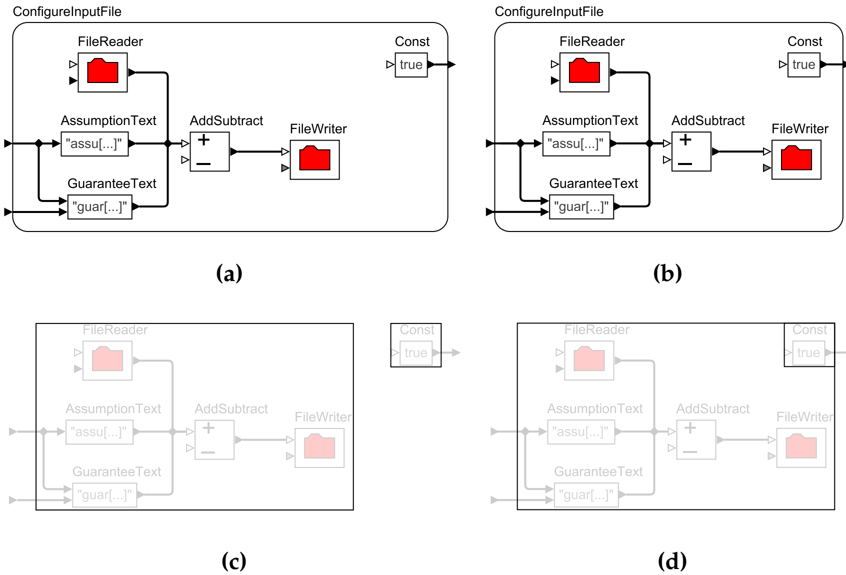


Figure 6.4. Placing the subgraphs of a hierarchical node must assert that no short hierarchical edge crosses another subgraph. In the depicted example, the Const node (top right corner) must not be placed to the left of the other nodes because its short hierarchical edge must connect to the right border of the hierarchical node ConfigureInputFile. (a) shows a feasible placement, and (b) shows the placement optimized with one-dimensional compaction. (c) and (d) show the bounding boxes of the two subgraphs and illustrate how they must overlap to obtain the placement (b).

offset is its right-most point and the width points to the left. Intuitively, it represents a line or a strip attached to the border of a rectangle which extends infinitely into the specified direction. We say an extension (d, δ, ϵ) is *horizontal* if $d \in \{w, e\}$ and *vertical* if $d \in \{n, s\}$. A set of components $\mathcal{C}$ is considered *proper* if no pair of components overlaps and no external extension overlaps a component; horizontal extensions may overlap vertical extensions. The *length* of an external extension is defined as the distance between the extension's attachment point and the intersection point with

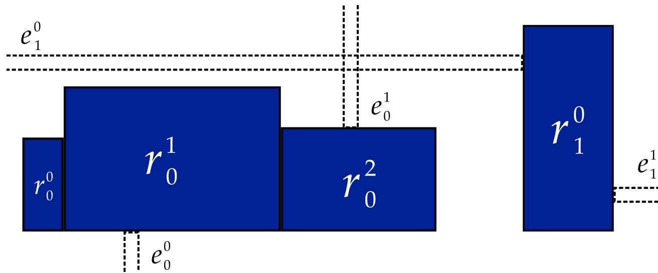


Figure 6.5. The figure shows two components c_0 and c_1. c_0 consists of three rectangles and two external extensions and c_1 consists of a single rectangle and two extensions. The external extensions e_1^0 and e_0^1 are allowed to overlap since one is vertical and the other one is horizontal. They are not, however, allowed to overlap with any of the rectangles.

the bounding box that surrounds all component rectangles. See Figure 6.5 for an illustration.

Problem Compacting Components with External Extensions (CEP)

> Given a set of components $\mathcal{C}$, transform it into a proper set $\mathcal{C}'$ that uses little area and keeps external extensions as short as possible.

In terms of placing the subgraphs of a hierarchical node, a component represents a subgraph's rectilinear convex hull, and each external extension represents a short hierarchical edge.

For the use case of compacting layer-based drawings, as described in Chapter 5, it was sufficient to compact within the x dimension only. This time, however, it is necessary to compact in both dimensions. This is possible by continuously applying one-dimensional compaction in alternating dimensions and directions until no further, or little, progress is made. The compaction procedure for a given set of components $\mathcal{C}$ that has been arranged with the cell packing algorithm, thus proper, then looks as follows. As the set is proper, a grouped constraint graph (cf. Section 5.1, p. 176) can be constructed. Each component is represented by a group and the component's rectangles are added to it. The external extensions are con-

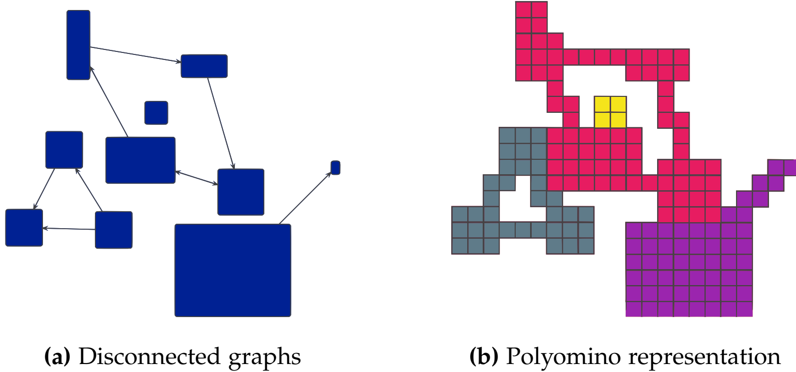
verted into finite rectangles: each external extension is cut at the point where it intersects with the bounding box surrounding all components of $\mathcal{C}$. After each compaction pass these lengths have to be adjusted to prevent components from permuting.

Rectangle representing horizontal and vertical extensions cannot be present at the same time during one-dimensional compaction since in that case the set of rectangles to compact may not be valid: External extensions are allowed to overlap with each other in the context of CEP but the representing rectangles for compaction are not allowed to overlap. Nevertheless, it is important that the horizontal extensions are considered during vertical compaction, to prevent nodes from overlapping with external extensions. And the same is true for vertical extensions during horizontal compaction. The independent application of horizontal and vertical compaction allows to use two different sets of rectangles during compaction based on the current direction: in addition to the $\mathcal{R}_i$ of the components, the rectangles for vertical external extensions are considered during horizontal compaction, and the rectangles for horizontal external extensions are considered during vertical compaction.

As mentioned earlier, the goal is not only to achieve a small area but also to achieve short external extensions. To address this, an additional compaction pass can be executed in each dimension in conjunction with one of the locking strategies discussed in Section 5.2.1 that aim at shorter edges. Alternatively, the network simplex approach, discussed in that section as well, can be used with an artificial source and sink node connected to rectangle representing nodes with corresponding external edges.

Polyomino Compaction

A disadvantage of the cell packing approach and the subsequent one-dimensional compaction is the lack of control regarding the shape of the final drawing, i. e. its aspect ratio. The compacted drawing is often either flat and wide or narrow and tall, depending on which dimension is compacted first. To overcome this, the polyomino approach of Freivalds et al., mentioned above, can be used [FDK02]. By default, it creates drawings with an aspect ratio around 1.0, and the authors describe an easy way to



(a) Disconnected graphs **(b)** Polyomino representation

Figure 6.6. Illustration of the polyomino representation as used by Freivalds et al. to obtain compact placements of disconnected graphs [FDK02]. Figures based on [Cyr17].

explicitly target certain aspect ratios. However, to work for the scenario tackled here, it must be extended to support short hierarchical edges.

As a brief reminder, polyominoes approximate the area occupied by a subgraph using squares, see Figure 6.6. The smaller the squares, the better the approximation. The polyominoes are then placed iteratively on a cellular grid, subject to an objective function. Let (x, y), $x, y \in \mathbb{Z}$, denote the position of a polyomino. Freivalds et al. place a polyomino such that $\max\{|x|, |y|\}$ is minimized. Note that this may allow multiple best positions for a polyomino. Furthermore, three interesting points are to be decided on: (a) the concrete location of a polyomino, (x, y), e. g. the top left corner or the barycenter of filled cells; (b) the order with which the polyominoes are placed, e. g. based on size; and (c) the tie-breaking strategy in case of multiple possible positions.

The practical realizability of the polyomino approach for CEP has been examined by Cyruk in his Master's thesis [Cyr17]. In essence, he associated a *state* with each grid cell: *empty*, *filled*, or *weakly filled*. The last state is used to represent external extensions and the fact that they are allowed to overlap each other but are not allowed to overlap the elements of other components. Additionally, he divides the grid into quadrants, with $(0, 0)$

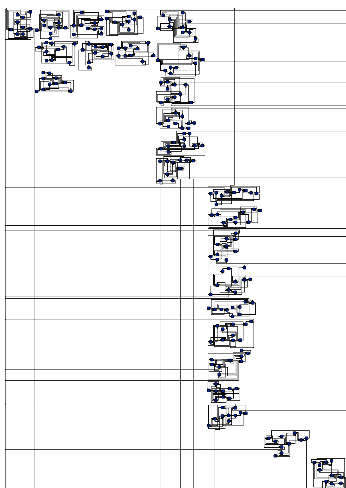
being the origin, and limits the placement of certain polyominoes to certain quadrants, depending on the directions of the external extensions associated with the polyomino. For instance, a polyomino with extensions to the west and north side is only allowed to be placed in the top left quadrant. The idea behind this is to place polyominoes in areas that yield short edges and avoid high numbers of edge crossings.

A comparison of drawings of the same graph created with the cell packing algorithm, with additional one-dimensional compaction, and with the polyomino approach can be seen in Figure 6.7. The figure shows an artificially created hierarchical graph with a single hierarchical node that comprises 25 subgraphs with around 10 nodes each that, taken together, have 34 short hierarchical edges. It can be seen that the one-dimensional compaction technique significantly reduces the area of the cell packing algorithm (cf. (a) and (b)) and also yields the smallest area. The polyomino approach, on the other hand, is able to consider a desired aspect ratio, which is 1.6 in this case, and comes closest to it (1.3). It also yields a smaller number of edge crossings between the short hierarchical edges and shorter short hierarchical edges.

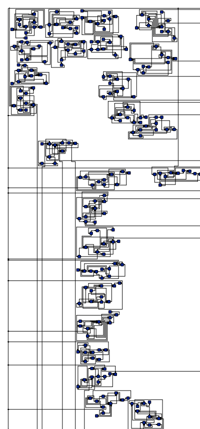
Summary

This section discussed the problem to find positions for disconnected subgraphs of a hierarchical node h that themselves have already been laid out and possibly are connected to h via short hierarchical edges. Two strategies to find such positions were discussed both of which can be enhanced further in the future. Apart from that, the scenario discussed here could be relaxed: instead of assuming a present "direction" for the short hierarchical edges, one could try to identify those directions that allow for the shortest edges and the fewest edge crossings (if port constraints permit). Also, allowing additional bendpoints to be used for the short hierarchical edges would be beneficial for compactness: consider the top left subgraph in Figure 6.7a, which could be moved slightly to the right to reduce the used drawing area, at the cost of two new bendpoints.

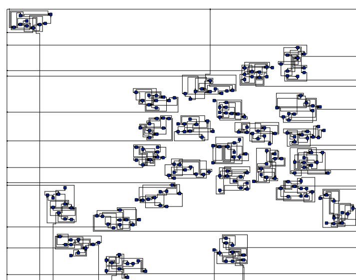
Moreover, considering the surroundings of h may help to avoid unnecessary crossings on other hierarchy levels.



(a) After cell packing
$a \approx 19.1$ Mpx, $ar \approx 0.7$,
$c = 59$, $e \approx 5.1$ kpx



(b) After compaction of (a)
$a \approx 10.8$ Mpx, $ar \approx 0.5$,
$c = 59$, $e \approx 4.6$ kpx



(c) After polyomino-based packing
$a \approx 11.3$ Mpx, $ar \approx 1.3$,
$c = 24$, $e \approx 3.3$ kpx

Figure 6.7. Drawings of the same graph, created with different strategies to place the subgraphs. The same scaling factor has been used for the three drawings. a denotes the required area in pixels, ar the aspect ratio, c the number of crossings between short hierarchical edges, and e the average length of the short hierarchical edges in pixels.

6.2 O-NO Drawing Avoidance

Graph drawers usually approach the task to lay out a graph with a combination of optimization problems whose individual objectives are backed by aesthetics criteria. Despite many efforts to identify most relevant and commonly accepted aesthetics criteria [Pur97; Har98; Pur02; BRS+07; PPP12; MPW+12; KPS14], the subject has not been answered conclusively yet, and it may never be.

Alternatively, one could guide the layout task from the opposite perspective: avoid what feels unaesthetic, or rather, avoid what is *obviously not optimal (O-NO)* to a human viewer. The idea behind the term O-NO is similar to what is known as the *law of least astonishment* [Jam86]. It states "that [a] program should always respond to the user in the way that astonishes him least." Transferred to the area of graph drawing, a user should not be astonished by the way a drawing's elements are arranged and should not feel the urge to manually alter the drawing.

Next, a set of O-NOs is discussed to further exemplify the term. Most of the presented drawings originate from the Ptolemy diagrams (see Section 1.9) and are stripped-down to those parts that are relevant for each illustrated issue. For each drawing three points are discussed: (1) what is considered O-NO, (2) what would be a better solution, and (3) why is it drawn like it is. Since all presented drawings were created using ELK Layered, point (3) will mainly be concerned with explaining limitations of the layer-based approach, or rather its concrete implementation in ELK. It will be seen that often detailed knowledge is required to understand the issues. All drawings except the last one are drawn from left to right, in which case layers are vertical strips.

Further note that the discussed examples are not necessarily to be seen as open problems but as illustrations of what disturbs the perception of a drawing in practice. Many of the illustrated issues have been addressed before or would simply not occur if one had used a different layout method. However, it does not help much if each individual issue can be solved *somehow*; for a user to be satisfied with a drawing, all issues must be absent *simultaneously*.

Edge Path and Compactness

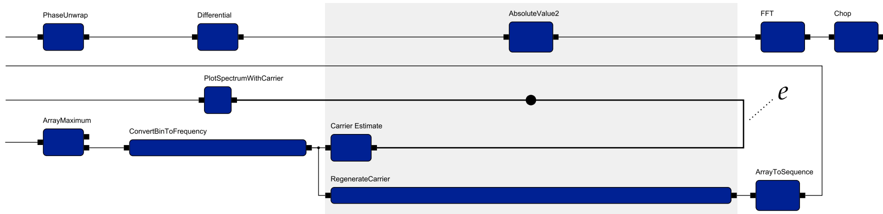


Figure 6.8. O-NO example.

The two issues seen in Figure 6.8 are part of the motivation for the research presented in Chapter 5, and they are also solved by it. For one thing, the edge e is unnecessarily long. For another thing, the two nodes in the top right corner of the drawing are located further to the right than necessary, increasing the drawing's width and elongating connected edges.

The cause for both issues is the wide node at the bottom of the drawing, which results in an equally wide layer l indicated by the gray area. Regarding the first issue: the edge e is conservatively routed to the end of the wide layer. Afterwards, its vertical part is routed in the area in-between l and its right neighbor layer. Finally, the edge runs westwards to the inverted port dummy (small black circle) that is created internally by the layout algorithm. Being conservative is necessary because the area between e's two horizontal segments could contain further nodes that must not be overlapped by the edge path. Regarding the second issue: the two nodes in the top right corner are pushed to the right as the wide layer may, by definition of a layering, contain only a single node of the sequence of nodes at the top of the drawing.

As stated above, both issues are solved by the methods presented in Chapter 5, where a one-dimensional compaction technique is used. However, the first issue is easier to solve: when creating the path between the inverted port dummy and the corresponding node within the same layer, one could explicitly check if other elements lie in-between. If that is not the case, the path could be routed as short as possible, "through" the layer itself.

Edge Path

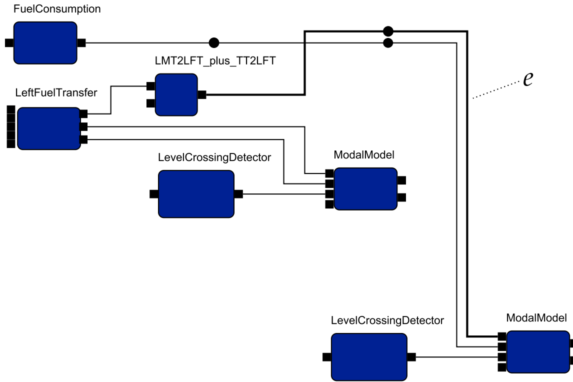


Figure 6.9. O-NO edge path.

In Figure 6.9, the edge e makes a superfluous detour upwards, elongating the edge's path. It is obvious how to improve matters: move the topmost horizontal segment of e's path downwards until it aligns with the leftmost horizontal segment. This yields two fewer bendpoints, an edge path as short as possible, and an unchanged number of edge crossings. The ELK Layered algorithm creates the depicted drawing for the following reason. Prior to the third step, crossing minimization (CM), long edges are split by dummy nodes, indicated by small black circles in the figure. e and the adjacent edge have to cross because the port order is fixed in this example. However, as far as CM is concerned, it does not matter if the crossing occurs before or after the layer l that contains two dummy nodes. In the given drawing, CM placed the crossing before l. The fourth step, coordinate assignment, then decides which parts of the edges should be drawn straight, i. e. x axis parallel. In this case, it favored the adjacent edge over e.

If CM had decided to let the edges cross after l, the desired outcome would have been created. Consequently, to be able to make better decisions, CM would have to know which parts of the edges will be drawn straight afterwards, which in turn depends on the outcome of the CM step itself.

Edge Paths and Edge Crossings

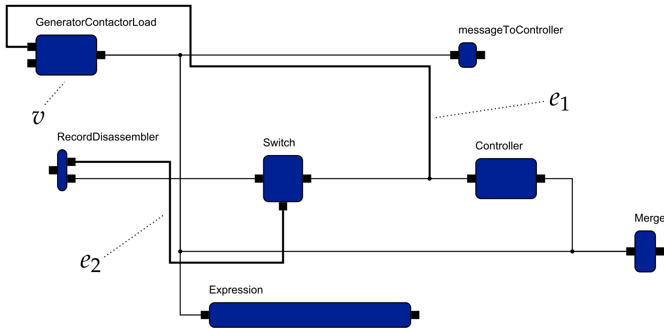


Figure 6.10. O-NO edge paths.

Figure 6.10 contains two obviously improvable edge paths. First, it is possible to improve e_1's path by moving its right horizontal segment upwards to the top of the drawing. This saves two bendpoints and visual clutter. Second, it is possible to improve e_2's path by moving its bottom horizontal segment upwards and its left vertical segment rightwards. This saves an edge crossing and reduces the path's length. Note that the port order must not be changed by convention.

At first sight, the path of e_1 resembles the case that has just been described for Figure 6.9. This is true if one accepts that e_1's path runs along the upper side of the node in the top left corner, v, with the exception that the relocation of the edge crossing does not yield a shorter edge length this time. The reason for the path running along the very top is more intricate. e_1 crosses a hyperedge. Counting the number of crossings during CM is imprecise if hyperedges are involved [San04; EGB06; SSR+14a] since the final number of crossings depends on the concrete edge paths created during the final edge routing step. Consequently, the CM must work with estimates and may think it has to create two crossings with the hyperedge's vertical segment when it routes e_1 around the bottom of v – as opposed to one crossing with the horizontal segment, as seen in the drawing. This explains e_2's path as well: no matter where the crossings occur, the CM step counts three crossings between e_2 and the involved edges.

Hyperedge Paths

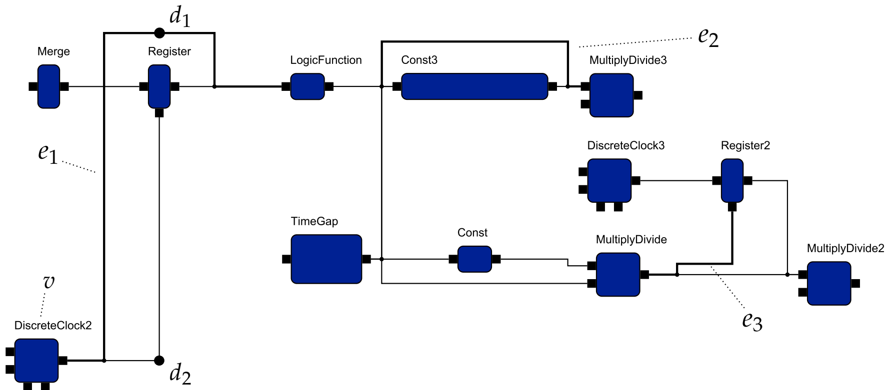


Figure 6.11. O-NO hyperedge paths.

The three marked edge paths in Figure 6.11 are part of hyperedges. There are two points to criticize: the hyperedges' paths split earlier than necessary for e_1 and e_3, and they take unnecessary detours around the nodes they connect to for e_1 and e_2.

ELK Layered works with hyperedges as if they were sets of simple edges [SSR+14b]. This has the advantage that apart from the edge routing step no other step must be aware of hyperedges, which significantly reduces the complexity of step implementations. On the downside, it may yield unfortunate edge paths as seen in the figure. e_1 shall serve as the example to explain the cause, which is the same for all three edge paths. During the layout process, e_1 is split by a dummy node d_1. Additionally, a south port dummy d_2 is introduced for the second simple edge incident to node v in the bottom left. The edge routing step routes edges locally between adjacent layers. Thus, between the first and the second layer it routes the edge paths from v to d_1 and from v to d_2. The vertical edge segment between d_2 and the port it connects to is routed by a separate *intermediate processor* after the actual edge routing step [SSH14]. This clearly separated course of action explains why the edge paths split earlier than necessary.

The superfluous edge length stems from two separate issues in this

example. First, as explained before, the crossing minimization step does not know what a specific node order within a layer implies with respect to the final edge lengths. In particular, it does not know that the individual simple edges could originate from the same hyperedge and thus may share parts of the final hyperedge path, which explains e_2. Second, the reason for d_1 being placed at the top of its layer is due to crossing minimization being a heuristic and not finding the optimal solution in terms of the number of crossings in this case. Placing d_1 at the very bottom of its layer would save a crossing but would still yield excessive edge length.

Edge Crossings, Node Positions

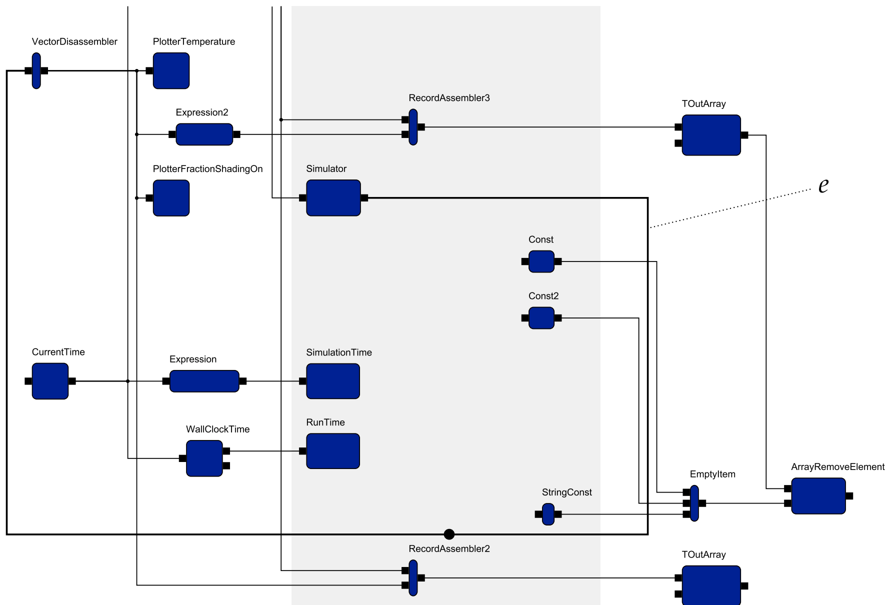


Figure 6.12. O-NO node positions and edge crossings.

Figure 6.12 demonstrates two issues. To understand them, it is important to know that the grayish area represents a single layer. The layer is as wide

as it is because it contains a node as wide as the area that is omitted in the figure.

Now to the issues: For one thing, it stands out that the two small nodes in the center of the drawing can be moved downwards, which would decrease the length of the incident edges. The cause for the node positions is the following. A layer can be seen as a single column with as many rows as it contains nodes. Each row may only hold a single node, which determines the y coordinate. In ELK Layered the horizontal alignment of a node within its row (the x coordinate) depends on the connected edges: nodes without westward edges are aligned rightwards, nodes without eastward edges leftwards. This, together with the extensive width of the layer yields the unfortunate node positions seen in the figure.

For another thing, it is obvious how to improve the path of edge *e*: (a) the rightmost vertical segment of *e*'s path can be moved to the left side of the previously discussed two small nodes, decreasing *e*'s total length and saving three edge crossings; (b) one can go further and move *e*'s bottom horizontal segment upwards to further decrease the edge length, although this introduces an additional edge crossing. Within the context of the layer-based approach (a) is not feasible since edges must not be routed "within" a layer but only in the area between layers. (b), on the other hand, is feasible and can be achieved by moving the inverted port dummy (small black circle) upwards, directly below the node *e* is incident to. It would also be a better solution in terms of edge crossings: three instead of five. The reason for the sub-optimal node order in this example is again the heuristic nature of the crossing minimization. Increasing the number of performed sweeps finds a better solution.

Hierarchy

The final O-NO example, shown in Figure 6.13, originates from an SCG with basic blocks (cf. Section 1.4 and p. 37), is drawn from top to bottom, and illustrates a series of obvious issues. Two of them have already been discussed above: unnecessarily elongated edge paths for inverted ports as seen below node `N52` and above node `N63`, and premature splitting of hyperedges as seen above node `N52` and above node `N61`. Apart from

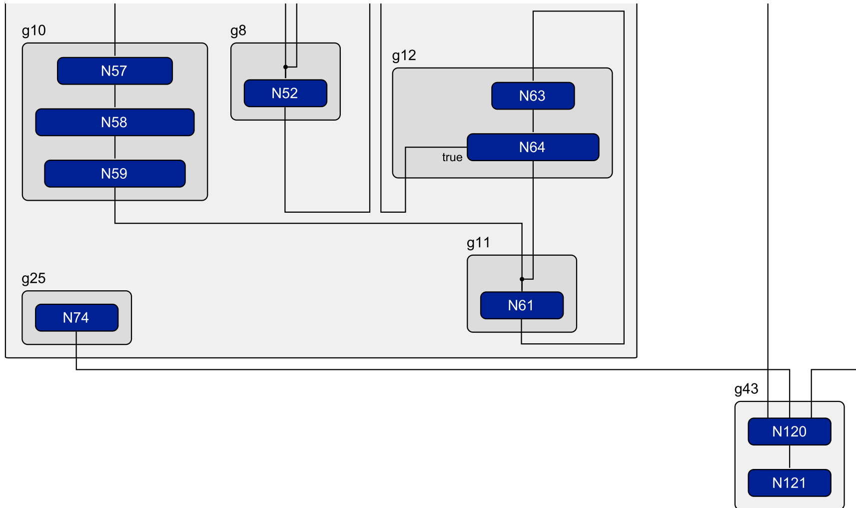


Figure 6.13. O-NO hierarchical.

that, there are three issues that have not been discussed so far. First, the hierarchical node `g25` could be moved further to the right, which would decrease the length of the incident edge. Second, there is an edge crossing in the upper part of the hierarchical node `g43`. It is obvious that the crossing can be removed by swapping the attachment points of the two involved edges on the northern border of node `N120`, without negatively affecting the omitted part of the drawing. Third, the westward edge of `N64` makes an unnecessary detour downwards. Instead, it could leave the hierarchical node `g12` on the west side, reducing the edge's length.

The cause for all three issues is the local nature of the bottom-up strategy used by ELK Layered to lay out hierarchical graphs: it lays out the "inner" graph of a hierarchical node without any knowledge of the surrounding graph. Edges that connect to nodes outside of the current hierarchical node are internally split into multiple edges, and only the edge that connects the hierarchical node's boundary to an inner node is routed. For instance, the three northern edges of `N120` are routed to the northern border of `g43`. This

explains the unnecessary crossing: regardless of how the three northern edges of `N120` are ordered horizontally, no crossing is introduced. Only when it comes to deciding the order of the edges incident to the northern side of `g43` from the outside, the algorithm may notice that the previous decision was unfortunate. Schelten addresses this particular problem in his Master's thesis [Sch16].

The situation is similar for `g25`. When the algorithm assigns a position to `g25`, any horizontal position at the bottom of the surrounding node is fine since it yields the shortest possible length of the southward edge that extends to the boundary of the surrounding node. Only when the algorithm handles the topmost hierarchy level it decides where to position `g43`.

Summary

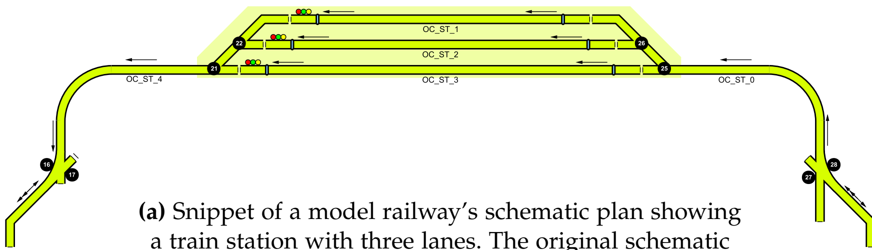
The previous examples illustrated a range of different types of O-NOs. As seen, there are usually reasonable explanations for the flaws of each individual drawing. However, they are far from understandable by users without background knowledge in the area of graph drawing. An interesting question therefore is whether it is possible to formalize and classify O-NOs, potentially based on a specific diagram type at hand, in ways that allow to either come up with new algorithms that eliminate the O-NOs in given drawings or to guide existing layout algorithms to avoid O-NOs in the first place.

This section shall close with the observation that many of the presented O-NO examples are concerned with a flawed orthogonal edge routing. The edge routing of the layer-based approach depends in parts on the placement of dummy nodes during the earlier steps. On the positive side, this ensures that enough space is available to properly route the edges. On the negative side, both the process to place the dummy nodes and the process to route the edges between pairs of layers only work locally and with limited knowledge about their decisions' implications. It may thus be an option to use dedicated orthogonal edge routing techniques after the other elements of a diagram have been placed [WMS10; WMS12; MSW14]. When routing edges with splines, this strategy has been applied before by different authors [GKN+93; NRL08].

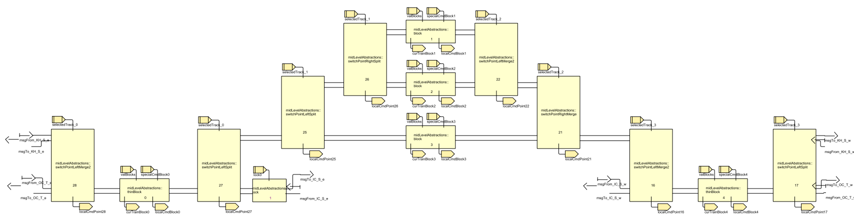
6.3 Knowledge-Based Drawings

Often diagrams are drawn based on some kind of background knowledge, be it manually by a human or automatically by an algorithm. The exact type of additional information used is manifold and depends on the specific use case. Misue et al. distinguish two types of layout processes: *layout creation* and *layout adjustment*, where the first one is concerned with the creation of a drawing and the second one is concerned with altering the positions of the elements within an existing drawing [MEL+95]. The second type is required, for instance, in a scenario where nodes and edges have been added to a graph for which a drawing already exists: existing elements may have to be moved to properly incorporate the new elements into the drawing. This process is also known as drawing *dynamic graphs* [Bra01], for which a variety of dedicated visualization techniques exists that has recently been surveyed by Beck et al. [BBD+17]. During both layout types, layout creation and layout adjustment, additional information can be used in various ways, which is detailed in the reminder of this section.

Layout Creation The process to create a drawing is usually guided by an underlying *intention*. This is already true for the traditional layout methodologies discussed in Section 1.5: the layer-based approach intends to emphasize direction and the force-directed approach intends to distribute nodes evenly and symmetrically. However, often there is a more specific kind of intention when it comes to laying out concrete diagram instances that is based on further knowledge about the concrete instance. For instance, when humans manually create a drawing of a graph, they often try to resemble something they already know. This may be due to various reasons. For one thing, it may help them to come up with a step-wise plan, a strategy, to pursue and to justify their actions. For another thing, they may have the intention to ease the understanding of the created drawing for others. And while the previous statements may seem intuitive, they are speculative. Petre approached the topic how humans arrange and read diagrams scientifically by means of user studies and observing expert users during their daily work [Pet95]. He found that an essential element of a diagram's drawing is *secondary notation*, something that is not explicitly part of the diagram



(a) Snippet of a model railway's schematic plan showing a train station with three lanes. The original schematic was created by Höhrmann [Höh06].



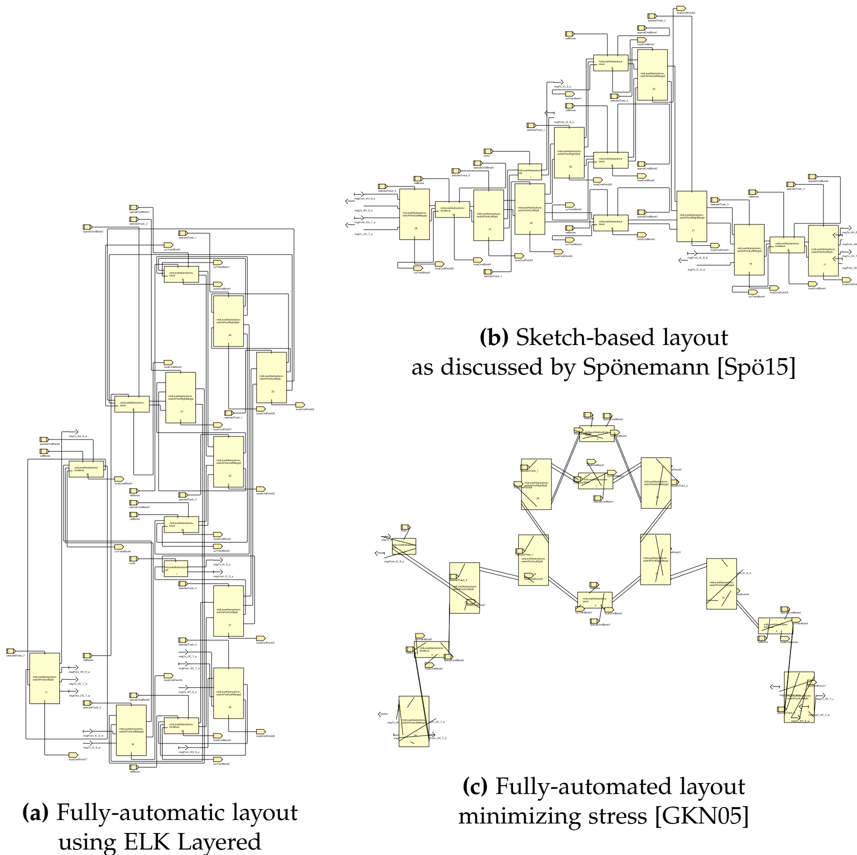
(b) Hand-crafted drawing of a dataflow diagram modeling the controlling unit. The nodes' positions in the drawing resemble the physical topology.

Figure 6.14. Illustration of how knowledge of a system's physical topology can influence the positions of the system's elements in a corresponding drawing.

type's syntax. More precisely, it is "the use of layout and perceptual cues (elements such as adjacency, clustering, white space, labelling, and so on) to clarify information (such as structure, function, or relationship) or to give hints to the reader." As such, secondary notation relies on *experience* and *conventions* and is therefore utilized extensively by experts, as opposed to novices. He further notes that if used incorrectly, secondary notation may be misleading.

To give a concrete example, consider Figure 6.14. It shows a snippet of a schematic representation of a model railway and a dataflow diagram that models the control unit of the depicted part of the overall track, i. e. it models the dataflow between train stations, signals, switches, and further

sensors and actuators. The schematic representation is an abstraction of the physical world in the sense that distances are not true to scale but that the topology is preserved. It is clearly visible, for instance, that a train station with three tracks connected by a set of switches is depicted. The drawing of the dataflow diagram has been created by hand and the drawer chose to resemble the structure of the schematic. This may help viewers of the diagram, who are familiar with either the physical structure of the tracks or the schematic, to immediately find the parts within the drawing of the dataflow diagram they are interested in. Figure 6.15a shows a drawing of the same diagram that has been created with a creational layout algorithm without considering any additional information. As a result, no similarities between the laid out diagram and the schematic can be observed, and the question arises how to capture to desire to resemble something known with a layout algorithm. This has been studied before: In terms of creational graph drawing techniques that use background knowledge, Brandes et al. discussed *sketch-driven layout*, which creates an orthogonal-style drawing based on a *sketch* [BEK+02]. Such a sketch does not necessarily have to be created by a human and can be as simple as merely defining x and y coordinates for a graph's nodes. This work inspired the *interactive layout* discussed by Spönemann [Spö15]. It realizes the first three steps of the layer-based approach with implementations that derive the desired topology of a final drawing, i. e. the horizontal node order (layering) and the vertical node order (order within each layer), from a given sketch and executes the approach's latter two steps to obtain a tidy drawing. Furthermore, the extension of *stress minimizing layout* with *separation constraints* presented by Dwyer et al. allows to flexibly guide the layout algorithm to obey to user-requirements [DKM06a], e. g. to preserve topology. On a meta level, several authors looked at how humans lay out graphs, how human-created drawings compare to automatically created ones, and what aesthetics humans pursue [**HamR08**; DLF+09; PPP12]; others looked at visual structures within drawings that aid memorability [MPW+12]. Recently, Kieffer et al. developed an algorithm, Human-Like Orthogonal Layout (HOLA), which aims at creating drawings that are similar to what a human would create [KDM+16]. They used user studies to identify what makes drawings "human-like", designed their algorithm based on these findings, and double-checked the



(b) Sketch-based layout as discussed by Spönemann [Spö15]

(c) Fully-automated layout minimizing stress [GKN05]

(a) Fully-automatic layout using ELK Layered

Figure 6.15. Three alternative drawings of the diagram seen in Figure 6.14b, this time created by layout algorithms. In (b) the hand-crafted drawing is fed as sketch to the layer-based approach with the interactive extensions of Spönemann. Method-inherent limitations impair the quality of the drawing: the rigid layering prevents connected nodes from being placed horizontally overlapping, something that has extensively been done in the manual layout. In (c) a stress minimizing method is used (cf. Section 1.5.1) that produces a layout structurally similar to the manual one without any further hints. In can be seen that analogously to the manual layout, the small nodes are located close to the larger nodes, and that by aligning certain edges with the x axis, the drawing could become quite similar to the manual one.

results with further user studies.

Using two of the aforementioned methods, the drawings seen in Figures 6.15b and 6.15c are possible whose appearances come closer to that of the hand-crafted one. Note that a drawing very similar to (b) would also be possible with the classic layer-based approach by providing a dedicated cycle removal implementation that reverses the edges of the diagram according to a predefined direction. That is, the larger nodes of the diagram are connected pairwise with two edges. For each node, one of the two edges is an outgoing edge and the other one is an incoming edge (not visible within the drawing). Since therefore each of these pairs of edges forms a cycle, it is hard for the default implementations of the cycle removal step to come up with a reasonable FAS. With the knowledge about the graph's structure just outlined, however, one can ensure that always the same "kind" of edge, say the upper one in the drawing, is reversed.

Two further illustrative examples from practice that incorporate domain-specific knowledge into drawings are:

Metro maps The task to lay out metro maps involves preserving the topology of the individual metro stations' actual geographic locations relative to each other while abandoning true distances between pairs of stations to aid clarity [SR04; HMN04; NW11].

Control-flow diagrams In control-flow diagrams, such as SCCharts and SCGs, it is usually desired to place the nodes that represent the start of the execution in a dedicated place (cf. secondary notation). For instance, if the overall layout direction is left-to-right, it is sensible to place such nodes on the very left.

Layout Adjustment As opposed to layout creation, layout adjustment works with an existing drawing, i. e. at a minimum each node has a position. Depending on the complexity of the underlying diagram, edges may have been routed and labels may have been placed. Note that to some extent the sketch-based approaches mentioned above can be classified as adjusting algorithms. The adjustment task may pursue various objectives, among which *layout stability* and preserving a user's *mental map* [ELM+91] after structural changes are the most discussed topics. Two examples of structural

changes are:

- new elements that have been added to the diagram and must now be incorporated into the drawing, and
- elements that represent hierarchy have been expanded by a user to look at their internals and consequently require more space within the drawing, possibly overlapping other elements.

Informally, the concept of the mental map covers the desire that when users look at a newly laid out diagram they were familiar with beforehand they should immediately recognize the diagram and be able to orient themselves. Formally, mental map preservation is defined as preserving *topology*, *proximity*, and *orthogonal ordering*, and studies have been conducted to determine meaningful metrics to measure the similarity of a pair of drawings [BT98; BT02]. An overview is also given by Branke [Bra01]. However, while the identified metrics give a general feeling of similarity, they may miss domain-specific knowledge and requirements.

Topology-preserving methods have been presented for most of the well-known graph drawing approaches. For instance, Bridgeman et al. addressed orthogonal drawing methods [BFG+97], Dwyer et al. used a stress model and separation constraints for force-based drawing methods [DMW09b], and North and Woodhull extended the layer-based layout method [NW02]. Still, the produced drawings reflect the aesthetics selected by the specific method. Alternatively, Freivalds and Kikusts presented a method that aimed at preserving a drawing's topology by minimizing the distance between a node's new and old position subject to certain ordering constraints [FK01].

Three illustrative applications of layout adjustment in practice are:

Model migration In the area of *model-driven engineering (MDE)* one often differentiates the *concrete syntax* and the *abstract syntax* of a modeling language [KRV07; MMM08]. For graphical modeling languages, i. e. languages that use diagrams as representation, the concrete syntax defines the appearance and dimensions of the diagrams' elements and thus affects the way the diagrams are laid out. Modeling languages, and thus both syntaxes, are subject to modifications over time, which gives rise to the task of *model migration*. For instance, after a language change, the width of a diagram element might have increased, resulting

in overlaps within the drawings. While this problem has been discussed, and existing tools have been compared in the form of contests [RHM+12], Paige et al. note that the migration of the concrete syntax has not been given enough attention yet [PMR16]. A technique targeting LabVIEW diagrams in particular has been discussed recently [RLP+16].

Node overlap removal Force-directed layout methods often have no notion of node dimensions. As such, they create drawings in which nodes overlap, and to create satisfying drawings, one seeks to remove those overlaps in a post-processing step. Several authors tackle the problem using a broad variety of techniques [HIM+98; MST+03; LEN05; DMS06; HLS+07; GH09], often aiming at a trade-off between preserving the mental map and minimizing area.

Interactive diagramming support Even in cases where users have a clear idea of how a drawing should look like and reject fully automatic layout algorithms, handy tool features are usually welcome: preserving manually specified alignments between nodes and re-routing the edges connected to a node that is being manually moved interactively. An editor providing such supporting editing facilities is *Dunnart* [DMW09a].

Summary

Users working with and creating diagrams as part of their job on a daily basis are often hesitant when it comes to fully automatic layout of diagrams [Spö15]. In parts this may be due to the lack of secondary notation when diagrams are laid out automatically. The goal of future research should therefore be to identify the points that cause users' reservations and to provide methods that overcome these reservations. A step into this direction could be to provide a set of supporting tool-mechanisms that are perceived convenient by users. Apart from this, over the years many diagrams have already been laid out without the aid of automatic layout algorithms. Being able to maintain such diagrams, be it in the context of model migration or due to the desire to clean up the drawings for better readability, requires to take into account conscious user decisions, i. e. extract and preserve secondary notation.

Apart from the identification of the problems' sources, further work in at least two areas is required:

Layout methods Highly flexible layout methods are required. As explained, it strongly depends on the use case how a drawing should look like and what domain-specific knowledge may influence the positioning of elements. At this point, the fitting layout methodology must usually be chosen on a per use case basis and often be specifically tailored. Moreover, the methods have technical limitations which are at times inexplicable and unacceptable to users. In the context of the layer-based approach, for instance, no plausible explanation exists as to why it is a "good idea" to forbid connected nodes from being placed one above the other at times in a left-to-right drawing (apart from the formal definition of a layering).

In my eyes, constrained stress minimizing layout (cf. Section 1.5.1) along the lines of the methods presented by Dwyer et al. [DK05; DKM06a; DKM09] is the most promising existing approach to base on here. It starts with a relatively unbiased objective: minimize stress, which merely states that nodes should be somewhat distributed and that those nodes connected by an edge should likely be somewhat closer to each other. By identifying the right set of constraints, one can use it to lay out dataflow diagrams [RKD+14a], a diagram type whose layout conventions are rather different from what stress-based approaches produce naturally. Recently, Wang et al. followed up on these approaches in that they reformulated the traditionally employed *stress majorization* technique [GKN05] to allow a larger variety of constraints [WWS+17] and to allow faster execution, e. g. by parallelization on GPUs.

Tooling From my experience, tools' user interfaces for layout algorithms are often incomprehensible and the performed actions are perceived as intrusive black-boxes. For instance, a common feature is a "layout button" that, upon being pressed, executes a layout algorithm and rearranges the elements of a diagram. If users are unsatisfied with the result, some tools allow to configure the layout algorithm further. To this end, they face the users with a dialog window, presenting vast numbers of configuration options. Understanding these options and identifying

the values that yield the desired effect often requires detailed knowledge of the internal functioning of the used layout algorithm.

Either way, a user should not be locked out of the layout process – something that is often unavoidable due to the nature of creational layout algorithms – but should be able to guide the layout process in an adequate way by providing *hints* on how certain elements should, or could, be positioned. Layout methods then must be able to regard these hints, and the tools must be able to persist them for future layout executions.

Therefore, further questions to be examined in this context are: What kind of layout support helps users with certain tasks at hand? What is an intuitive way for the user to use and configure a layout algorithm? And what is the right level of detail when presenting configuration options? Answering these questions requires to take results of the area of *human computer interaction (HCI)* into account, which is directly concerned with designing intuitive and user-friendly interfaces between humans and computers.

6.4 Layout Method Configuration

To cope with the requirements of different diagram types and use cases, layout algorithms can usually be *configured* extensively using parameters that are referred to as *layout options*, or simply *options*, in what follows.

An example of how much of an impact the configuration of a layout algorithm can have on the quality of the resulting drawing is shown in Figure 6.16. (a) shows a rather chaotic drawing of an SCChart, which is the result of applying ELK Layered in default configuration. (b) shows a drawing of the same graph that makes a more structured and tidy impression, this time laid out with Graphviz's *dot* [GKN02] in default configuration. Just like ELK Layered, dot is an implementation of the layer-based approach. To create the third drawing, (c), ELK Layered was configured to use the same cycle removal strategy as dot and to use a smaller spacing between adjacent layers. The effect is that ELK Layered's drawing now looks similar to dot's and nothing like (a) anymore. This highlights two points: (a) a slight modification of a layout algorithm's configuration can significantly improve (or worsen) the resulting drawing's quality, and (b) can yield dissimilar drawings, for which a human could not immediately tell that they originate from the same graph. Layout options therefore have to be used cautiously and are both a blessing and a curse.

Depending on its purpose, a layout option has a certain scope: it can be set for different graph elements (nodes, edges, etc.), for individual elements of a concrete graph instance (e. g. "node `n12`"), or for the layout algorithm itself. To give a few examples:

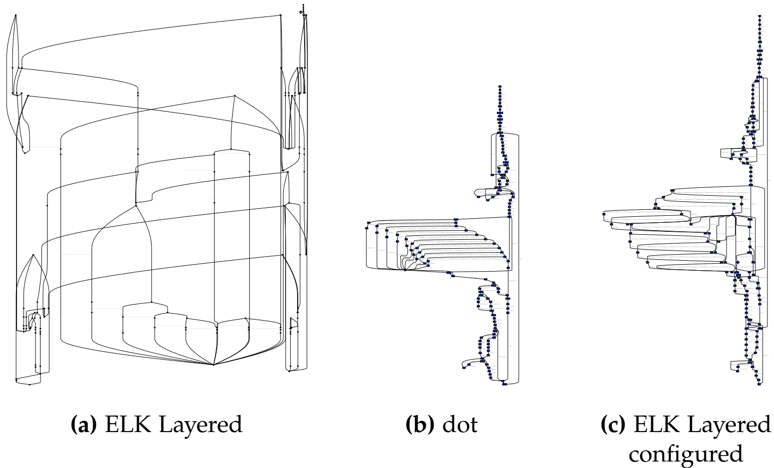
Algorithm: configure the preferred direction into which edges should point.

Hierarchical node: configure the layout algorithm used for the represented simple graph of a concrete hierarchical node.

Simple node: configure the port constraints of a concrete node or the spacing to be preserved between any pair of nodes.

Edge: configure the importance of a specific edge to be short.

Label: configure the desired placement of a diagram's labels relative to its nodes or edges.



(a) ELK Layered **(b)** dot **(c)** ELK Layered configured

Figure 6.16. Different drawings of the same SCChart, illustrating that different configurations of a layout algorithm can lead to significantly different drawings.

Some layout algorithms offer large numbers of layout options. For instance, ELK Layered's documentation page[2] lists over one hundred layout options, and Graphviz's website[3] lists over two hundred *attributes*, although not all of them are supported by each of their layout algorithms and not every attribute influences the layout, e. g. the attribute to set the color of a node. Nevertheless, as soon as the default configuration of a layout algorithm does not produce the desired drawing, someone has to understand the layout options and has to modify them in a way that creates the desired effect. In practical applications, e. g. in the context of tools that build on automatic layout, it is possible to configure layout algorithms on at least two levels: first, specifying a default configuration for a *diagram type* that yields reasonable drawings for the majority of the possible diagrams, and second, fine-tuning such a configuration for a concrete *diagram instance*. The first scenario requires detailed knowledge of the layout approach and of most of the layout options and should therefore be performed by an expert

[2] `http://www.eclipse.org/elk/reference/algorithms/org-eclipse-elk-layered.html` [Acc. 27/09/17]

[3] `http://www.graphviz.org/content/attrs` [Acc. 27/09/17]

or a tool-smith. The second scenario is usually performed by users that work with a tool and want to slightly alter the appearance of a drawing, for instance, to improve it for presentation purposes or because they know that an alternative arrangement of a subset of elements would aid the drawing's comprehensibility for other users. However, not every option can easily be understood and configured by a user, which is why it may make sense to expose only a subset of the available options to a users.

An orthogonal question is how to come up with "good" configurations in either of the two mentioned scenarios. Since the number of possible combinations of layout options is large, it is worthwhile to consider supportive algorithms. Spönemann et al. call the process to come up with a reasonable configuration *meta layout* [SDH14]. They define an *abstract layout* of a graph to be a description of which layout algorithm to apply and which values to use for its supported layout options. Together with a concrete graph instance, the abstract layout yields a *concrete layout*, a different term for a drawing. To identify an abstract layout that has the desired effects, they use an evolutionary optimization technique that rates concrete layouts based on commonly used aesthetics criteria [BRS+07].

As seen, when it comes to configuring layout algorithms to yield the desired outcome, there is research potential in at least two areas:

User interfaces How should layout options be exposed to users, what are intuitive ways to manipulate them, and which layout options should better be hidden? When users are allowed to alter or influence the layout, how can such decisions be persisted for and integrated into future layout executions?

Automatic layout configuration What are good strategies to identify layout configurations that yield the best results not only for a single graph but for a class of graphs? How can the user be integrated into this process?

Chapter 7

Conclusions

This thesis presents several modifications and extensions to the layer-based approach in order to improve and broaden its usefulness in practice, and at this it lays a specific focus on dataflow diagrams. Next, the main contributions and gained insights are summarized, afterwards, the thesis closes with ideas for future research.

Layer Assignments The traditional formulation of a layering has two crucial inherent limitations:

- ▻ The requirement that all edges have to point "forward" significantly limits the solution space of possible layerings, in particular, for certain graph instances it prevents compact drawings and drawings that suite a prescribed drawing area.
- ▻ The width of a layer is dictated by its widest node, possibly resulting in large amounts of whitespace to the left and right side of narrower nodes within the same layer.

The first limitation is addressed with the introduction of the *Generalized Layering Problem (GLP)* (Section 3.2) and the introduction of a *wrapping procedure* (Section 3.3). GLP is allowed to reverse certain edges if it aids the creation of a layering that yields a more compact drawing. An integer program and a heuristic are presented for the new problem and evaluated extensively. For the used set of graphs, GLP is able to create layerings that, when compared to traditional layerings, are more compact, and it is able to control the aspect ratio of final drawings to a certain extent. For use cases where the directionality is not that important, i. e. an arbitrary number of

edges may be reversed, the compactness can be improved even further as demonstrated using a variant of GLP that directly optimizes the max scale measure. Where directionality is of greater interest the proposed wrapping procedure is able to tailor layerings to prescribed drawing areas while resembling traditional layerings in large parts. It does so by splitting a traditional layering into chunks at reasonable points and by placing the chunks side by side. Evaluations based on SCGs show that the proposed procedure is successful. The last contribution in the area of layerings addresses the problem of finding layerings with a restricted number of nodes per layer. Existing work did not consider actual node dimensions, a shortcoming that is overcome here (Section 3.1).

The second limitation mentioned above is addressed in a section below.

Coordinate Assignments The layout of dataflow diagrams poses challenges on the coordinate assignment algorithms:

- ▻ Edges may not connect to the nodes itself but to ports that are located on the nodes' boundaries. The ports may be free to move alongside the boundaries, and node sizes may be free to change.
- ▻ The coordinate assignment step determines whether an edge can be drawn parallel to the x axis, simply referred to as *straight* below, in a left-to-right drawing. This is particularly relevant when edges are to be routed orthogonally since any edge that is not straight requires at least two edge bendpoints.

To address these challenges, two established methods are extended and modified in ways that allow them to be flexibly adjusted to the needs of a broader range of diagram types. First, the method of Gansner et al. [GKN+93] has been modified to optionally allow flexible port positions on a node's border and to allow a node to resize in height, both if it helps increasing the total number of straight edges (Section 4.1). Furthermore, it has been shown how certain edge paths can be post-processed, again to increase the number of straight edges. Second, the method of Brandes and Köpf [BK02] has been extended to support different node sizes as well as ports (Section 4.2). Additionally, modifications to further increase the number of straight edges have been presented, potentially at the cost of additional drawing height.

Drawing Compaction As mentioned above, drawings created with the layer-based approach often suffer from significant amounts of whitespace that stem from nodes with different widths being placed in common layers. To improve matters, it is explained how the simple technique of *one-dimensional compaction* can be applied (Chapter 5). While the technique is well-known and widely used to compact electrical circuit boards in the area of VLSI, for instance, various peculiarities have to be considered when compacting drawings of diagrams and in particular drawings of dataflow diagrams. An evaluation based on diagrams from practice shows that the width of the corresponding drawings can often be reduced significantly by applying this strategy.

Hierarchical Layout Laying out hierarchical graphs is a challenging task. To cope with complexity, one can apply a bottom-up strategy that lays out a sequence of simple graphs that, taken together, result in a drawing of the whole graph. The simplification comes at the cost that each invocation of a layout algorithm to lay out a simple graph has to work with limited knowledge of the subgraph's surroundings, possibly making decisions that turn out to be unfortunate in the final drawing. Instead of resorting to a global layout strategy, however, it is often possible to improve on certain subproblems while keeping the modifications of the bottom-up layout strategy to a minimum.

Two subproblems were discussed here: computing layouts of hierarchical graphs that fit prescribed drawing areas (Section 6.1.1) and packing disconnected subgraphs of a hierarchical node that comprise short hierarchical edges (Section 6.1.2). For the first problem a methodology is proposed and outlined that assigns a "local" target drawing area to each hierarchical node such that the combined drawing matches the prescribed drawing area. An initial evaluation based on a small set of SCCharts shows promising results. For the second problem two strategies are briefly discussed that are based on one-dimensional compaction and *polyomino packing*, respectively.

7.1 Lessons Learned

During my thesis I faced two recurring difficulties that are briefly outlined next.

Objectives Must be Clear A central question worked on in the context of graph drawing that has not been answered conclusively yet is: What do humans like and dislike about the way elements are arranged in certain drawings? The aesthetics criteria pursued by layout algorithms are often defined by graph drawers, not by domain experts who actually work with the created diagrams. To find reliable answers, however, domain experts and layout algorithm engineers should closely work together.

Nevertheless, many efforts are being made to verify or refute the importance of certain aesthetics criteria, often in form of user studies. Unfortunately, such user studies are difficult to conduct both due to the lack of expert participants and the relatively simple questions that can be looked at. Furthermore, it is not clear if a particular aesthetics criterion that appears visually pleasing to a human or aids perception actually improves the performance with respect to a certain work task. Similarly, pursuing specific aesthetics criteria may be advantageous for certain use cases and simultaneously be disadvantageous for other use cases.

Results Must be Explainable It is often hard for humans to comprehend and accept certain decisions of a layout algorithm because they see an easy way to improve a drawing according to their subjective perception. At times, the unsatisfying results are owed to methodological limitations, at other times they could be improved but the ways a user can interact with a layout algorithm to tell it to produce the desired result are often not intuitive and require unjustifiable knowledge acquisition about the algorithm's internals from the user. The methodological limitations often stem from splitting the overall layout task into manageable subproblems. Each subproblem is then solved as good as possible, which may have negative consequences for subsequent steps. To give an example that is discussed in this thesis: The cycle removal step of the layer-based approach dictates what is achievable during the layer assignment step (cf. Chapter 3), and seeking for minimum

number of reversed edges may eliminate those layerings from the feasible set of solutions that would yield desired results. One could therefore say that striving for optimality for subproblems is too much and too little at the same time. Subproblems should be solved while taking their context and their effects into account. An alternative way of thinking about the overall layout problem has been proposed in Section 6.2: Instead of seeking optimal solutions with respect to quantifiable aspects, avoid what subjectively feels not optimal to a human.

7.2 Future Work

During the course of this thesis, several points emerged that are worthwhile to pursue further; some concern enhancements of methods presented in this thesis, others concern whole areas of research that involve many questions yet to be answered. Chapter 6 already covers several topics that I believe are relevant for practice and still offer many open questions, in particular:

- laying out hierarchical graphs,
- avoiding obviously not optimal (O-NO) drawings, and
- allowing a user to get involved with flexible layout methods.

Another interesting question is whether techniques from the recently revitalized area of *machine learning*, in particular *deep learning* and *neural networks*, can be used to create drawings of diagrams. A starting point could be to let a network resemble the behavior of a traditional layout method and afterwards train it to avoid O-NOs.

In addition to these topics, there are concrete points in which the methods presented in this thesis can be improved further.

Layer Assignments Computing alternative layer assignments can be improved in two ways. On the one hand, improved heuristics for GLP could be designed that are not only superior in terms of the resulting numbers of dummy nodes and reversed edges but are also more flexible and able to address graph classes such as paths and trees. Besides, the impact of alternative cycle removal algorithms could be examined. The effect of pursuing a

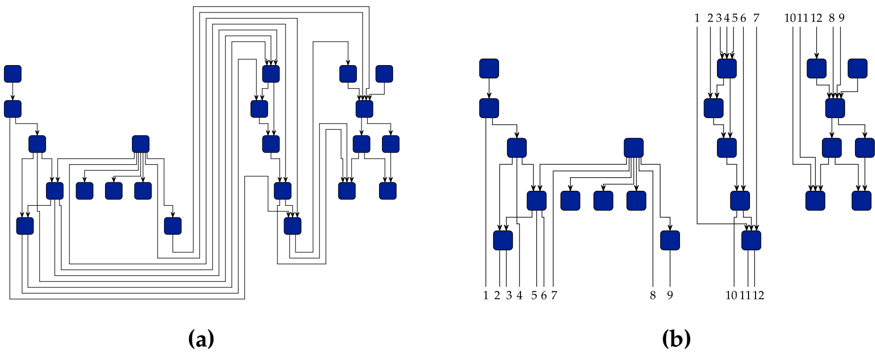


Figure 7.1. A wrapped graph (a) and an alternative presentation (b) in which only parts of the cut edges are drawn; labelling numbers are used to indicate connections.

slightly different strategy to eliminate cycles has already been demonstrated in Figure 6.16 (p. 248). On the other hand, alternative presentation styles could be explored in the context of the presented wrapping procedure that avoid cluttering the drawing with excessively long edge paths, see Figure 7.1 for an example.

The traditional notion of a layering disregards node dimensions, which is one of the main reasons for poor compactness of drawings in practice, where the dimensions of the nodes can vary significantly. And while GLP already alters the notion of a layering by allowing backward edges, it still ignores node dimensions. It may be worthwhile to explore alternative layering formulations that explicitly incorporate node dimensions, ideally not affecting the remaining steps. This would also address the problem that one has to work with estimates of the width and height of the final drawing during the layer assignment step, which can deviate significantly and may be misleading at times [RES+16a].

Coordinate Assignments The two discussed methods for coordinate assignment focus on supporting ports and on drawing a preferably large number of edges straight to cater for orthogonally routed edges. In terms of balance, the computed positions leave room for improvement. It should

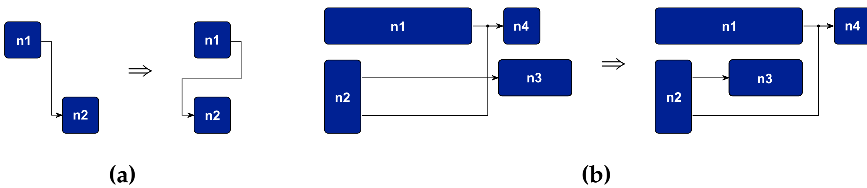


Figure 7.2. Two possibilities to reduce the width of a drawing: (a) allowing an edge to take a detour, and (b) allowing a node to jump over the adjacent edge.

be explored if it is possible to apply a simple post-processing procedure to a given coordinate assignment that preserves the deliberate decisions of the previous steps, e. g. which edges are to be drawn straight, while seeking for the most balanced positions possible. Moreover, drawing many edges straight can have a negative side-effect: it can excessively increase the drawing's area. One could identify edges that would have this effect and intentionally ignore them.

Drawing Compaction Diagram types may prescribe various different spacings to be preserved between diagram elements (cf. Section 5.2.3). During the proposed compaction procedure they are handled by enlarging the rectangles that represent the diagram's elements. This decision turned out to be unfortunate: many special cases have to be considered, and which rectangles to enlarge at which point in time must carefully be thought through. Even worse, certain spacings cannot be guaranteed anymore. It should therefore be examined whether it is possible to consider prescribed spacings during a fast constraint calculation algorithm, assigning an individual minimum length to the separating constraints.

Furthermore, drawings could be compacted even further if certain assumptions were relaxed. For instance, if edge segments were allowed to be split, edges could take small detours with the effect that the overall diagram becomes smaller, see Figure 7.2a. Additionally, if the relative order of elements to each other were allowed to change, nodes could for instance jump over vertical segments wherever it decreases the overall diagram area. Figure 7.2b shows an example. In both cases it would have to be evaluated to which extent users accept such modifications.

Glossary

Acronyms

CPU Central processing unit

DSL Domain-specific language

DSVL Domain-specific visual language

ELK Eclipse Layout Kernel
`http://www.eclipse.org/elk`

ETAS Engineering Tools, Application and Services
`http://www.etas.com/`

FAS Feedback arc set

GPU Graphics processing unit

HALS Hierarchy-Aware Layer Sweep

HCI Human computer interaction

HOLA Human-Like Orthogonal Layout

IBM International Business Machines
`http://www.ibm.com/`

ILP Integer linear program

KCSS Kiel Computer Science Series
`http://www.informatik.uni-kiel.de/kcss`

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client
`http://www.informatik.uni-kiel.de/rtsys/kieler/`

LabVIEW Laboratory Virtual Instrumentation Engineering Workbench
http://www.ni.com/labview

LAP Linear arrangement problem

LP Linear program

MDE Model-driven engineering

MDS Multi-dimensional scaling

OGDF Open Graph Drawing Framework
http://www.ogdf.net/

O-NO Obviously not optimal

SCADE Safety Critical Application Development Environment

SCChart Sequentially Constructive Chart
http://www.sccharts.com/

SCG Sequentially Constructive Graph

VLSI Very large scale integration

Problems

CEP Compacting Components with External Extensions

DAHLP Drawing Area-Aware Hierarchical Layout

DLP Directed Layering

GLP Generalized Layering

LRLP Layering With a Restricted Number of Nodes per Layer

WLGP Wrapping Layered Graphs

Bibliography

[AGR14] Mohamed Almory, John Grundy, and Ulf Rüegg. "HorusCML: context-aware domain specific visual languages designer". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*. 2014, pp. 133–136.

[AHN07] Radoslav Andreev, Patrick Healy, and Nikola S. Nikolov. "Applying ant colony optimization metaheuristic to the DAG layering problem". In: *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS'07)*. 2007, pp. 1–9. DOI: 10.1109/IPDPS.2007.370426.

[BAM07] R. Bourqui, D. Auber, and P. Mary. "How to draw clustered-weighted graphs using a multilevel force-directed graph drawing algorithm". In: *Information Visualization, 2007. IV '07. 11th International Conference*. July 2007, pp. 757–764. DOI: 10.1109/IV.2007.65.

[BBD+17] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. "A taxonomy and survey of dynamic graph visualization". In: *Comput. Graph. Forum* 36.1 (2017), pp. 133–159. DOI: 10.1111/cgf.12791.

[BC01] Ralf Brockenauer and Sabine Cornelsen. "Drawing clusters and hierarchies". In: *Drawing Graphs: Methods and Models*. Ed. by Michael Kaufmann and Dorothea Wagner. LNCS 2025. Berlin, Germany: Springer-Verlag, 2001, pp. 193–227. ISBN: 3-540-42062-2.

[BD07] M. Balzer and O. Deussen. "Level-of-detail visualization of clustered graph layouts". In: *Visualization, 2007. APVIS '07. 2007 6th International Asia-Pacific Symposium on*. Feb. 2007, pp. 133–140. DOI: 10.1109/APVIS.2007.329288.

[BEK+02] Ulrik Brandes, Markus Eiglsperger, Michael Kaufmann, and Dorothea Wagner. "Sketch-driven orthogonal graph drawing". In: *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*. Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, pp. 1–11. ISBN: 978-3-540-00158-4. DOI: 10.1007/3-540-36151-0.

[BFG+97] Stina S. Bridgeman, Jody Fanto, Ashim Garg, Roberto Tamassia, and Luca Vismara. "InteractiveGiotto: an algorithm for interactive orthogonal graph drawing". In: *Graph Drawing (Proc. GD '97)*. Vol. 1353. Springer, 1997, pp. 303–308.

[BJL01] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. "A fast layout algorithm for k-level graphs". In: *Proceedings of the 8th International Symposium on Graph Drawing (GD '00)*. Ed. by Joe Marks. Vol. 1984. LNCS. Springer, 2001, pp. 229–240. ISBN: 978-3-540-41554-1. DOI: 10.1007/3-540-44541-2.

[BJM02] Wilhelm Barth, Michael Jünger, and Petra Mutzel. "Simple and efficient bilayer cross counting". In: *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*. Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, pp. 331–360.

[BK02] Ulrik Brandes and Boris Köpf. "Fast and simple horizontal coordinate assignment". In: *Proceedings of the 9th International Symposium on Graph Drawing (GD '01)*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. LNCS. Springer, 2002, pp. 33–36. ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4.

[BLM+02] Jürgen Branke, Stefan Leppert, Martin Middendorf, and Peter Eades. "Width-restricted layering of acyclic digraphs with consideration of dummy nodes". In: *Information Processing Letters* 81 (2002), pp. 59–63.

[BM99] François Bertault and Mirka Miller. "An algorithm for drawing compound graphs". In: *Graph Drawing*. Ed. by Jan Kratochvíyl.

Vol. 1731. LNCS. Springer Berlin / Heidelberg, 1999, pp. 197–204. ISBN: 978-3-540-66904-3. DOI: 10.1007/3-540-46648-7.

[Boe80] Wolfgang Boehm. "Inserting new knots into b-spline curves". In: *Computer-Aided Design* 12.4 (1980), pp. 199–201. ISSN: 0010-4485. DOI: http://dx.doi.org/10.1016/0010-4485(80)90154-2.

[Bra01] Jürgen Branke. "Dynamic graph drawing". In: *Drawing Graphs: Methods and Models*. Ed. by Michael Kaufmann and Dorothea Wagner. Vol. 2025. LNCS. Springer, 2001.

[BRS+07] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. "The aesthetics of graph visualization". In: *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CAe'07)*. Banff, Alberta, Canada: Eurographics Association, 2007, pp. 57–64.

[BT02] Stina S. Bridgeman and Roberto Tamassia. "A user study in similarity measures for graph drawing". In: *Journal of Graph Algorithms and Applications* 6.3 (2002), pp. 225–254.

[BT98] Stina Bridgeman and Roberto Tamassia. "Difference metrics for interactive orthogonal graph drawing algorithms". In: *Graph Drawing*. Springer, 1998, pp. 57–71.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[Car12] John Julian Carstens. "Node and label placement in a layered layout algorithm". Master's thesis. Kiel University, Department of Computer Science, Sept. 2012.

[Cat95] Tiziana Catarci. "The assignment heuristic for crossing reduction". In: *IEEE Trans. Systems, Man, and Cybernetics* 25.3 (1995), pp. 515–521. DOI: 10.1109/21.364865.

[CG72] Edward G. Coffman. and Ronald L. Graham. "Optimal scheduling for two-processor systems". In: *Acta Informatica* 1.3 (1972), pp. 200–213. ISSN: 0001-5903. DOI: 10.1007/BF00288685.

[CL97] Hsiao-Feng Steven Chen and D.T. Lee. "A faster one-dimensional topological compaction algorithm". In: *Algorithms and Computation*. Ed. by HonWai Leong, Hiroshi Imai, and Sanjay Jain. Vol. 1350. Springer, 1997, pp. 303–313. ISBN: 978-3-540-63890-2. DOI: 10.1007/3-540-63890-3_33.

[CMT02] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. "A framework for the static and interactive visualization of statecharts". In: *Journal of Graph Algorithms and Applications* 6.3 (2002), pp. 313–351.

[CMT04] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. "ViSta – visualizing statecharts". In: *Graph Drawing Software*. Ed. by Michael Jünger and Petra Mutzel. Springer, 2004, pp. 299–320. DOI: 10.1007/978-3-642-18638-7.

[Cyr17] Michael Cyruk. "Graph layout of connected components". Master's thesis. Kiel University, Department of Computer Science, May 2017.

[DET+99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing: Algorithms for the visualization of graphs*. Prentice Hall, 1999. ISBN: 0-13-301615-3.

[DGC+05] Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. "A compound graph layout algorithm for biological pathways". In: *Graph Drawing*. Ed. by János Pach. Vol. 3383. LNCS. Springer Berlin / Heidelberg, 2005, pp. 442–447. ISBN: 978-3-540-24528-5. DOI: 10.1007/978-3-540-31843-9.

[DGC+09] Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. "A layout algorithm for undirected compound graphs". In: *Information Sciences* 179.7 (2009), pp. 980–994. ISSN: 0020-0255. DOI: 10.1016/j.ins.2008.11.017.

[DGL+97a] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Armando Parise, Roberto Tamassia, Emanuele Tassinari, Francesco Vargiu, and Luca Vismara. "Drawing directed acyclic graphs: an experimental study". In: *Proceedings of the Symposium on*

Graph Drawing (GD '96). Ed. by Stephen C. North. Vol. 1190. LNCS. Springer, 1997, pp. 76–91. DOI: 10.1007/3-540-62495-3_39.

[DGL+97b] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Roberto Tamassia, Emanuele Tassinari, and Francesco Vargiu. "An experimental comparison of four graph drawing algorithms". In: *Computational Geometry* 7.5-6 (1997), pp. 303–325.

[DH96] Ron Davidson and David Harel. "Drawing graphs nicely using simulated annealing". In: *ACM Transactions on Graphics* 15.4 (Oct. 1996). http://www.acm.org/pubs/toc/Abstracts/0730-0301/234538.html, pp. 301–331. ISSN: 0730-0301.

[DK05] Tim Dwyer and Yehuda Koren. "Dig-CoLa: directed graph layout through constrained energy minimization". In: *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'05)*. Oct. 2005, pp. 65–72. DOI: 10.1109/INFVIS.2005.1532130.

[DKM06a] Tim Dwyer, Yehuda Koren, and Kim Marriott. "IPSep-CoLa: an incremental procedure for separation constraint layout of graphs". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 821–828. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.156.

[DKM06b] Tim Dwyer, Yehuda Koren, and Kim Marriott. "Stress majorization with orthogonal ordering constraints". In: *Proceedings of the 13th International Symposium on Graph Drawing (GD '05)*. Ed. by Patrick Healy and Nikola S. Nikolov. Vol. 3843. LNCS. Springer, 2006, pp. 141–152. DOI: 10.1007/11618058.

[DKM09] Tim Dwyer, Yehuda Koren, and Kim Marriott. "Constrained graph layout by stress majorization and gradient projection". In: *Discrete Mathematics* 309.7 (2009), pp. 1895–1908. ISSN: 0012-365X. DOI: 10.1016/j.disc.2007.12.103.

[DLF+09] Tim Dwyer, Bongshin Lee, Danyel Fisher, Kori Inkpen Quinn, Petra Isenberg, George Robertson, and Chris North. "A comparison of user-generated and automatic graph layouts." In: *IEEE transactions on visualization and computer graphics* 15.6 (2009), pp. 961–8. ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.109.

[DMS+08] Tim Dwyer, Kim Marriott, Falk Schreiber, Peter Stuckey, Michael Woodward, and Michel Wybrow. "Exploration of networks using overview+detail with constraint-based cooperative layout". In: *IEEE Transactions on Visualization and Computer Graphics* 14.6 (Nov. 2008), pp. 1293–1300. ISSN: 1077-2626. DOI: 10.1109/TVCG.2008.130.

[DMS06] Tim Dwyer, Kim Marriott, and Peter J. Stuckey. "Fast node overlap removal". In: *Proceedings of the 13th International Symposium on Graph Drawing (GD '05)*. Ed. by Patrick Healy and Nikola S. Nikolov. Vol. 3843. LNCS. Springer, 2006, pp. 153–164. DOI: 10.1007/11618058.

[DMW09a] Tim Dwyer, Kim Marriott, and Michael Wybrow. "Dunnart: a constraint-based network diagram authoring tool". In: *Revised Papers of the 16th International Symposium on Graph Drawing (GD '08)*. Vol. 5417. LNCS. Springer, 2009, pp. 420–431. ISBN: 978-3-642-00218-2. DOI: 10.1007/978-3-642-00219-9.

[DMW09b] Tim Dwyer, Kim Marriott, and Michael Wybrow. "Topology preserving constrained graph layout". In: *Revised Papers of the 16th International Symposium on Graph Drawing (GD '08)*. Vol. 5417. LNCS. Springer, 2009, pp. 230–241. ISBN: 978-3-642-00218-2. DOI: 10.1007/978-3-642-00219-9.

[Dre94] Stefan Dresbach. "A new heuristic layout algorithm for dags". In: *Operations Research Proceedings 1994*. Springer, 1994, pp. 121–126.

[Ead84] Peter Eades. "A heuristic for graph drawing". In: *Congressus Numerantium* 42 (1984), pp. 149–160. ISSN: 0384-9864.

[EF96] Peter Eades and Qingwen Feng. *Orthogonal grid drawing of clustered graphs*. Tech. rep. University of Newcastle, Australia, 1996.

[EGB03] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. "Crossing reduction for orthogonal circuit visualization". In: *Proceedings of the 2003 International Conference on VLSI*. CSREA Press, 2003, pp. 107–113.

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

[EGK+04] Markus Eiglsperger, Carsten Gutwenger, Michael Kaufmann, Joachim Kupke, Michael Jünger, Sebastian Leipert, Karsten Klein, Petra Mutzel, and Martin Siebenhaller. "Automatic layout of UML class diagrams in orthogonal style". In: *Information Visualization* 3.3 (2004), pp. 189–208. DOI: 10.1057/palgrave.ivs.9500078.

[EH00] Peter Eades and Mao Lin Huang. "Navigating clustered graphs using force-directed methods". In: *J. Graph Algorithms Appl.* 4.3 (2000), pp. 157–181.

[EK86] Peter Eades and David Kelly. "Heuristics for reducing crossings in 2-layered networks". In: *Ars Combinatoria* 21 (1986), pp. 89–98.

[ELM+91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. "Preserving the mental map of a diagram". In: *Proceedings of the First International Conference on Computational Graphics and Visualization Techniques*. 1991, pp. 34–43.

[ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. "A fast and effective heuristic for the feedback arc set problem". In: *Information Processing Letters* 47.6 (1993), pp. 319–323. ISSN: 0020-0190. DOI: 10.1016/0020-0190(93)90079-O.

[ES90] Peter Eades and Kozo Sugiyama. "How to draw a directed graph". In: *Journal of Information Processing* 13.4 (1990), pp. 424–437. ISSN: 0387-6101.

[EW94] Peter Eades and Nicholas C. Wormald. "Edge crossings in drawings of bipartite graphs". In: *Algorithmica* 11.4 (1994), pp. 379–403. DOI: 10.1007/BF01187020.

[FDK02] Karlis Freivalds, Ugur Dogrusoz, and Paulis Kikusts. "Disconnected graph layout and the polyomino packing approach". English. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 378–391. ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4_30.

[FH10] Hauke Fuhrmann and Reinhard von Hanxleden. "Taming graphical modeling". In: *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS '10)*. Vol. 6394. LNCS. Springer, Oct. 2010, pp. 196–210. DOI: 10.1007/978-3-642-16145-2.

[FHK+14] Patrick Frey, Reinhard von Hanxleden, Christoph Krüger, Ulf Rüegg, Christian Schneider, and Miro Spönemann. "Efficient exploration of complex data flow models". In: *Proceedings of Modellierung 2014*. 2014, pp. 321–336.

[FK01] Karlis Freivalds and Paulis Kikusts. "Optimum layout adjustment supporting ordering constraints in graph-like diagram drawing". In: *Proceedings of the Latvian Academy of Sciences*. Vol. 55. 2001, pp. 43–51.

[Flo17] Astrid Mariana Flohr. "Edge routing with immutable node positions". Master's thesis. Kiel University, Department of Computer Science, Aug. 2017.

[For02] Michael Forster. "Applying crossing reduction strategies to layered compound graphs". In: *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*. Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, pp. 115–132. ISBN: 978-3-540-00158-4. DOI: 10.1007/3-540-36151-0.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph drawing by force-directed placement". In: *Software—Practice & Experience* 21.11 (1991), pp. 1129–1164. ISSN: 0038-0644. DOI: http://dx.doi.org/10.1002/spe.4380211102.

[FS04] Carsten Friedrich and Falk Schreiber. "Flexible layering in hierarchical drawings with nodes of arbitrary size". In: *Proceedings of the 27th Australasian Conference on Computer Science (ACSC'04)*. Australian Computer Society, Inc., 2004, pp. 369–376.

[FS09] Thibaut Feydy and Peter J. Stuckey. "Lazy clause generation reengineered". In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*. Ed. by Ian P. Gent. Vol. 5732. Lecture Notes in Computer Science. Springer, 2009, pp. 352–366. DOI: 10.1007/978-3-642-04244-7_29.

[FT04] Yaniv Frishman and Ayellet Tal. "Dynamic drawing of clustered graphs". In: *INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 191–198. DOI: 10.1109/INFOVIS.2004.18.

[Fuh12] Insa Fuhrmann. "Layout of compound graphs". Diploma thesis. Kiel University, Department of Computer Science, Feb. 2012.

[Fur06] George W. Furnas. "A fisheye follow-up: further reflections on focus + context". In: *Proceedings of the 2006 Conference on Human Factors in Computing Systems CHI '06*. ACM, 2006, pp. 999–1008. DOI: 10.1145/1124772.1124921.

[GH09] Emden R. Gansner and Yifan Hu. "Efficient node overlap removal using a proximity stress model". In: *Graph Drawing*. Ed. by Ioannis G. Tollis and Maurizio Patrignani. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 206–217. ISBN: 978-3-642-00218-2.

[GHK13] Emden R. Gansner, Yifan Hu, and Shankar Krishnan. "COAST: A convex optimization approach to stress-based embedding". In: *Proceedings of the 21th International Symposium on Graph Drawing (GD '13)*. 2013, pp. 268–279. DOI: 10.1007/978-3-319-03841-4_24.

[GHM+14] Carsten Gutwenger, Reinhard von Hanxleden, Petra Mutzel, Ulf Rüegg, and Miro Spönemann. “Examining the compactness of automatic layout algorithms for practical diagrams”. In: *Proceedings of the Workshop on Graph Visualization in Practice (GraphViP ’14)*. 2014, pp. 42–52.

[GHN12] Emden R. Gansner, Yifan Hu, and Stephen C. North. “A maxent-stress model for graph layout”. In: *Proceedings of the 2012 IEEE Pacific Visualization Symposium*. 2012, pp. 73–80. DOI: 10.1109/PacificVis.2012.6183576.

[GHN13] Emden R. Gansner, Yifan Hu, and Stephen C. North. “Interactive visualization of streaming text data with dynamic maps”. In: *J. Graph Algorithms Appl.* 17.4 (2013), pp. 515–540. DOI: 10.7155/jgaa.00302.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractibility: a guide to the theory of NP-completeness*. New York: W. H. Freeman & Co, 1979.

[GJ83] Michael R. Garey and David S. Johnson. “Crossing number is NP-complete”. In: *SIAM Journal on Algebraic and Discrete Methods* 4.3 (1983), pp. 312–316. DOI: 10.1137/0604033.

[GJK+03] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. “A new approach for visualizing UML class diagrams”. In: *Proceedings of the ACM Symposium on Software Visualization (SoftVis’03)*. San Diego, California: ACM, 2003, pp. 179–188. ISBN: 1-58113-642-0. DOI: 10.1145/774833.774859.

[GKN+93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. “A technique for drawing directed graphs”. In: *Software Engineering* 19.3 (1993), pp. 214–230.

[GKN02] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. Technical Report. Murray Hill, NJ, USA: AT&T Bell Laboratories, Feb. 2002, p. 40.

[GKN05] Emden R. Gansner, Yehuda Koren, and Stephen C. North. "Graph drawing by stress majorization". In: *Graph Drawing*. Ed. by János Pach. Vol. 3383. LNCS. Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-24528-5. DOI: 10.1007/978-3-540-31843-9_25.

[GKS07] Dennis Goehlsdorf, Michael Kaufmann, and Martin Siebenhaller. "Placing connected components of disconnected graphs". In: *6th International Asia-Pacific Symposium on Visualization, 2007*. Feb. 2007, pp. 101–108. DOI: 10.1109/APVIS.2007.329283.

[Gre17] Daniel Grevismühl. "Stable diagram compaction". Master's thesis. Kiel University, Department of Computer Science, Dec. 2017.

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[Har88] David Harel. "On visual formalisms". In: *Communications of the ACM* 31.5 (1988). http://doi.acm.org/10.1145/42411.42414, pp. 514–530. ISSN: 0001-0782.

[Har98] David Harel. "On the aesthetics of diagrams". In: *LNCS, Mathematics of Program Construction* 1422/1998 (1998), pp. 1–5.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274. Edinburgh, UK: ACM, June 2014.

[HE98] Mao Lin Huang and Peter Eades. "A fully animated interactive system for clustering and navigating huge graphs". In: *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*. London, UK: Springer-Verlag, 1998, pp. 374–383. ISBN: 3-540-65473-9.

[HIM+98] Kunihiko Hayashi, Michiko Inoue, Toshimitsu Masuzawa, and Hideo Fujiwara. "A layout adjustment problem for disjoint rectangles preserving orthogonal order". In: *Graph Drawing*. Springer. 1998, pp. 183–197.

[HLS+07] Xiaodi Huang, Wei Lai, A.S.M. Sajeev, and Junbin Gao. "A new algorithm for removing node overlapping in graph visualization". In: *Information Sciences* 177.14 (2007), pp. 2821–2844.

[HMN04] Seok-Hee Hong, Damian Merrick, and Hugo A. D. do Nascimento. "The metro map layout problem". In: *Proceedings of the 12th International Symposium on Graph Drawing (GD '04)*. 2004, pp. 482–491. DOI: 10.1007/978-3-540-31843-9_50.

[HN02a] Patrick Healy and Nikola S. Nikolov. "A branch-and-cut approach to the directed acyclic graph layering problem". In: *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*. Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, pp. 98–109. ISBN: 978-3-540-00158-4. DOI: 10.1007/3-540-36151-0.

[HN02b] Patrick Healy and Nikola S. Nikolov. "How to layer a directed acyclic graph". In: *Proceedings of the 9th International Symposium on Graph Drawing (GD '01)*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. LNCS. Springer, 2002, pp. 16–30. DOI: 10.1007/3-540-45848-4_2.

[HN13] Patrick Healy and Nikola S. Nikolov. "Hierarchical drawing algorithms". In: *Handbook of Graph Drawing and Visualization*. Ed. by Roberto Tamassia. CRC Press, 2013, pp. 409–453.

[Höh06] Stephan Höhrmann. "Entwicklung eines modularen Feldbussystems zur Steuerung einer Modellbahnanlage". Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Mar. 2006.

[HT92] Susanne E. Hambrusch and Hung-Yi Tu. "Minimizing total wire length during 1-dimensional compaction". In: *Integration, the VLSI Journal* 14.2 (1992), pp. 113–144. ISSN: 0167-9260. DOI: http://dx.doi.org/10.1016/0167-9260(92)90023-R.

[IMM+09] T. Itoh, C. Muelder, Kwan-Liu Ma, and Jun Sese. "A hybrid space-filling and force-directed layout method for visualizing multiple-category graphs". In: *Visualization Symposium, 2009. PacificVis '09. IEEE Pacific*. Apr. 2009, pp. 121–128. DOI: 10.1109/PACIFICVIS.2009.4906846.

[Jam86] Geoffrey James. *The tao of programming*. Info Books, 1986. ISBN: 978-0931137075.

[JM04] Michael Jünger and Petra Mutzel, eds. *Graph drawing software*. Springer, 2004. ISBN: 978-3-642-62214-4.

[JM97] Michael Jünger and Petra Mutzel. "2-layer straightline crossing minimization: performance of exact and heuristic algorithms". In: *Journal of Graph Algorithms and Applications* 1 (1997), pp. 1–25. ISSN: 1526-1719.

[JMM+16] Adalat Jabrayilov, Sven Mallach, Petra Mutzel, Ulf Rüegg, and Reinhard von Hanxleden. "Compact layered drawings of general directed graphs". In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD '16)*. 2016, pp. 209–221. DOI: 10.1007/978-3-319-50106-2.

[Kar72] Richard M. Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*. Ed. by Raymond E. Miller and James W. Thatcher. New York: Plenum Press, 1972, pp. 85–103.

[KD10] Lars Kristian Klauske and Christian Dziobek. "Improving modeling usability: automated layout generation for Simulink". In: *Proceedings of the MathWorks Automotive Conference (MAC'10)*. 2010.

[KDM+16] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. "HOLA: human-like orthogonal network layout". In: *IEEE Trans. Vis. Comput. Graph.* 22.1 (2016), pp. 349–358. DOI: 10.1109/TVCG.2015.2467451.

[KHK+12] Marc Khoury, Yifan Hu, Shankar Krishnan, and Carlos Eduardo Scheidegger. "Drawing large graphs by low-rank stress majorization". In: *Computer Graphics Forum* 31.3 (2012), pp. 975–984. DOI: 10.1111/j.1467-8659.2012.03090.x.

[KK89] Tomihisa Kamada and Satoru Kawai. "An algorithm for drawing general undirected graphs". In: *Information Processing Letters* 31.1 (1989), pp. 7–15. ISSN: 0020-0190.

[Kla01] Gunnar Werner Klau. "A combinatorial approach to orthogonal placement problems". PhD thesis. Universität des Saarlandes, Sept. 2001.

[Kla12] Lars Kristian Klauske. "Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus". PhD thesis. Technische Universität Berlin, 2012.

[KPS14] Stephen G. Kobourov, Sergey Pupyrev, and Bahador Saket. "Are crossings important for drawing large graphs?" In: *Proceedings of the 22nd International Symposium on Graph Drawing (GD '14)*. Vol. 8871. Springer, 2014, pp. 234–245. ISBN: 978-3-662-45802-0. DOI: 10.1007/978-3-662-45803-7_20.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "Integrated definition of abstract and concrete syntax for textual languages". In: *Model Driven Engineering Languages and Systems (MoDELS'07)*. Vol. 4735. LNCS. Springer Berlin / Heidelberg, 2007, pp. 286–300. ISBN: 978-3-540-75208-0. DOI: 10.1007/978-3-540-75209-7_20. URL: http://www.springerlink.com/content/c9l5p5267118607m/.

[KSS+12] Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Improved layout for data flow diagrams with port constraints". In: *Proceedings of the 7th International Conference on the Theory and Application of*

Diagrams (DIAGRAMS '12). Vol. 7352. LNAI. Springer, 2012, pp. 65–79. DOI: 10.1007/978-3-642-31223-6.

[KW01] Michael Kaufmann and Dorothea Wagner, eds. *Drawing graphs: Methods and models*. LNCS 2025. Berlin, Germany: Springer-Verlag, 2001. ISBN: 3-540-42062-2.

[LA94] Ying K. Leung and Mark D. Apperley. "A review and taxonomy of distortion-oriented presentation techniques". In: *ACM Transactions on Computer-Human Interaction* 1.2 (June 1994), pp. 126–160.

[LEN05] Wanchun Li, Peter Eades, and Nikola Nikolov. "Using spring algorithms to remove node overlapping". In: *Proceedings of the 2005 Asia-Pacific symposium on Information visualisation-Volume 45*. Australian Computer Society, Inc. 2005, pp. 131–140.

[Len90] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN: 0-471-92838-0.

[LLW+01] Jianbang Lai, Ming-Shiun Lin, Ting-Chi Wang, and Li-C Wang. "Module placement with boundary constraints using the sequence-pair representation". In: *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. ACM. 2001, pp. 515–520.

[Man17] Kim Christian Mannstedt. "Alternative layering strategies for sugiyama layout". Master's thesis. Kiel University, Department of Computer Science, Dec. 2017.

[McA99] Andrew J. McAllister. *A new heuristic algorithm for the linear arrangement problem*. Tech. rep. University of New Brunswick, 1999.

[MEL+95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. "Layout adjustment and the mental map". In: *Journal of Visual Languages & Computing* 6.2 (June 1995), pp. 183–210. DOI: 10.1006/jvlc.1995.1010.

[ML03] Rafael Martí and Manuel Laguna. "Heuristics and meta-heuristics for 2-layer straight line crossing minimization". In: *Discrete Applied Mathematics* 127.3 (2003), pp. 665–678. DOI: 10.1016/S0166-218X(02)00397-9.

[MMM08] Sonja Maier, Steffen Mazanek, and Mark Minas. "Layout specification on the concrete and abstract syntax level of a diagram language". In: *Proceedings of the Second International Workshop on Layout of (Software) Engineering Diagrams (LED'08)*. Vol. 13. Electronic Communications of the EASST. Berlin, Germany, 2008.

[MPW+12] Kim Marriott, Helen Purchase, Michael Wybrow, and Cagatay Goncu. "Memorability of Visual Features in Network Diagrams". In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (Dec. 2012), pp. 2477–2485. ISSN: 1077-2626. DOI: 10.1109/TVCG.2012.245.

[MST+03] Kim Marriott, Peter Stuckey, Vincent Tam, and Weiqing He. "Removing node overlapping in graph layout using constrained optimization". In: *Constraints* 8.2 (Apr. 2003), pp. 143–171. ISSN: 1383-7133.

[MSW14] Kim Marriott, Peter J. Stuckey, and Michael Wybrow. "Seeing around corners: fast orthogonal connector routing". In: *Proceedings of the 8th International Conference on Diagrammatic Representation and Inference (Diagrams '14)*. 2014, pp. 31–37. DOI: 10.1007/978-3-662-44043-8_4.

[MTL78] Robert McGill, John W. Tukey, and Wayne A. Larsen. "Variations of box plots". In: *The American Statistician* 32.1 (1978), pp. 12–16. ISSN: 00031305.

[NNB+16] Lev Nachmanson, Arlind Nocaj, Sergey Bereg, Leishi Zhang, and Alexander Holroyd. "Node overlap removal by growing a tree". In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD '16)*. Ed. by Yifan Hu and Martin Nöllenburg. Lecture Notes in Computer Sci-

ence. Springer, 2016, pp. 33–43. ISBN: 978-3-319-50106-2. DOI: 10.1007/978-3-319-50106-2_3.

[NRL08] Lev Nachmanson, George Robertson, and Bongshin Lee. "Drawing graphs with GLEE". English. In: *Graph Drawing*. Ed. by Seok-Hee Hong, Takao Nishizeki, and Wu Quan. Vol. 4875. LNCS. Springer Berlin Heidelberg, 2008, pp. 389–394. ISBN: 978-3-540-77536-2. DOI: 10.1007/978-3-540-77537-9_38.

[NRL11] Lev Nachmanson, George Robertson, and Bongshin Lee. *Layered graph layouts with a given aspect ratio*. US Patent 7,932,907. Apr. 2011. URL: http://www.google.com.ar/patents/US7932907.

[NT06] Nikola S. Nikolov and Alexandre Tarassov. "Graph layering by promotion of nodes". In: *Discrete Applied Mathematics* 154.5 (2006), pp. 848–860. ISSN: 0166-218X. DOI: 10.1016/j.dam.2005.05.023.

[NTB05] Nikola S. Nikolov, Alexandre Tarassov, and Jürgen Branke. "In search for efficient heuristics for minimum-width graph layering with consideration of dummy nodes". In: *Journal of Experimental Algorithmics* 10 (2005). ISSN: 1084-6654. DOI: 10.1145/1064546.1180618.

[NW02] Stephen C. North and Gordon Woodhull. "Online hierarchical graph drawing". In: *Revised Papers of the 9th International Symposium on Graph Drawing*. Vol. 2265. LNCS. Springer, 2002, pp. 232–246. ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4.

[NW11] Martin Nöllenburg and Alexander Wolff. "Drawing and labeling high-quality metro maps by mixed-integer programming". In: *IEEE Trans. Vis. Comput. Graph.* 17.5 (2011), pp. 626–641. DOI: 10.1109/TVCG.2010.81. URL: https://doi.org/10.1109/TVCG.2010.81.

[OKB17] Mark Ortmann, Mirza Klimenta, and Ulrik Brandes. "A sparse stress model". In: *Journal of Graph Algorithms and Applications* 21.5 (2017), pp. 791–821. DOI: 10.7155/jgaa.00440.

[OSC07] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. “Propagation = lazy clause generation”. In: *Principles and Practice of Constraint Programming – CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 544–558. DOI: 10.1007/978-3-540-74970-7_39.

[Pet95] Marian Petre. “Why looking isn’t always seeing: Readership skills and graphical programming”. In: *Communications of the ACM* 38.6 (June 1995), pp. 33–44.

[PMD+12] J.J. Pantrigo, R. Martí, A. Duarte, and E.G. Pardo. “Scatter search for the cutwidth minimization problem”. In: *Annals of Operations Research* 199.1 (2012), pp. 285–304. DOI: 10.1007/s10479-011-0907-2.

[PMR16] Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. “Evolving models in Model-Driven Engineering: state-of-the-art and future challenges”. In: *Journal of Systems and Software* 111 (2016), pp. 272–280. ISSN: 0164-1212.

[PPP12] Helen C. Purchase, Christopher Pilcher, and Beryl Plimmer. “Graph drawing aesthetics—created by users, not algorithms”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2012), pp. 81–92. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.269.

[Pto14] Claudius Ptolemaeus, ed. *System design, modeling, and simulation using Ptolemy II*. Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.

[Pur02] Helen C. Purchase. “Metrics for graph drawing aesthetics”. In: *Journal of Visual Languages and Computing* 13.5 (2002), pp. 501–516.

[Pur97] Helen C. Purchase. “Which aesthetic has the greatest effect on human understanding?” In: *Proceedings of the 5th International Symposium on Graph Drawing (GD ’97)*. Vol. 1353. LNCS. Springer, 1997, pp. 248–261.

[R C15] R Core Team. *R: a language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, Austria, 2015. URL: https://www.R-project.org/.

[Rai06] Marcus Raitner. "Efficient Visual Navigation of Hierarchically Structured Graphs". PhD thesis. Universität Passau, Feb. 2006.

[Rei15] Sven Oliver Reimers. "Port-aware node placement in a layer-based layout algorithm". Diploma thesis. Kiel University, Department of Computer Science, Mar. 2015.

[RES+15] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. *A generalization of the directed graph layering problem*. Technical Report 1501. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2015.

[RES+16a] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. "A generalization of the directed graph layering problem". In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD '16)*. 2016, pp. 196–208. DOI: 10.1007/978-3-319-50106-2_16.

[RES+16b] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. *A generalization of the directed graph layering problem*. arXiv:1608.07809 [cs.OH]. 2016. arXiv: 1608.07809[cs.OH].

[RES+17] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. "Generalized layerings for arbitrary and fixed drawing areas". In: *Journal of Graph Algorithms and Applications* 21.5 (2017), pp. 823–856. DOI: 10.7155/jgaa.00441.

[RH18a] Ulf Rüegg and Reinhard von Hanxleden. "Wrapping layered graphs". In: *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*. To appear. Springer, 2018.

[RH18b] Ulf Rüegg and Reinhard von Hanxleden. *Wrapping layered graphs*. Technical Report 1803. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2018.

[RHM+12] Louis M. Rose et al. “Graph and model transformation tools for model migration”. In: *Software & Systems Modeling* 13.1 (2012), pp. 323–359. ISSN: 1619-1374.

[RKD+14a] Ulf Rüegg, Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. “Stress-minimizing orthogonal layout of data flow diagrams with ports”. In: *Proceedings of the 22nd International Symposium on Graph Drawing (GD '14)*. 2014, pp. 319–330. DOI: 10.1007/978-3-662-45803-7_27.

[RKD+14b] Ulf Rüegg, Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. *Stress-minimizing orthogonal layout of data flow diagrams with ports*. arXiv:1408.4626 [cs.OH]. 2014. arXiv: 1408.4626[cs.OH].

[RLP+16] Ulf Rüegg, Rajneesh Lakkundi, Ashwin Prasad, Anand Kodaganur, Christoph Daniel Schulze, and Reinhard von Hanxleden. “Incremental diagram layout for automated model migration”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS '16)*. 2016, pp. 185–195. DOI: 10.1145/2976767.2976805.

[RSC+15] Ulf Rüegg, Christoph Daniel Schulze, John Julian Carstens, and Reinhard von Hanxleden. “Size- and port-aware horizontal node coordinate assignment”. In: *Proceedings of the 23rd International Symposium on Graph Drawing and Network Visualization (GD '15)*. 2015, pp. 139–150. DOI: 10.1007/978-3-319-27261-0_12.

[RSG+16a] Ulf Rüegg, Christoph Daniel Schulze, Daniel Grevismühl, and Reinhard von Hanxleden. “Using one-dimensional compaction for smaller graph drawings”. In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS '16)*. 2016, pp. 212–218. DOI: 10.1007/978-3-319-42333-3_16.

[RSG+16b] Ulf Rüegg, Christoph Daniel Schulze, Daniel Grevismühl, and Reinhard von Hanxleden. *Using one-dimensional compaction for smaller graph drawings*. Technical Report 1601. ISSN 2192-6247. Kiel University, Department of Computer Science, Apr. 2016.

[RT81] E. M. Reingold and J. S. Tilford. "Tidier drawing of trees". In: *IEEE Transactions on Software Engineering* 7 (Mar. 1981), pp. 223–228.

[Rus13] Adrian Rusu. "Tree drawing algorithms". In: *Handbook of Graph Drawing and Visualization*. Ed. by Roberto Tamassia. CRC Press, 2013, pp. 155–192.

[San04] Georg Sander. "Layout of directed hypergraphs with orthogonal hyperedges". In: *Proceedings of the 11th International Symposium on Graph Drawing (GD '03)*. Ed. by Giuseppe Liotta. Vol. 2912. LNCS. Springer, 2004, pp. 381–386. ISBN: 978-3-540-20831-0. DOI: 10.1007/978-3-540-24595-7_35.

[San96a] Georg Sander. "A fast heuristic for hierarchical Manhattan layout". In: *Proceedings of the Symposium on Graph Drawing (GD '95)*. Ed. by Franz J. Brandenburg. Vol. 1027. LNCS. Springer, 1996, pp. 447–458. ISBN: 3-540-60723-4. DOI: 10.1007/BFb0021828.

[San96b] Georg Sander. *Layout of compound directed graphs*. Tech. rep. A/03/96. 66041 Saarbrücken: Universität des Saarlandes, FB 14 Informatik, June 1996.

[San99] Georg Sander. "Graph layout for applications in compiler construction". In: *Theoretical Computer Science* 217.2 (1999), pp. 175–214.

[SB92] Manojit Sarkar and Marc H. Brown. "Graphical fisheye views of graphs". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1992, pp. 83–91. ISBN: 0-89791-513-5. DOI: http://doi.acm.org/10.1145/142750.142763.

[Sch16] Alan Schelten. "Hierarchy-aware layer sweep". Master's thesis. Kiel University, Department of Computer Science, Sept. 2016.

[SDH14] Miro Spönemann, Björn Duderstadt, and Reinhard von Hanxleden. *Evolutionary meta layout of graphs*. Technical Report 1401. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Jan. 2014.

[SFH+10] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. "Port constraints in hierarchical layout of data flow diagrams". In: *Proceedings of the 17th International Symposium on Graph Drawing (GD '09)*. Vol. 5849. LNCS. Springer, 2010, pp. 135–146. DOI: 10.1007/978-3-642-11805-0.

[Skr15] Sandra Skrlac. "Enhanced port constraints in a layer-based layout". Bachelor's thesis. Kiel University, Department of Computer Science, Sept. 2015.

[SM91] Kozo Sugiyama and Kazuo Misue. "Visualization of structural information: automatic drawing of compound digraphs". In: *IEEE Transactions on Systems, Man and Cybernetics* 21.4 (July 1991), pp. 876–892. ISSN: 0018-9472. DOI: 10.1109/21.108304.

[SM96] Kozo Sugiyama and Kazuo Misue. "A generic compound graph visualizer/manipulator: d-abductor". In: *Graph Drawing*. Ed. by Franz Brandenburg. Vol. 1027. LNCS. Springer Berlin / Heidelberg, 1996, pp. 500–503. ISBN: 978-3-540-60723-6. DOI: 10.1007/BFb0021834.

[Spö15] Miro Spönemann. *Graph layout support for model-driven engineering*. Kiel Computer Science Series 2015/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2015. ISBN: 9783734772689.

[Spr16] Carsten Sprung. "Edge bundling techniques for dataflow diagrams". Master's thesis. Kiel University, Department of Computer Science, Oct. 2016.

[SR04] Jonathan M. Stott and Peter Rodgers. "Metro map layout using multicriteria optimization". In: *8th International Conference on Information Visualisation, IV 2004*. 2004, pp. 355–362. DOI: 10.1109/IV.2004.1320168.

[SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing layered graphs with port constraints". In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[SSR+14a] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. “Counting crossings for layered hypergraphs”. In: *Proceedings of the 8th International Conference on the Theory and Application of Diagrams (DIAGRAMS ’14)*. Vol. 8578. LNAI. Springer, 2014, pp. 9–15. DOI: 10.1007/978-3-662-44043-8_2.

[SSR+14b] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. *Drawing layered hypergraphs*. Technical Report 1404. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Apr. 2014.

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

[Tam13] Roberto Tamassia, ed. *Handbook of graph drawing and visualization*. CRC Press, 2013. ISBN: 978-1584884125.

[Tam87] Roberto Tamassia. “On embedding a graph in the grid with the minimum number of bends”. In: *SIAM Journal of Computing* 16.3 (1987), pp. 421–444. ISSN: 0097-5397.

[TDB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. “Automatic graph drawing and readability of diagrams”. In: *IEEE Transactions on Systems, Man and Cybernetics* 18.1 (1988), pp. 61–79. ISSN: 0018-9472.

[TNB04] Alexandre Tarassov, Nikola S. Nikolov, and Jürgen Branke. “A heuristic for minimum-width graph layering with consideration of dummy nodes”. In: *Experimental and Efficient Algorithms* (2004), pp. 570–583. ISSN: 03029743.

[Toe14] Tibor Toepffer. “Schöne Kurven: Ebenenbasiertes Kantenrouting mit Splines”. Diploma thesis. Kiel University, Department of Computer Science, Nov. 2014.

[Tut63] William Thomas Tutte. “How to draw a graph”. In: *Proceedings of the London Mathematical Society*. Vol. 3. 1963, pp. 743–768.

[VH81] Paul F. Velleman and David C. Hoaglin. *Applications, basics, and computing of exploratory data analysis*. Duxbury Press, 1981. ISBN: 0-87150-409-X.

[Wal90] John Q. Walker, II. "A node-positioning algorithm for general trees". In: *Software – Practice and Experience* 20.7 (1990), pp. 685–705. ISSN: 0038-0644.

[WM96] Xiaobo Wang and Isao Miyamoto. "Generating customized layouts". In: *Graph Drawing*. Ed. by Franz Brandenburg. Vol. 1027. LNCS. Springer Berlin / Heidelberg, 1996, pp. 504–515. ISBN: 978-3-540-60723-6. DOI: 10.1007/BFb0021835.

[WMS10] Michael Wybrow, Kim Marriott, and Peter J. Stuckey. "Orthogonal connector routing". In: *Proceedings of the 17th International Symposium on Graph Drawing (GD '09)*. Vol. 5849. LNCS. Springer, 2010, pp. 219–231. DOI: 10.1007/978-3-642-11805-0.

[WMS12] Michael Wybrow, Kim Marriott, and Peter J. Stuckey. "Orthogonal hyperedge routing". In: *Proceedings of the 7th International Conference on Diagrammatic Representation and Inference (Diagrams'12)*. Vol. 7352. LNAI. Springer, 2012, pp. 51–64. DOI: 10.1007/978-3-642-31223-6.

[WPC+02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. "Cognitive measurements of graph aesthetics". In: *Information Visualization* 1.2 (2002), pp. 103–110. ISSN: 1473-8716.

[WWS+17] Y. Wang, Y. Wang, Y. Sun, L. Zhu, K. Lu, C. W. Fu, M. Sedlmair, O. Deussen, and B. Chen. "Revisiting stress majorization as a unified framework for interactive constrained graph visualization". In: *IEEE Transactions on Visualization and Computer Graphics* PP.99 (2017), pp. 1–1. ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2745919.