Upward Planarization and Layout

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Hoi-Ming Wong

Dortmund
2011

Tag der mündlichen Prüfung:
26.09.2011

Dekanin:
Prof. Dr. Gabriele Kern-Isberner

Gutachter:
Prof. Dr. Petra Mutzel, Technische Universität Dortmund
Prof. Dr. Christoph Buchheim, Technische Universität Dortmund

Abstract

Drawing directed graphs has many applications and occurs whenever a natural flow of information is to be visualized. Given a directed acyclic graph (DAG) G, we are interested in an upward drawing of G, that is, a drawing of G in which all arcs are drawn as curves that are monotonically increasing in the vertical direction. Besides the upward property, it is desirable that the number of arc crossings arising in the drawing should be minimized.

In this thesis, we propose a new approach for drawing DAGs based on the idea of upward planarization. We first introduce a novel upward planarization approach for upward crossing minimization of DAGs that utilizes new ideas for subgraph computation and arc reinsertion. In particular, it is the first upward crossing minimization algorithm which does not utilize any layering techniques known from the framework by Sugiyama et al. [STT81] or from the upward planarization algorithm by Eiglsperger et al. [EKE03].

Our approach addresses the main weakness of the classical two step upward crossing minimization approaches, where in the first step a layering of the input DAG is computed and then, in the second step, the number of crossings is minimized by solving the so-called k-level crossing minimization problem (k-LCM). However, choosing an inappropriate layering in the first step can negatively effect the subsequent k-level crossing minimization step, thus causing many unnecessary arc crossings.

As shown by experimental evaluations, our new approach—referred to as *layer-free upward crossing minimization* (LFUP)—outperforms the state-of-the-art crossing minimization heuristics based on layering even if an exact algorithm for k-level crossing minimization is used. Furthermore, LFUP also outperforms the existing approaches following the idea of upward planarization, that is, the approaches by Di Battista et al. [BPTT89] and Eiglsperger et al. [EKE03].

We also present two extensions for the new approach: an extension for upward planarization of directed hypergraphs and an extension for handling given port constraints, that is, drawing constraints that arise due to the prescribed positions where arcs can be connected to the drawing of the nodes.

The upward planarization approach LFUP computes an upward planar representation (UPR) $\mathcal{R}$ of the input graph G, where crossings are modeled by dummy nodes. We introduce a new layout approach for realizing UPRs, that is, a drawing algorithm for constructing upward drawings where the arc crossings arising in the drawing are the ones modeled by the dummy nodes in $\mathcal{R}$. Only few algorithms exist for realizing UPRs, most of these algorithms are based on simple ideas and originally developed for drawing planar st-graphs, hence our layout approach constitutes the first approach specialized for realizing UPRs. It offers two main advantages over the popular Sugiyama framework: It benefits from the advantage of the upward planarization approach LFUP, thus producing upward drawings with significantly less arc crossings.

Therefore, the drawing quality increases considerably. Furthermore, while the upward drawings produced by layer-based drawing approaches are often unstructured and appear unnaturally flat, the new layout approach constructs upward drawings that better reflect the structure of the digraphs and give a tidier impression to the viewer.

Zusammenfassung

Die Visualisierung von gerichteten azyklischen Graphen (DAGs) gehört zu den wichtigsten Aufgaben im *automatischen Zeichnen von Graphen*. Hierbei suchen wir für einen gegebenen DAG G eine Zeichnung von G (Aufwärtszeichnung von G genannt), sodass alle Kanten als Kurven streng monoton in vertikaler Richtung steigend gezeichnet werden. Um die Lesbarkeit der Zeichnung zu erhöhen, sollte neben der Aufwärtseigenschaft auch die Anzahl der Kantenkreuzungen in der Zeichnung möglichst gering sein.

In dieser Dissertation entwerfen wir einen neuen Ansatz zur Visualisierung von gerichteten Graphen, der auf der Idee der Aufwärtsplanarisierung basiert. Wir stellen zuerst ein innovatives Aufwärtsplanarisierungverfahren vor, das neue Techniken für die Berechnung aufwärtsplanare Untergraphen und die anschließende Kanteneinfügephase einsetzt. Vor allem werden in dem neuen Verfahren keine Schichtungstechniken zur Kreuzungsminimierung benutzt, wie wir sie aus dem Zeichenverfahren von Sugiyama et al. [STT81] oder aus dem Aufwärtsplanarisierungsverfahren von Eiglsperger et al. [EKE03] kennen. Die Festlegung einer Schichtung kann nämlich zu sehr schlechten Ergebnissen führen. Folglich besitzt das neue Verfahren nicht die Nachteile der bisherigen Kreuzungsminimierungsverfahren.

Experimentellen Analysen zeigen, dass das neue Aufwärtsplanarisierungsverfahren deutlich bessere Ergebnisse liefert als das klassische, auf Schichtungen basierende Kreuzungsminimierungsverfahren, und dies unabhängig von den benutzten Lösungsansätzen (heuristisch oder optimal) für die k-level Kreuzungsminimierungsphase. Auch im Vergleich mit den bekannten Aufwärtsplanarisierungsverfahren (Di Battista et al. [BPTT89] und Eiglsperger et al. [EKE03]) zeigt sich, dass der neue Ansatz weitaus bessere Ergebnisse liefert. Wir stellen auch zwei Erweiterungen des neuen Ansatzes vor: eine Erweiterung zur Aufwärtsplanarisierung von gerichteten Hypergraphen und eine zur Unterstützung von Port Constraints.

Das Ergebnis der Aufwärtsplanarisierung ist eine aufwärtsplanare Repräsentation (UPR) — ein eingebetteter DAG, in dem Kreuzungen durch künstliche Dummy-Knoten modelliert werden. Wir stellen ein Layoutverfahren zur Realisierung solcher UPRs vor, d.h., ein Verfahren, das aus einem UPR eine Aufwärtszeichnung konstruiert, sodass die Kantenkreuzungen in der Zeichnung zu den Dummy-Knoten des gegebenen UPR korrespondieren. Die wenigen existierenden Zeichenverfahren zur Realisierung von UPRs sind sehr einfach und wurden ursprünglich entwickelt, um planare st-Graphen zu zeichnen. Unser neues Verfahren stellt somit das erste Layoutverfahren dar, das speziell im Hinblick auf die Realisierung von UPRs entworfen wurde. Es bietet zwei wichtige Vorteile gegenüber dem etablierten Standardzeichenalgorithmus von Sugiyama et al.: Die Zeichnungen besitzen wesentlich weniger Kreuzungen, was zur deutlichen Verbesserung der Lesbarkeit führt. Ferner sind sie strukturierter und machen einen aufgeräumteren Eindruck.

Acknowledgment

Many people supported me during the time of research. Without their help, this thesis would never have been possible. I wish to express here my gratitude to all of them.

First of all, I want to thank my advisor Prof. Dr. Petra Mutzel who gave me the opportunity to work with her and her research group at TU Dortmund. I am grateful to my co-authors Carsten Gutwenger and Markus Chimani for giving valuable advices regarding upward planarization and layouts and for sharing their experience and knowledge. I also want to thank Miro Spönnemann for the fruitful cooperation and for sharing his knowledge about port constraints and orthogonal drawings. Many thanks go to Nils Kriege and Bernd Zey for proofreading parts of this thesis and Gundel Jankord for doing all the administrative work. I am very grateful to Karsten Klein, the brave fighter against "Kraut und Rüben", who gave me many helpful advices and who sacrificed a lot of free time for proofreading this thesis. I want to thank the members of my defense commission: Prof. Dr. Christoph Buchheim, Prof. Dr. Ernst-Erich Doberkat, Prof. Dr. Petra Mutzel, and Dr. Moritz Martens. Finally, I wish to thank my family and friends for their support.

Hoi-Ming Wong
Dortmund, 2011

Variable	Description
a	arc
$\tilde{a}$	arc to be reinserted
e	arc/edge
f	face
G	graph
$\mathcal{H}$	hypergraph
$\mathcal{L}$	layering
$\mathcal{R}$	upward planar representation
s	source
$\hat{s}$	super source
t	sink
$\hat{t}$	super sink
U	subgraph
u, v	nodes
α	hyperarc
γ	combinatorial embedding
Γ	(upward) planar embedding
ϵ	line segment
Θ	weight function
τ	sink-switch

Notation	Description
$\mathcal{D}$	drawing
$u \rightsquigarrow v$	a path from u to v
$u \not\rightsquigarrow v$	no path exists from u to v
$\|V\|$	cardinality of set V
$X(u)$	x-coordinate of node u
$Y(u)$	y-coordinate of node u
$u \prec v$	$Y(u) < Y(y)$

Abbreviation	Description
BFS	breadth-first search
DAG	directed acyclic graph
DFS	depth-first search
FUPS	feasible upward planar subgraph
ILP	integer linear programming
k-LCM	k-level (multi-level) crossing minimization
LFUP	layer-free upward planarization
SDP	semidefinite programming
sT-graph	single source DAG
UPL	upward planarization layout
UPR	upward planar representation
UIP	upward insertion path

Table 1: Notation

Contents

1 Introduction **1**
1.1 Corresponding Publications 3
1.2 Main Results 5
1.3 Organization of the Thesis 6

2 Preliminaries **9**
2.1 Graphs 9
2.2 Drawings and Embeddings 11
2.3 Upward Planarity 16
2.4 Planarization 17

3 Upward Drawing Algorithms **21**
3.1 Framework by Sugiyama, Tagawa, and Toda 21
3.1.1 Cycle Removal 22
3.1.2 Layer Assignment 23
3.1.3 Crossing Minimization 27
3.1.4 Coordinate Assignment 32
3.1.5 Extensions 34
3.2 Alternative Upward Drawing Algorithms 36
3.2.1 Mixed Upward Planarization 37
3.2.2 Dominance Drawing 38
3.2.3 Visibility Representation 38

4 Upward Planarization **41**
4.1 Introduction 41
4.1.1 Motivation 42
4.1.2 Challenges 43
4.2 Upward Planarization Algorithm 44
4.2.1 Algorithm Overview 44
4.2.2 Feasible Subgraph 46
4.2.3 Arc Reinsertion 50
4.2.4 Runtime Analysis 66
4.3 Experimental Evaluation 68
4.3.1 Benchmark Sets 68

4.3.2 Evaluated Algorithms 70
4.3.3 Comparison 71
4.3.4 Deeper Analysis 74
4.3.5 Runtime 84
4.3.6 Summary 84
4.4 Extension to Port Constraints 86
4.4.1 Chain Substitution 87
4.4.2 Feasible Subgraph 87
4.4.3 Arc Reinsertion 88
4.5 Extension to Hypergraphs 90
4.5.1 Pre-processing 91
4.5.2 Feasible Subgraph and Arc Reinsertion 91
4.6 Example 98

5 Upward Planarization Layout 109
5.1 Introduction 109
5.2 Straight-forward Approach 111
5.3 Upward Planarization Layout 111
5.3.1 Layer Assignment and Node Ordering 113
5.3.2 Post-processing 118
5.3.3 Polyline Hierarchical Layout 120
5.3.4 Experimental Evaluation 125
5.3.5 Orthogonal Layout 139
5.4 Example 147
5.5 Drawing Gallery 150

6 Discussion 163
6.1 Conclusion 163
6.2 Future Works 166

Bibliography 169

Index 179

Chapter 1

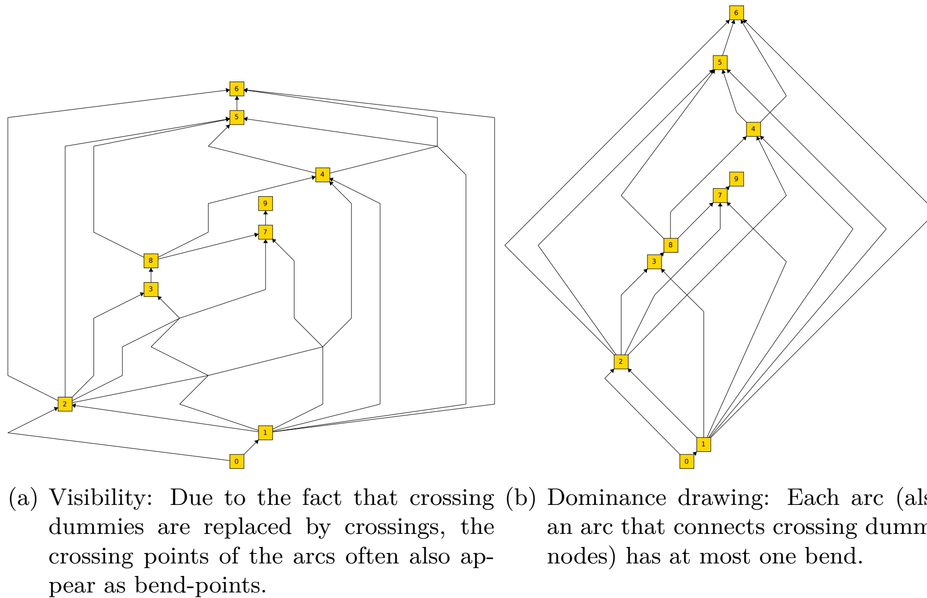
Introduction

Since the publication of the drawing framework by Sugiyama, Tagawa, and Toda in 1981 [STT81], the suggested crossing minimization approach based on layering, that is, first computing a layering of the input graph and then solving the corresponding k-level crossing minimization problem, is a synonym for upward crossing minimizing of directed acyclic graphs (DAGs).

Although two alternative techniques based on the idea of planarization were proposed—the first one by Di Battista, Pietrosanti, Tamassia, and Tollis [BPTT89] and the second one by Eiglsperger, Eppinger, and Kaufmann [EKE03]—they had not managed to establish themselves in practice. One main reason was the poor performance of the first upward planarization approach (see Di Battista et al. [DGL$^+$00]) which produces upward planar representations, that is, upward planar embedded DAGs where arc crossings are modeled by dummy nodes (crossing dummies), with too many crossing dummies.

Another reason was the absent of sophisticated algorithms for constructing upward drawings based on a given upward planar representation. Existing algorithms for this task are slightly modified upward drawing algorithms which were originally developed for visualizing planar st-graphs, for example, the *dominance* drawing approach suggested by Di Battista, Tamassia, and Tollis [DTT92] or the *visibility* drawing approach published by Rosenstiehl and Tarjan [RT86] and independently by Tamassia and Tollis [TT86]. These algorithms are based on simple topological numbering of the nodes and the modification is limited to the replacement of the images of the crossing dummies by arc crossings after the complete upward planar representation has been drawn; for an example, see Figure 1.1. Unsurprisingly, these drawing approaches are by far not competitive in comparison to the well known and widespread drawing framework by Sugiyama et al.

The negative impression of the upward planarization idea regarding upward crossing minimization reported in [DGL$^+$00] was corrected by Eiglsperger et al. [EKE03]. They suggested an approach called *mixed upward planarization* and showed that upward planarization can outperform the com-



(a) Visibility: Due to the fact that crossing dummies are replaced by crossings, the crossing points of the arcs often also appear as bend-points.

(b) Dominance drawing: Each arc (also an arc that connects crossing dummy nodes) has at most one bend.

Figure 1.1: Limitations of the visibility and the dominance drawing approaches.

mon crossing minimization heuristics based on layering, hence give a hint regarding the true potential of the upward planarization idea in visualization of DAGs. However, their approach has some limits: the basic idea of the upward planar subgraph computation step is quite simple and the edge insertion algorithm uses a layering technique which may reduce the number of possible insertion paths, in particular insertion paths causing few arc crossings may not be considered.

In view of these facts, the main goal of this thesis is to develop a new drawing algorithm for drawing digraphs (and directed hypergraphs) based on the idea of upward planarization. For this, we have to face the following challenges in order to overcome the drawbacks of the existing approaches:

1. Develop an upward planarization approach which can exploit the potential of the upward planarization idea.

2. Develop a layout approach tailored to the construction of upward drawings based on a given upward planar representation.

The goal is motivated by the fact that drawing DAGs/directed graphs has many applications and occurs whenever a natural flow of information is to be visualized. Application domains for drawing DAGs are for example the following:

Bioinformatics: The *systems biology graphical notation* (SBGN) [NHM+09] is a standardized graphical representation developed for describing com-

plex biological networks and processes. The underlying graph of many SBGN diagrams, in particular the *activity flows*, is a directed graph.

Figure 1.2(a) gives an example of an activity flow diagram where a *signaling pathway* is illustrated.

Hardware engineering: Visualizing directed graphs arises in nearly all modern hardware development tools, for example, Simulink[1], ASCET[2], or LabVIEW[3]. Such tools offer a graph editor for modeling hardware systems. The underlying graphs of the models are mostly either directed graphs or directed hypergraphs. Since the modeled objects are chips or electric components, arcs connecting the objects are often given prescribed connecting positions, called *ports*.

Figure 1.2(b) gives an example drawing of a model visualized by Simulink.

Software engineering: The *Unified Modeling Language* (UML) is a modeling language for specification, construction and documentation of software. Automatically drawing UML diagrams is a typical example where graph drawing algorithms are to be applied.

Figure 1.2(c) gives an example of a hierarchical UML class diagram.

Business process modeling: Directed graphs are used for modeling typical business processes like workflows, communication or control processes.

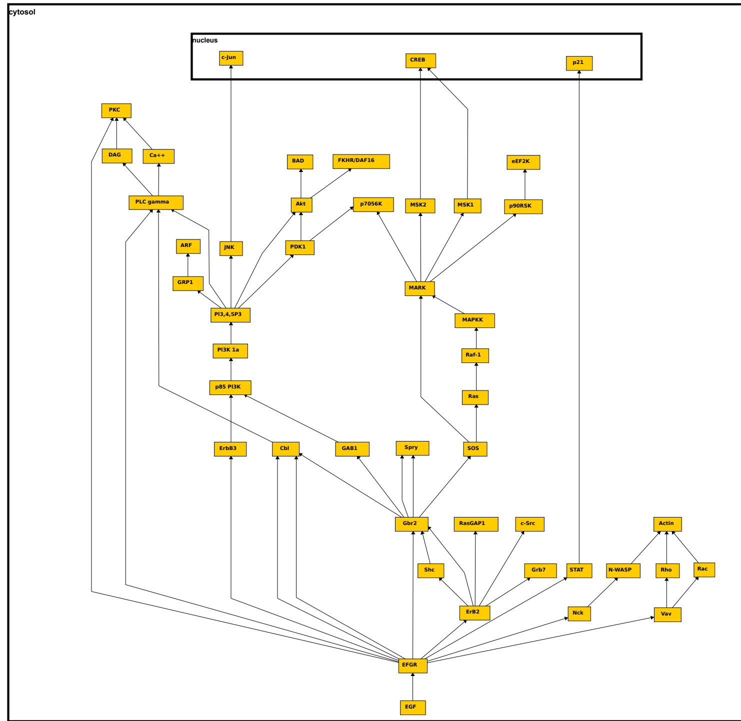
This list of application domains and the corresponding examples can be arbitrarily extended and illustrates that graph drawing algorithms, and in particular algorithms for drawing directed graphs, are widely used in practice.

1.1 Corresponding Publications

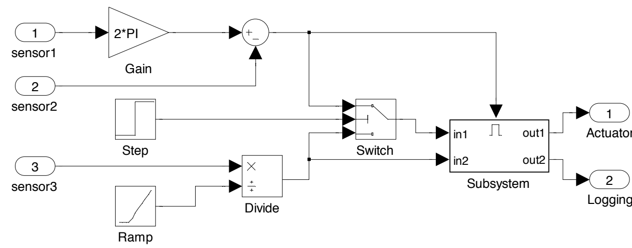
The results presented in this thesis have been partially published in conferences and journals. In the following we list these publications with references to the corresponding chapters and sections of this thesis.

- The layer-free upward planarization algorithm introduced in Sections 4.1–4.3 was first published in the conference proceeding of the *Workshop on Experimental Algorithms* (WEA) 2008 [CGMW08]. An extended version was published in the *Journal of Experimental Algorithmics* (JEA) 2010 [CGMW10a].

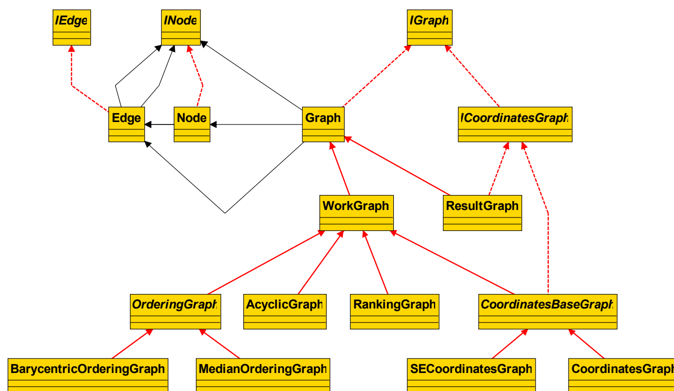
[1] http://www.mathworks.com/products/simulink
[2] http://www.etas.com
[3] http://www.ni.com/labview



(a) An underlying directed graph of an activity flow diagram from [MSLN+09].



(b) A model of a hardware component drawn by Simulink (Spöneman [Spö09]).



(c) A hierarchical UML class diagram (www.oreas.com).

Figure 1.2: Application examples.

- The upward planarization layout approach introduced in Chapter 5 was published first in the conference proceeding of *Graph Drawing* (GD) 2009 [CGMW10b]. An extended version was published in the *Journal of Graph Algorithms and Applications* (JGAA) 2011 [CGMW11].

- The extension of the layer-free upward planarization algorithm to port constraints and directed hypergraphs (Sections 4.4–4.5), as well as the layout algorithm for digraphs and directed hypergraphs with port constraints (Chapter 5) were published in the conference proceedings of *Graph Drawing* (GD) 2010 [CGM$^+$10].

1.2 Main Results

We introduce in this thesis a novel approach for upward crossing minimization of directed acyclic graphs based on the idea of upward planarization. It is the first approach for upward crossing minimization that computes a so-called upward planar representation without to adopt any layering techniques. The conducted experiments reveal, that the new approach outperforms *all* known upward crossing minimization heuristics on the used benchmark sets, including the state-of-the-art crossing minimization heuristics based on layering, the *mixed upward planarization* by Eiglsperger et al. [EKE03] and the *grid sifting* approach by Bachmaier et al. [BBG11]. For example, in comparison to the classical layered crossing minimization approach (based on longest path layering and the barycenter heuristic) and grid sifting, the new approach achieved up to 90% and 70% crossing reduction, respectively. Hence it can be considered as a new state-of-the-art upward crossing minimization heuristic.

We also develop two extensions for the new approach: an extension to directed hypergraphs and to port constraints. To the best of our knowledge, it is the first approach for upward planarizing directed hypergraphs with port constraints.

Directed graphs are usually drawn with the framework by Sugiyama et al. Alternative drawing approaches are rare and most of them focus on special classes of directed graphs. In this thesis, we also propose a layout approach for drawing directed graphs and hypergraphs based on a given upward planar representation. It is the first approach specialized for drawing these classes of graphs with given prescribed port positions. In comparison to the Sugiyama drawing framework, the new layout approach not only produces drawings with much fewer arc crossings, but the drawings also have better drawing characteristics, that is, the drawings are structured and give a tidier impression to the viewer; for example, see Figure 1.3.

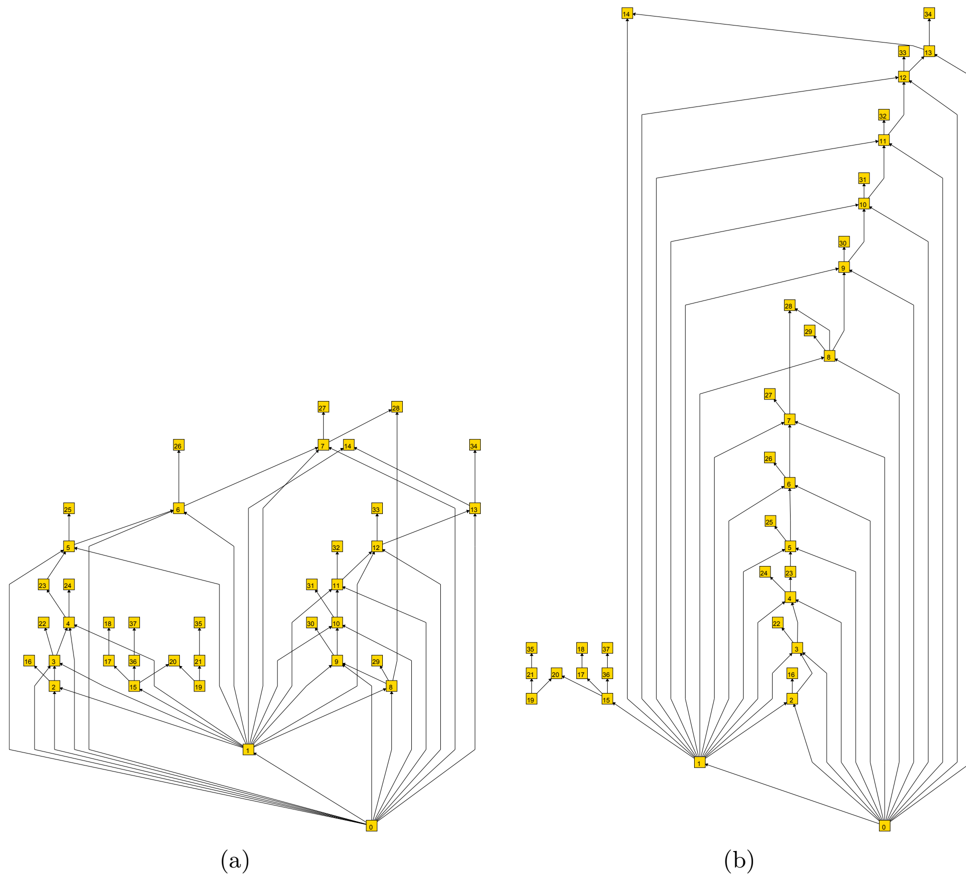


Figure 1.3: Two upward drawings of instance g.38.21 (North DAGs): (a) A drawing with 27 crossings produced by the drawing framework by Sugiyama et al. (b) A drawing with one crossing produced by the new upward planarization drawing approach (see Chapter 5). In contrast to (a) the drawing offers clarity, is well structured, and tidy. Obviously, the drawing in (b) reflects some characteristics of the graph which are completely missing or hard to detect in (a).

1.3 Organization of the Thesis

We start with an introduction to the basics and notations required for this thesis (Chapter 2). The upward planarization approach proposed in this thesis is based on the idea of planarization. We give a sketch of this approach in the last section of Chapter 2.

In Chapter 3, we give an overview of the main solutions for each individual step of the drawing framework by Sugiyama et al. In particular, we focused on the heuristics for the k-level crossing minimization problem. We also briefly describe some extensions for the framework: the extension to hypergraphs and the extension to port constraints. At the end of Chapter 3, we depict the mixed upward planarization approach by Eiglsperger et al. and describe

two alternative approaches for upward drawing DAGs: the dominance and the visibility drawing approach. Both approaches can be used for visualizing upward planar representations.

Chapter 4 is dedicated to the problem of upward planarization. We propose a new approach referred to as *layer-free upward planarization* for upward planarization of DAGs. In Section 4.1, an introduction to upward crossing minimization is given, where we also explain the motivations for developing a new approach and depict the arising challenges. After an overview of the whole upward planarization algorithm (Section 4.2.1), we describe in detail how to tackle the two main problems, namely the feasible subgraph (Section 4.2.2) and the arc reinsertion problem (Section 4.2.3). In Section 4.2.4, we theoretically analyze the runtime of the new approach and in Section 4.3, we evaluate the layer-free upward planarization approach by experiments. We then show how to extend the new approach for handling port constraints (Section 4.4) and how to upward planarize directed hypergraphs (Section 4.5). In the last section of the Chapter 4, we give an illustrated example of the new upward planarization approach.

The final outcome of the layer-free upward planarization approach is an upward planar representation. In Chapter 5, we depict an approach called *upward planarization layout* for constructing hierarchical layouts based on upward planar representations. After an introduction in Section 5.1, we introduce a straight-forward approach for constructing upward drawings (Section 5.2). In Section 5.3.1, we depict how to improve the straight-forward approach, and in Section 5.3.3, we explain a layout approach for constructing upward drawings based on a given layering. The new approach is then experimentally evaluated (Section 5.3.4). We compare it with the alternative dominance and visibility drawing approaches and with the traditional Sugiyama drawing framework. We also introduce a layout approach for orthogonal upward drawings (Section 5.3.5). In Section 5.4, we illustrate the new layout algorithm by an example and in the last section of Chapter 5, we give some upward drawings produced by the new and by the Sugiyama drawing framework.

In the final chapter, we give a conclusion of the achieved results and discuss possible future works.

Chapter 2

Preliminaries

This chapter introduces the required theoretical basics and notations for this thesis. Section 2.1 provides elementary definitions related to graphs. The basics of *graph drawing* are described in Section 2.2. Upward planarity is a crucial property that plays an important role in upward drawing digraphs. Backgrounds and basics regarding this property are introduced in Section 2.3. The last section of this chapter is dedicated to the planarization approach by Batini, Talamo, and Tamassia [BTT84].

The following bibliographical sources are used for this chapter: the textbook by Diestel [Die05]; the textbook by Di Battista, Eades, Tamassia, and Tollis [DETT99] and the textbook by Kaufmann and Wagner [KW01].

2.1 Graphs

Undirected graph. A *graph* is a pair $G = (V, E)$, where V is a finite set of nodes and E is a set of pairs (u, v) with $u, v \in V$. An element $e = (u, v)$ of E is called an *edge* and u and v are called the *end nodes* of e. The edge e is a *self-loop* if $u \equiv v$. An edge e is *incident* to a node u if u is an end node of e. u is *adjacent* to v if $(u, v) \in E$. The *degree* of u is the number of edges incident to u. If x is a element of G, that is, $x \in V$ or $x \in E$, then we denote $x \in G$.

Path and cycle. A *path* $p = \langle v_0, \ldots, v_k \rangle$ in a graph $G = (V, E)$ is a non-empty sequence of mutually distinct nodes of G such that $(v_i, v_{i+1}) \in E$ with $0 \leq i \leq k - 1$. Instead of a sequence of mutually distinct nodes, a path can also be described by a sequence of mutually distinct edges $p = \langle (v_0, v_1), \ldots, (v_{k-1}, v_k) \rangle$. A *cycle* is a path p with the additional edge (v_k, v_0) that connects the last node with the first node of p. A graph which does not contain any cycle is called a *acyclic graph.* A *length function* $\ell : E \to \mathbb{R}$ assigns for each edge of E a length. The *length* of a path p is the sum of the lengths of the edges in p.

Subgraph and component. A graph $U = (V', E')$ is an *(induced) subgraph* of a graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$ and E' contains all the edges $(u, v) \in E$ with end nodes $u, v \in V'$.

$G = (V, E)$ is *connected* if for any pair of nodes $\{u, v\}$, there is an undirected path from u to v in G. A maximal connected subgraph of G is a *component* of G. G is k-connected with $k \in \mathbb{N}$, if and only if $|V| > k$ and for every subset $X \subseteq V$ with $|X| < k$, the graph obtained from G by deleting nodes of X is still connected. A maximal 2-connected subgraph of G is also called a *block* of G.

Tree and forest. An undirected acyclic graph is called a *forest* and a connected forest is called a *tree*. The *leaves* of a tree $\mathcal{T}$ are the nodes with degree one. Nodes with degree greater than one are called inner nodes of $\mathcal{T}$.

In the following we assume that a graph is connected unless otherwise noted.

Directed graph. A *directed graph* or *digraph* is a pair $G = (V, A)$, where V is a finite set of nodes and A is a set of ordered pairs (u, v) with $u, v \in V$. An element $a = (u, v)$ of A is called an *arc* and u is the *source node* and v is the *target node* of a. $a = (u, v)$ is called an *outgoing arc* of u and an *incoming arc* of v. The *in-degree* of u is the number of incoming arcs and the *out-degree* of u is the number of outgoing arcs of u. A node $u \in G$ is a *sink* of G if its out-degree is zero and u is a *source* of G if its in-degree is zero. The (undirected) *underlying graph* of G is the graph $G' = (V, E)$, where E is obtained by considering the arcs of A as undirected edges.

A *directed tree* $\mathcal{T}$ is a DAG such that its underlying graph is a undirected tree.

Analogously to the definition of undirected paths and cycles, a *directed path* and *directed cycle* in a digraph G can be defined. We say u *dominates* v, if there exists a directed path in G from node u to node v. We denote this by $u \rightsquigarrow v$ and use $u \not\rightsquigarrow v$ if u does not dominate v, that is, no path exists in G from u to v. An arc $a = (u, v)$ dominates a node w or another arc $a' = (u', v')$ if v dominates w or u', respectively.

Multigraph. $G = (V, E)$ is a *multi-graph* if E is a *multi-set*, that is, there is at least one element $e \in E$ with occurrence greater than one. e is called a *multi-edge*. G is called a *simple graph* if it contains no multi-edges. Analogously, a *directed multi-graph* can be defined.

***sT*-graph and *st*-graph.** An important role in this thesis plays the class of *sT-graphs*. sT-graphs are connected, directed acyclic graphs (*DAGs*) with exactly one source s. In such digraphs, s dominates all nodes of G. A restricted class of sT-graph are the st-graphs. These are sT-graphs with exactly one sink t and in addition, an st-graphs contains the arc (s, t).

Directed hypergraph. A *directed hypergraph* is a pair $\mathcal{H} = (V, \mathcal{A})$, where V is again a finite set of nodes and $\mathcal{A}$ is a set of pairs (S, T) with non-empty sets $S, T \subseteq V$. The elements of $\mathcal{A}$ are called *hyperarcs*, S are the source nodes, and T the target nodes. Hence, a directed hyperarc can connect several source nodes with several target nodes. While the definition conceptually allows $S \cap T \neq \emptyset$ (that is, a hyperarc may be or contain a self-loop), we will not consider such cases in our thesis.

Now let $\mathcal{H}$ be a self-loop free directed hypergraph and $\alpha = (S, T)$ a hyperarc of $\mathcal{H}$. A directed tree $\mathcal{T} = (V_\alpha, A_\alpha)$ with $V_\alpha = (S \cup T \cup N)$ is an *underlying tree* of α if:

(i) for each source node $s \in S$ there is a node $n \in N$ with $(s, n) \in A_\alpha$;

(ii) for each target node $t \in T$ there is a node $n' \in N$ with $(n', t) \in A_\alpha$;

(iii) the degree of each $v \in S \cup T$ is exactly one within $\mathcal{T}$ and

(iv) each $n \in N$ is only adjacent to vertices of V_α and has degree at least two.

We call N the *hypernodes* of α.

Informally, the source and target nodes are the leaves and the hypernodes are the inner nodes of $\mathcal{T}$. $\mathcal{T}$ is called *confluent* if each source node dominates all target nodes. (This definition is very similar to the definition of Dickerson, Eppstein, Goodrich, and Meng in the context of *confluent drawings* of directed and undirected graphs [DEGM03], where several edges can be merged together to one edge.) Analogously to Chimani and Gutwenger [CG07] (see *tree-based* and *point-based* drawing style of hyperedges), we distinguish two cases: If the cardinality $|N|$ of N is exactly one, then $\mathcal{T}$ is called a *star-based* underlying tree and if $|N| > 1$, $\mathcal{T}$ is called a *tree-based* underlying tree. A trivial observation: While the star-based underlying tree of a hyperarc is unique, the tree-based is not. An example of star-based and tree-based underlying trees is given in Figure 2.1.

A *directed underlying graph* H of a directed hypergraph $\mathcal{H}$ is obtained by substituting each hyperarc α by an underlying tree $\mathcal{T}_\alpha$, that is, H consists of the nodes of $\mathcal{H}$ together with the hypernodes and arcs of all underlying trees. If each α is replaced by its star-based underlying tree, then H is called a *star-based* underlying graph of $\mathcal{H}$. H is *confluent* if all underlying trees are confluent.

2.2 Drawings and Embeddings

Given a graph G, a *drawing* $\mathcal{D}$ of G is a function that maps the nodes to non-overlapping points on a plane surface $\mathcal{S}$ and each edge (u, v) to a simple open curve that connects the corresponding points of u and v in $\mathcal{S}$ and does not cross any other node points.

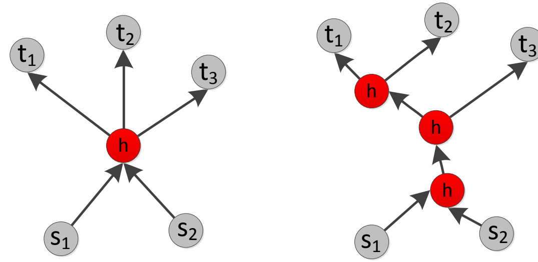


Figure 2.1: Directed underlying trees of a hyperarc: A star-based (left) and a tree-based (right) directed underlying tree of the hyperarc $\alpha = (\{s_1, s_2\}, \{t_1, t_2, t_3\})$. The hypernodes are labeled with "h" and are drawn in red. Both underlying trees are confluent since the sources dominate the targets.

Drawing style. We distinguish between two main types of drawings: planar and non-planar. A *planar drawing* $\mathcal{D}$ is a drawing on the plane where no any two edges intersect each other except at the end-points. Clearly, not every graph admits a planar drawing. A graph that can be drawn in planar fashion is called a *planar graph*. A special case of planar drawings are upward planar drawings: A *(strictly) upward drawing* $\mathcal{D}$ is a drawing of a digraph such that each arc is drawn (strictly) monotonically increasing in the vertical direction. If the drawing is also planar, then $\mathcal{D}$ is (a strictly) an *upward planar drawing*. A digraph which admits an upward planar drawing is called an *upward planar digraph*. It is easy to see that only DAGs can be drawn in upward fashion.

There are different drawing styles for drawing planar and non-planar graphs and each style has its advantages and disadvantages. Some common drawing styles are:

Polyline Drawing: In this style, the edges of the graph are drawn on the plane as polygonal chains.

Straight-line Drawing: The edges are drawn as straight-lines. Straight-lines drawings can be considered as a special case of the polyline style where the images of the edges do not have any bend-points.

Orthogonal Drawing: This style can also be considered as a special case of the polyline style where each arc is drawn as a chain of horizontal and vertical line segments.

Layered/Hierarchical Drawing: The nodes of the graph are assigned to certain layers, and nodes on the same layer are placed on the same horizontal line. This style has become a quasi standard style for drawing digraphs.

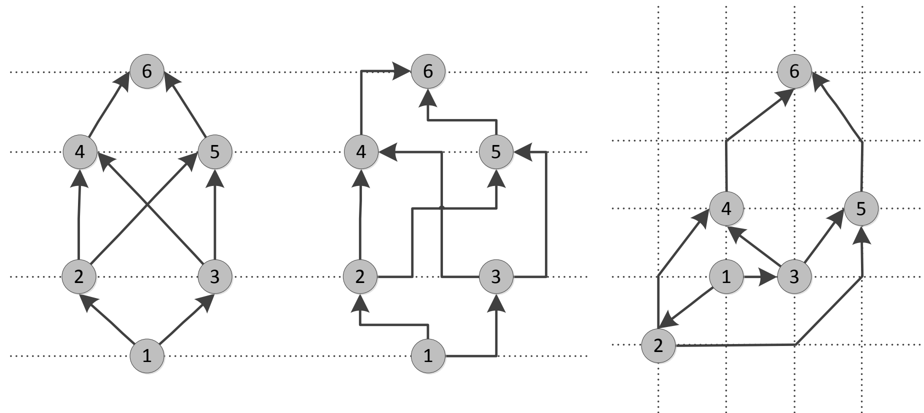


Figure 2.2: Illustration of the different drawing styles and combinations: The three figures show drawings of the same planar but non-upward planar DAG. (left) A non-planar layered straight-line strictly upward drawing. (middle) A non-planar layered orthogonal upward drawing. (right) A planar but non-upward polyline grid drawing.

Grid Drawing: In a grid drawing, the coordinates of the nodes, bend-points, and edge crossings are integers.

Figure 2.2 gives an illustration of the different drawing styles and combinations.

Aesthetic criteria. One main task in graph drawing is to find algorithms that draw graphs "nicely" . Whether a graph is drawn nicely or not depends on many factors and on the personal, therefore subjective, interpretation of "nice". Nonetheless, a drawing of a graph should reflect domain specific requirements and should help the reader easier to remember and understand the visualized information. Although there is no clear formal definition of "nice drawing", it turns out that there are several aesthetic criteria which can be used to characterize a nice drawing (see Purchase, Cohen, and James [PCJ96] or Purchase [Pur97]). Some of the commonly accepted criteria are:

crossings: A drawing with too many edge crossings may appear cluttered and hence makes difficult for the reader to track the edges. This is in particular the case when the drawing area is small. Therefore it is often preferable that the number of edge crossings occurring in a drawing is small. For many applications the number of edge crossings is considered as an important aesthetic criterion.

area: This criteria measures the compactness of a drawing. Since a drawing can scale up or down, this criterion is always related to some given drawing conditions called *resolution rules*. These conditions are, for

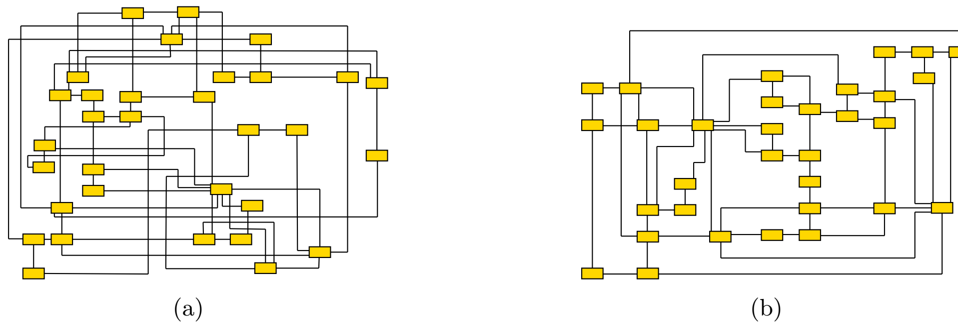


Figure 2.3: Aesthetic criteria and readability: Two orthogonal drawings of the same graph. (a) A drawing with many bend-points, arc crossings, and high average edge length. (b) In contrast to (a), a drawing with fewer arc crossings, bend-points and lower average edge length.

example, the minimal distance of the nodes, minimal edge length or minimal layer distances (for layered drawings). With respect to the given resolution rules, it is desirable that a drawing should be drawn with small area requirement, where the area of a drawing $\mathcal{D}$ can be defined, for example, as the area occupied by the bounding box of $\mathcal{D}$.

edge length: Short edges are desirable since they help the reader to find out easier whether two nodes are connected or not. Further, in a hierarchical drawing long edges can increase the drawing area. This is due to the characteristic of hierarchical drawings, that is, nodes are assigned to layers. Thus, for long arcs that span more than one layer, additional spaces on each spanned layer must be allocated. These spaces are later occupied by the drawings of the long arcs. Allocate enough spaces also allows that the given minimal distance between the arcs can be fulfilled.

There are several common measurements for this criterion: the total sum of the edge lengths, the average edge length, or the maximum edge length.

general direction: Whenever flows are modeled by digraphs, it is preferable that a drawing of the model helps the viewer to recognize the general flow direction easily. Hence, this criterion plays an important role in drawing digraphs.

angular resolution: It is eligible that the angle between two consecutive edges incident on the same node is as large as possible, large angles make the distinction of edges easier.

bends: An edge drawn as a polyline consists of a chain of straight-line segments. The point where two straight-line segments meet are called *bend-points*. An edge drawn with few or no bend-points is easier to follow than an edge drawn with a lot of bend-points. Furthermore, edges

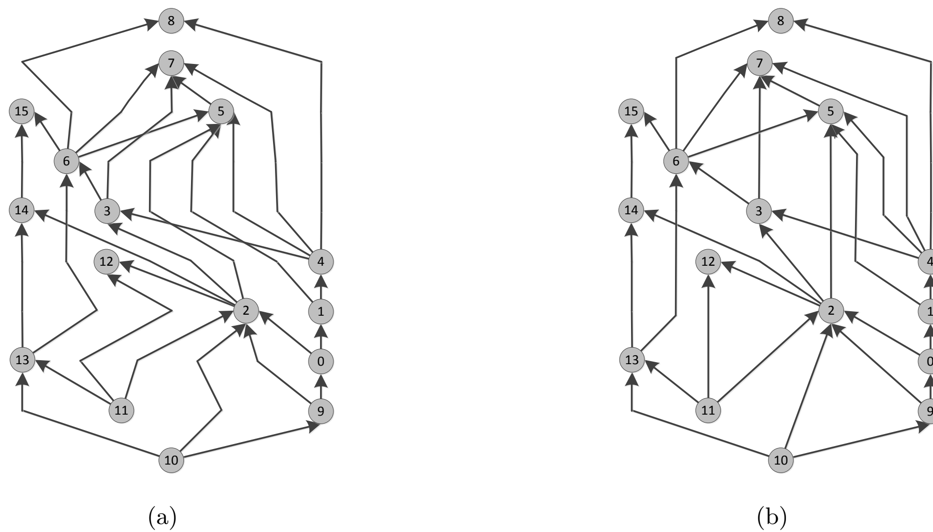


Figure 2.4: (a) "Spaghetti effect" caused by arcs with too many bend-points. (b) A tidier drawing obtained by eliminating some bend-points.

with many bend-points tend to take a zig zag course. This may cause the so-called "spaghetti effect" (see Figure 2.4).

*aspect ratio***:** Given the width w and the height h of a drawing, the aspect ratio is defined as w/h. For some applications it is desirable that the drawings fit to a given screen or printing area, hence also to a certain aspect ratio.

In consideration of these aesthetic criteria, drawing graphs can be regarded as a multi-criteria optimization task. We will see that for some criteria, this task includes several NP-hard problems.

Observe that whether a drawing is good or not strongly depends on the application demands. There may be situations where a compact representation is preferred over a representation with few arc crossings or vice versa. Also a set of numbers, for example, number of arc crossings, height or width may not correctly reflect the quality of a structured and tidy drawing. So manually inspecting the drawings is the best way to evaluate the quality.

Combinatorial and planar embeddings. A planar drawing of G on a plane $\mathcal{S}$ defines for each node a fixed circular order of its incident edges. According to this circular order, each planar drawing $\mathcal{D}$ can be assigned to a certain equivalence class called *combinatorial embedding* γ of G. We say $\mathcal{D}$ *induces* γ. The drawing $\mathcal{D}$ also subdivides $\mathcal{S}$ into regions called *faces*. A region f is an *internal face* if f is bounded by a cycle of edges, called *face-cycle* of f. If f is the outer unbounded region, then f is called the *external*

face of $\mathcal{D}$. Since $\mathcal{D}$ induces γ, we also say f is a face of γ and write $f \in \gamma$. In the following we use face and face-cycle as synonym if it is non-ambiguous. If an element x, for example, a node or an arc, is an element of the face-cycle f then we denote $x \in f$.

We observe that the definition of combinatorial embedding does not fix the external face. Thus, in contrast to a combinatorial embedding, a *planar embedding* Γ is defined as a combinatorial embedding γ with a fixed external face. γ is then the *underlying combinatorial embedding* of Γ.

2.3 Upward Planarity

Now let $G = (V, A)$ be a planar DAG and let Γ be a planar embedding of G with the given external face f^*. Γ is an *upward planar embedding* of G if there exists an upward drawing $\mathcal{D}$ inducing Γ such that f^* is the external face of $\mathcal{D}$.

A *sink-switch/source-switch* of a face-cycle f is a node $u \in f$ for which there is no arc of f starting/ending at u (see Figure 2.5).

Since the main goal is to construct an upward drawing of G with as few arc crossings as possible, we are interested in testing whether G admits an upward planar drawing or not. Unfortunately, such a test cannot be performed in polynomial time for general DAGs unless $NP = P$ (Garg and Tamassia [GT01]; Garey and Johnson [GJ90]). However, as shown by Bertolazzi, Di Battista, Mannino and Tamassia [BDMT98], for an important class, namely, the class of single source digraphs, upward planarity testing can be performed in $(|V| + |A|)$. We now give more facts on this class of graphs:

Let G be an sT-graph and Γ an upward planar embedding of G. We can observe that in an upward planar drawing $\mathcal{D}$ inducing Γ, there exists a sink-switch $\tau \in f$ for each internal face $f \in \Gamma$ such that τ is drawn higher than all other nodes incident to f (see Figure 2.5). We call τ the *top-sink-switch* of face f.

Let γ be the underlying combinatorial embedding of Γ. A *face-sink-graph* F of G with respect to γ is a graph such that:

- for each face f of γ, there is a node u_f in F which corresponds to f;
- for each sink-switch τ of γ, there is a node v_τ in F which corresponds to τ;
- graph F has an edge (u_f, v_τ) if τ is a sink-switch in f.

An example of a face-sink-graph is given in Figure 2.5.

As described by Bertolazzi et al. [BDMT98], a face-sink-graph can be used to test whether γ is an underlying embedding of an upward planar embedding Γ in time $\mathcal{O}(|V| + |A|)$. In the positive case, Γ can then be derived from γ by determining a suitable face f of γ as the external face of Γ.

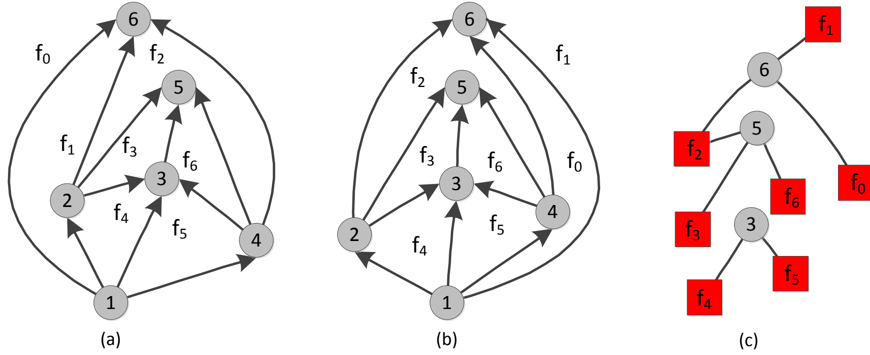


Figure 2.5: Two upward planar drawings of the same sT-graph inducing the upward planar embeddings Γ_a and Γ_b, respectively. Notice that Γ_a and Γ_b have the underlying combinatorial embedding γ in common. (b) The face-cycle of $f_2 \in \Gamma_b$ contains four switches: two sink-switches (node 5 and 6) and two source-switches (node 2 and 4). Node 6 is the top-sink-switch of f_2. (c) The face-sink-graph F with respect to γ.

2.4 Planarization

Regarding the aesthetic criterion *crossings*, it is desirable to draw a non-planar graph with minimized number of edge crossings. An approach to achieve this goal is the planarization algorithm suggested by Batini, Talamo, and Tamassia [BTT84] in 1984. It is widely recognized as the most successful heuristic for minimizing edge crossings for undirected graphs (Gutwenger and Mutzel [GM04]). Our approach for upward planarizing DAGs is inspired by their idea of planarization, hence we give here some basic definitions concerning planarization and then depict the individual phases of the planarization approach.

Let $G = (V, E)$ be a graph and $U = (V, E')$ a planar subgraph of G. U is a *maximum planar subgraph* of G if the number of edges of U is the highest among all planar subgraphs of G.

Definition 1 (Insertion Path)**.** An *insertion path* p with respect to some edge $e = (x, y) \in E \setminus E'$ is an ordered list $\langle e_1, \ldots, e_\kappa \rangle$ of edges of U such that the graph U' obtained after *realizing* p is planar. The realization works as follows: We split the edges $e_1, \ldots, e_\kappa$ obtaining the crossing dummy nodes $c_1, \ldots, c_\kappa$, and add the edges $(x, c_1), (c_1, c_2), \ldots, (c_\kappa, y)$ representing e.

Definition 2 (Dual Graph)**.** Let Γ be a planar embedding of a connected graph $G = (V, E)$. The *dual graph* $G^D = (V^D, E^D)$ of G with respect to Γ is a graph that has a node $w_f \in V^D$ for each face $f \in \Gamma$, and an edge $(u_g, v_h) \in E^D$ for each edge $e \in G$ that is on the face-cycle of g and h.

The planarization approach by Batini et al. can be divided into two main phases:

Planarization	
Phase 1	Planar Subgraph Computation
Phase 2	Edge Reinsertion

Planar Subgraph Computation: In this phase a planar subgraph U of G is computed. Following the idea that if few edges have to be reinserted in the subsequent step then few edge crossings can arise, it is eligible that U should be a maximum planar subgraph. However, the *maximum planar subgraph* problem, that is, to decide whether a given subgraph U is a maximum subgraph of G or not, is NP-complete (Garey and Johnson [GJ90]). Although there are exact algorithms to solve this problem, for example, Jünger and Mutzel [JM96], the runtime for large instances is far from being practical, hence heuristics are preferred.

An often used simple heuristic for computing planar subgraphs can be depicted as follows: We start with an arbitrary spanning tree U of G and iteratively try to add the edges $e \notin U$ one by one to U. Each time a planarity test is performed on the intermediate graph $U := U \cup e$. We continue with the next edge if U is planar, otherwise e is not added to U. It will be reinserted later in the edge reinsertion phase. As final result, a planar subgraph $U = (V, E')$ and a set of edges $B = E \setminus E'$ is obtained. The runtime of the algorithm is $\mathcal{O}(|E| \cdot |V|)$ since planarity testing can be performed in $\mathcal{O}(|V|)$ (Hopcroft and Tarjan [HT74]) and $\mathcal{O}(|E|)$ edges are iteratively added.

Edge Reinsertion: Obtaining a planar subgraph U and the arc set B, the task in this phase is to reinsert the edges of B one by one into U such that as few edge crossings as possible arise. As proved by Ziegler [Zie00], this task is again NP-hard even when the embedding of U is fixed, hence practical heuristics are used. Most of these heuristics use a dual graph U^D of U as routing network to compute an insertion path for the edges of B. Due to the construction of the routing network, a path p^D in U^D corresponds to an insertion path p for an edge $e \in B$. In addition, when setting the length of each edge of U^D to one, the length of p^D corresponds to the number of edges crossed by p. Hence, a short path in U^D corresponds to an insertion path for e which causes few edge crossings.

After realizing p in U, an embedded intermediate graph U' is obtained where crossings are represented by dummy nodes (*crossing dummies*).

> We call U' an *intermediate planar representation*. Using U' as starting point, the remaining edges of B are iteratively reinserted. The final result is a *planar representation* of G where crossings are modeled by crossing dummies.

Figure 2.6 illustrates the idea of the planarization approach where the non-planar graph K_6 is planarized.

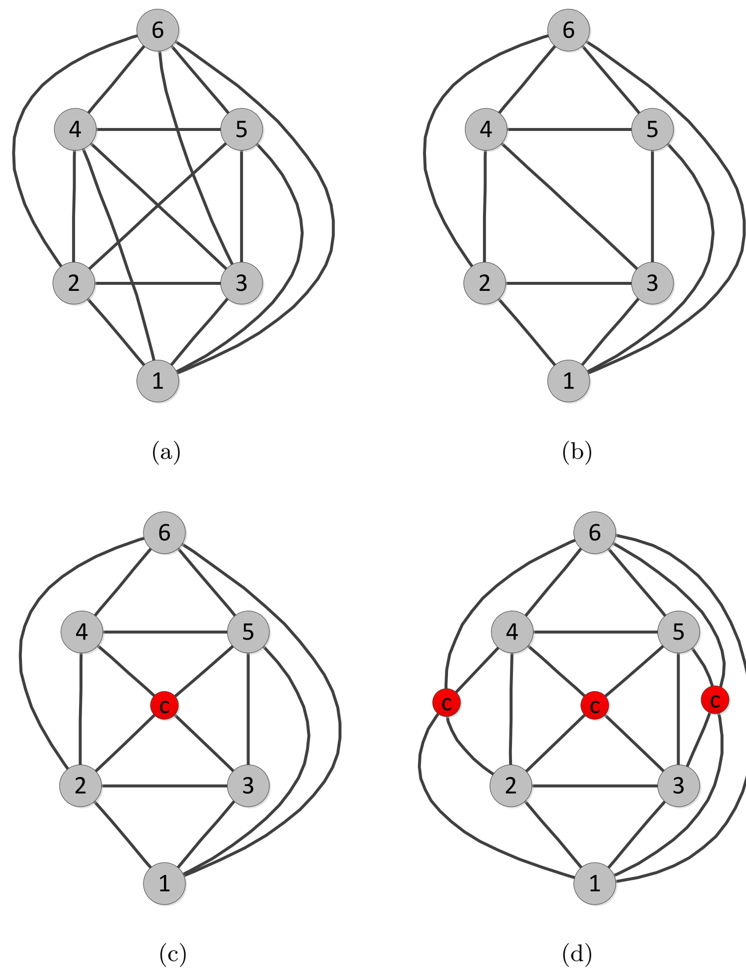


Figure 2.6: Planarization: (a) A drawing of the non-planar graph K_6. (b) A drawing of a planar subgraph of K_6 obtained after deleting the edges $(2,5)$, $(1,4)$, and $(3,6)$. (c) The corresponding intermediate planar representation obtained after realizing the insertion path p for edge $(2,5)$. p crosses only edge $(3,4)$. The crossing is modeled by a crossing dummy (red node labeled with c). (d) The final planar representation of K_6 obtained after reinserting all edges.

Chapter 3

Upward Drawing Algorithms

Drawing digraphs is one of the fundamental issues in graph drawing, having received a lot of attention in the past. Given a digraph G, we ask for a drawing $\mathcal{D}$ of G such that the crossings occurring in $\mathcal{D}$ are minimized. Further, $\mathcal{D}$ should reflect the general flow direction modeled by G. Ideally, the drawing should be drawn upward without any crossings if G is upward planar.

Several algorithms were proposed for drawing digraphs. Without doubt, the most popular one is due to Sugiyama, Tagawa, and Toda [STT81], suggested in 1981, which can be considered as *the* drawing framework for digraphs. We depict this framework in Section 3.1. In particular, we give an overview of the main solutions for each single step of the framework and describe also some published extensions to hypergraphs and to port constraints. In Section 3.2, we consider two alternative approaches for drawing DAGs: the dominance and the visibility drawing algorithm. Both algorithms are restricted to planar *st*-graphs, however they can be easily extended to general DAGs by using upward planarization algorithms. Such an upward planarization algorithm is introduced in the last section of this chapter.

3.1 Framework by Sugiyama, Tagawa, and Toda

The classical framework for drawing digraphs—in the following referred to as `Sugiyama`—was proposed by Sugiyama et al. [STT81]. Since its publication, a vast number of modifications and alternatives for the individual steps have been suggested. Due to the simplicity and universality of the framework—it can draw nearly all kinds of directed graphs—it has become one of the most popular drawing algorithms. The original algorithm produces polyline hierarchical drawings, but there also exist several modifications allowing it to produce layouts in different styles like orthogonal drawings (see Section 3.1.5) or drawings where arcs are drawn as splines (Gansner, Koutsofios, North, and Vo [GKNV93]). The framework can be divided into four main individual steps:

Sugiyama's Framework	
0. Cycle Removal	transform the input digraph G into a DAG
1. Layer Assignment	assign the nodes of G to layers
2. Crossing Min.	reorder the nodes on each layer subject to crossing reduction
3. Coord. Assignment	assign coordinates to nodes and bend-points

In the following subsection we concentrate on some popular solutions.

3.1.1 Cycle Removal

The cycle removal step can be considered as a pre-processing step where a general digraph G is transformed into a DAG. A common method for this task is to compute a *feedback arc set* F, that is, a set of arcs, such that by reversing the arcs of F, G becomes acyclic. The original direction of these arcs will be restored in the last step of the framework.

Regarding the aesthetic criterion *general direction*, it is eligible that the cardinality of F should be the smallest among all feedback arc sets of G. However, finding such a set for general digraphs[1] leads us to the NP-hard *minimum feedback arc set* problem (Garey and Johnson [GJ90]). Several approaches, both heuristic and exact, were suggested to tackle this problem:

DFS: The depth-first search heuristic partitions the arcs of G into forward-, back-, cross- and tree-arcs. By reversing the back-arcs, G becomes a DAG. Yet, this heuristic delivers unsatisfying results (Eades and Sugiyama [ES90]), that is, the size of the feedback arc set is unnecessary large. On the other hand, it is simple and fast (runtime $\mathcal{O}(|V| + |A|)$).

Greedy: Eades, Lin and Smyth [ELS93] proposed a greedy heuristic which also has runtime $\mathcal{O}(|V|+|A|)$. The approach is quite simple and can be sketched as follows: Given a sequence $S = \langle v_1, \dots, v_n \rangle$ of the nodes of G, an arc $a = (v_i, v_j)$ is a *forward-arc* if $i < j$ and a is a *backward-arc* if $j > i$. The set of backward-arcs corresponds to a feedback arc set. Thus the task is to find a sequence such that the number of the backward-arcs is minimal.

The idea of the greedy heuristic is very intuitive. It greedily puts nodes with high out-degree and low in-degree at the begin and nodes with high in-degree and low out-degree at the end of the sequence S (see Figure 3.1). The authors proved that for graphs with no *two-cycles*,

[1] For planar digraphs this problem is solvable in polynomial time (Lucchesi [Luc76]).

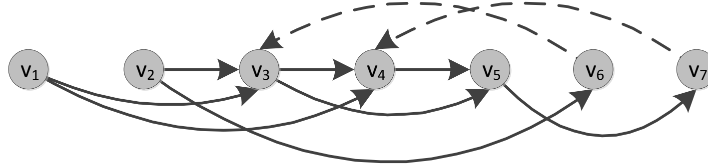


Figure 3.1: Heuristic `Greedy`: The nodes of the graph are placed on a horizontal line and then are ordered such that the number of backward-arcs (dashed line) is small.

that is, a cycle $C = \langle u, v, u \rangle$, the cardinality of the computed feedback arc set does not exceed $\frac{|A|}{2} - \frac{|V|}{6}$. Due to its good practical results (Eades, Lin and Smyth [ELS89]), simplicity, and speed, the heuristic is widely used for the cycle removal step.

We refer to the greedy approach in the following as `Greedy-CR`.

Optimal: Grötschel, Jünger and Reinert [GJR85] considered the *maximum acyclic subgraph* problem which is strongly related to the minimum feedback arc set. The results of their study on the polytopes associated with the maximum acyclic subgraph problem can be used for a branch-and-cut program. Further, the authors also showed: The maximum acyclic subgraph problem is closely related to the linear ordering problem. Therefore exact algorithms for the linear ordering problem, for example, Jünger and Mutzel [JM97], can be used to solve the minimum feedback arc set.

On the one hand, it is desirable to have a small feedback arc set, but on the other hand, in most real world applications, the feedback arcs are given or are easily identified due to the semantic of the models, for example, the feedback arcs are associated with signal feedback-loops. Moreover, according to Gansner et al. [GKNV93], most real-world digraphs have a natural flow direction even when they contain cycles and by removing some improper arcs, the final drawing may be disturbed. Due to this fact, Gansner et al. suggest to use simple and fast approaches like DFS-based heuristics or `Greedy-CR` instead of using complex algorithms.

3.1.2 Layer Assignment

A *layering/leveling* of a DAG $G = (V, A)$ is a partition $\mathcal{L} = \langle L_1, \ldots, L_k \rangle$ of V such that for each arc $a = (u, v)$ with $u \in L_i$ and $v \in L_j$, $i < j$ holds. L_i is called *layer/level* and G a *layered/leveled DAG* with respect to $\mathcal{L}$. The span of a is defined as $j - i$. If each arc of $\mathcal{L}$ has a span of one, then $\mathcal{L}$ is a *proper*

layering. Otherwise a proper layering can be achieved by splitting each arc a with span greater than one by introducing *long arc dummies*[2] on each layer between i and j. Then the arc a consists of a chain of *subarcs*. Note that a proper layering is required for the crossing minimization step.

The *height* of $\mathcal{L}$ is the number of layers of $\mathcal{L}$, thus k. The *width* of a layer L_i is the number of *regular nodes*, that is, original nodes of G, and long arc dummies. The *width* of a layering $\mathcal{L}$ is the number of nodes of the widest layer of $\mathcal{L}$. In some publications, the width is defined without taking the dummies into account. But since the drawings of the arcs should have a certain distance to the drawings of some neighboring elements, that is, nodes or arcs, the space occupied by long arcs is significant. The definition reflects this fact.

Let Λ be a horizontal line and let $Y(\Lambda)$ denotes the y-coordinate of Λ. A *hierarchical/layered drawing with respect to* $\mathcal{L}$ is an upward drawing of G such that the drawing of the nodes on layer L_i are placed on the i-th horizontal line Λ_i and $Y(\Lambda_i) < Y(\Lambda_{i+1})$ for $1 \leq i < k$.

Having regard to the aesthetic criteria *area* and *edge length*, the number of long arc dummies, the height, and the width of a layering should be as small as possible. This leads us to a multi criteria optimization task. However, the problem of finding a layering with minimum height when the width is restricted can be reduced to the *multiprocessor scheduling* problem which is known to be NP-hard (Garey and Johnson [GJ90]; Sugiyama, Tagawa, and Toda [STT81]). Lin showed in his PhD thesis that computing a layering such that the height and the number of long arcs dummies are minimized, is also NP-hard [Lin95]. The problem of layering DAGs with restricted width was investigated by Branke, Leppert and Middendorf [BLME02]. They showed that given a fixed integer w, finding a layering with width w is NP-hard. In addition, they also proved that for two given integers w and h, to decide whether a layering exists for a DAG with width w and height h can be done in polynomial time if at most a constant number of nodes are placed on each layer.
In the following we introduce some algorithms concerning layering DAGs:

Longest Path: This classical layering algorithm is based on a longest path algorithm (Eades and Sugiyama [ES90]; Mehlhorn [Meh84]). It first places each node v with zero out-degree on the top most layer L_k and then removes v and its incident arcs from G. Then the algorithm recursively places all nodes with zero out-degree on the next lower layer L_{k-1} and stops if G is an empty graph. The longest path heuristic with runtime $\mathcal{O}(|V|+|A|)$, is quite fast. It computes layerings with *minimum* height, but the algorithm tends to place the nodes to higher layers. Due to this fact, the width of the layering is often unnecessary large (Eades and Sugiyama [ES90]). We denote this heuristic with `LongestPath`.

[2] For the sake of convenience, instead of long arc dummies we often just say *dummies*.

Promotion Layering: Unlike the other algorithms introduced here, promotion layering by Nikolov and Tarassov [NT06] is a post-processing approach which can be applied only on an already obtained layering. It was developed for tackling the problem of unnecessary long arc dummies.

For a given layering $\mathcal{L}$, a single *promotion step* moves a node v from its layer $L(v)$ to a lower layer $L(v) - 1$. If the layer $L(v) - 1$ does not exist, then a new layer is added to $\mathcal{L}$. In the case when the upward property is violated, that is, there is a node u with (u, v) and $L(u) = L(v)$, then the promotion step is recursively applied to u. A promotion step is successful if the total number of long arc dummies is reduced thereafter. The promotion layering heuristic performs for each non-source node a promotion step. If the promotion step is successful, then the heuristic continues with the next node, otherwise the layering before the call of the promotion step is restored. The algorithm stops if no successful promotion step can be applied to the non-source nodes of G.

Although promotion layering in combination with `LongestPath` does not compute a layering with minimum number of dummies, the experimental evaluations of Nikolov and Tarassov reveal quite good results, but compared to an optimal solution (see `GKNV-Layering`) a small gap still exists. Furthermore, promotion layering needs runtime $\mathcal{O}(d|V|(|V| + |A|))$ where d is the number of dummies of $\mathcal{L}$. In practice it is noticeable slower than the optimal solution `GKNV-Layering`.

Coffman-Graham: This heuristic was developed by Coffman and Graham originally for the multiprocessor scheduling problem [CG72], but since this problem is closely related to the problem of finding a layering of a DAG with minimum height (see Sugiyama, Tagawa, and Toda [STT81]), it can also be used for layering DAGs.

The heuristic first computes for each node v a priority with respect to the in- and out-degree of v. In this process, nodes with high in-degree are assigned a high priority. For a given fixed number w, the algorithm "fills" the layers one by one with nodes (starting with a highest prioritized node). If a layer is full, that is, the number of regular nodes is w, then it continues to fill the next empty layer. A layering of G is obtained after all nodes are distributed. Observe that w is the number of regular nodes, otherwise if w gives the number of regular and dummy nodes, then the NP-hardness of layering a digraph with restricted width w would be contradicted.

According to the published results of Lam and Sethi [LS77], the heuristic computes a layering $\mathcal{L}$ such that the height of $\mathcal{L}$ is at least $(2 - \frac{2}{w} h_{min})k$ where h_{min} is the minimal height with respect to w. Although the heuristic gives us some control on the width of the computed layering,

the main drawback is that the approach tends to produce layerings with many dummies which can significantly slow down the crossing reduction step. Another flaw is that the input DAG must not contain transitive arcs. Thus a pre-processing step is necessary to transform G into a transitive arc free digraph. This transformation requires runtime $\mathcal{O}(|V|^2)$ which is also the final runtime.

GKNV: Long arc dummies correspond to the length of the arcs in the final drawing. Further, too many dummies can increase the runtime of the crossing reduction step. Gansner, Koutsofios, North and Vo tackled this problem by an integer linear programming (ILP) approach that computes a layering with the objective to minimize the number of long arc dummies [GKNV93]. It is formulated as follows:

$$\mathrm{L}(u) \geq 1 \quad \text{for each node } u \in V \tag{3.1}$$

$$\mathrm{L}(v) - \mathrm{L}(u) \geq 1 \quad \text{for each arc } (u,v) \in A \tag{3.2}$$

$$\mathrm{L}(v) - \mathrm{L}(u) \geq \delta(u,v) \quad \text{for each arc } (u,v) \in A \tag{3.3}$$

$$\min \sum_{a=(u,v)\in A} \Theta(a)(\mathrm{L}(v) - \mathrm{L}(u)) \quad \text{subject to (3.1) – (3.3)} \tag{3.4}$$

where $L(u)$ denotes the layer u is assigned to, $\delta(u,v)$ the given minimal span of (u,v), and $\Theta(a)$ the weight of the arc a. Usually, the weight of each arc and $\delta(u,v)$ are set to one. The inequations (3.1)–(3.2) ensure a feasible layering and (3.3) ensures that each arc fulfills the given minimal span. The authors show that the relaxed problem of this integer program has an integer solution, therefore an optimal solution can be computed in polynomial time.

In order to attain a better node distribution over the layers, Gansner et al. proposed to apply a post-processing step called *balancing* to the obtained layering. In this balancing step, nodes with the same in- and out-degree are reassigned to alternative layers with low width such that the upward property of the layering is not violated. We refer to the GKNV approach in combination with balancing as `GKNV-Layering`.

As shown by our experimental evaluations, `GKNV-Layering` is quite fast and computes layerings with nearly minimum height. Furthermore, the experimental evaluations of Healy and Nikolov [HN02b] reveal that drawings obtained by `Sugiyama` using `GKNV-Layering` required less drawing area than when using `LongestPath` or the approach by Coffman and Graham.

ILP: Healy and Nikolov investigated the DAG layering problem from the polyhedral point of view. They suggested an ILP [HN02b] and a branch-

and-cut [HN02a] approach for finding a height and width bounded layering with minimum number of dummies. For this, `GKNV-Layering` was used to determine feasible upper bounds for height and width. Although both solutions are not practical due to their runtime, the computed results can be used for testing the quality of existing layering heuristics.

MinWidth/StretchWidth: Based on the longest path algorithm, Nikolov, Tarassov, and Branke developed two heuristics called *MinWidth* and *StretchWidth* which try to minimize the width of a layering [NTB05]. Both approaches not only take into account the individual width of each nodes, but also the width of the dummies. The behavior of the algorithms can be controlled by a set of variables, that is, by choosing appropriate parameters, the number of dummies, the height and the width of a layering can be influenced. In order to improve the results, both algorithms use a modified version of the promotion layering heuristic [NT06] as post-processing step for dummy reduction.

The experimental evaluations of Tarassov et al. [NTB05] reveal that MinWidth with post-processing produces layerings which width is smaller than `GKNV-Layering`, but this advantage is bought dearly by high runtime, more dummies, and layers. Compared to `GKNV-Layering`, StretchWidth also produces layerings with more dummies and layers and moreover, the average width of the layerings is not significant smaller.

We conclude from our study of the published results, that among the described layering algorithms, `GKNV-Layering` computes the best layering in the sense of compactness, runtime, and number of long arc dummies.

3.1.3 Crossing Minimization

Having obtained a proper layering $\mathcal{L}$ of a DAG $G = (V, A)$, the next task is to minimize the arc crossings. Due to the fact that the number of crossings of a hierarchical drawing with respect to $\mathcal{L}$ depends only on the node order, the task is equivalent to finding an appropriate node order for each layer. Ideally, G is *level planar* with respect to $\mathcal{L}$, that is, there exists a crossing-free hierarchical drawing of G with respect to $\mathcal{L}$. This level planar property can be tested in $\mathcal{O}(|V|+|A|)$ as described by Jünger, Mutzel and Leipert in [JLM98]. In [JLM99], they also describe how to obtain a planar embedding for a level planar graph[3].

If $\mathcal{L}$ does not admit a crossing-free hierarchical drawing, then finding an appropriate node order for the layers to minimize the number of crossings leads to the *k-level (multi-level) crossing minimization* problem (*k-LCM*). This problem is NP-hard even for 2-LCM (Garey and Johnson [GJ83]). Regarding the latter problem, we distinguish two cases: the 1-sided and the

[3] It should be noticed here that level planarity testing is not part of the original framework by Sugiyama et al.

2-sided 2-LCM. Let L_1 and L_2 be the two layers of a 2-LCM instance. In *1-sided* 2-LCM, the node order of L_1 is fixed and we ask for a crossing minimizing node order of the non-fixed layer L_2. In *2-sided* 2-LCM, the node order of both layers is not fixed and we ask for a crossing minimizing node order of both layers.

There is a huge amount of publications regarding the k-LCM problem. We briefly sketch the main approaches here:

Layer-by-Layer Sweep: This approach is developed to extend existing 1-sided 2-LCM heuristics to deal with the k-LCM problem (Di Battista, Eades, Tamassia, and Tollis [DETT99]). In a *layer-by-layer sweep*, an initial node order of the first layer L_1 is determined at the beginning. Then a procedure called *up-sweep* is applied. In this up-sweep, the node order on a layer L_i is fixed and an 1-sided 2-LCM heuristic is used for computing the node order on the layer L_{i+1}. This procedure is applied until the up-sweep reaches the top layer L_k. Thereafter, the obtained node order of L_k is fixed and a *down-sweep* which works similar to an up-sweep is performed from L_k to L_1. Up- and down-sweep are alternately applied until the number of crossings induced by $\mathcal{L}$ remains unchanged or a certain number of iterations of up- and down-sweeps is reached.

Barycenter/Median: The barycenter crossing minimization heuristic (Sugiyama, Tagawa, and Toda [STT81]) and the median crossing minimization heuristic (Eades and Wormald [EW86]) are simple and fast heuristics for the 1-sided 2-LCM problem with runtime $\mathcal{O}(|L_2| \log |L_2|)$ and $\mathcal{O}(|L_2|)$ respectively. The barycenter heuristic is based on the intuitive idea, that a node v should be placed near its adjacent neighbors. Starting with an arbitrary initial node order, the barycenter value $Bc(v)$ is defined as follows:

$$\mathrm{Bc}(v) = \frac{1}{\mathrm{Deg}(v)} \sum_{u \in \mathrm{N}(v)} \mathrm{Rank}(u)$$

where $Deg(v)$ denotes the degree of v, $N(v)$ denotes the adjacent nodes of v and $Rank(u)$ the rank of u in the node order of its layer. Observe that $Bc(v)$ is the average of the rank value of the adjacent neighbors of v. Due to this, in some literature the barycenter heuristic is also called an *averaging heuristic*.

Another averaging heuristic is the median heuristic. Also starting with an arbitrary initial order, the median heuristic assigns the median of the ranks of the sorted adjacent neighbors of v to v.

The barycenter and median heuristic determine the final node order of L_2 by sorting the nodes according to their assigned barycenter respectively median value. A random version of barycenter or median can be

easily obtained by using a random initial node ordering combined with a randomized stable sorting algorithm. Based on the random versions a multi-run variant can be derived. In a *multi-run* version, a random heuristic is started several times and after obtaining the results of the runs, the best one is chosen.

As shown by Mutzel and Jünger in [JM97], multi-run variants can give a huge improvement. Their results reveal that on 1-sided 2-LCM, the multi-run barycenter heuristic performs quite well, that is, the results are not far away from the optimum.

We denote a layer-by-layer sweep algorithm based on the barycenter heuristic with `Barycenter`.

Greedy-Switch: The greedy-switch crossing minimization heuristic (Eades and Kelly [EK86]) swaps each pair of neighbored nodes u, v on L_2 when the number of crossings is reduced thereafter. The swap operations are applied to L_2 as long as a crossing minimizing pair can be found. Since there are at most $|L_2|^2$ node pairs on L_2, the runtime is $\mathcal{O}(|L_2|^2)$. Due to the property that it only applies a swap operation if the number of crossings can be reduced, the heuristic is often used as a post-processing step after performing any crossing minimization heuristic (Mäkinen [Mäk90]; Gansner, North and Vo [GNV88]).

We refer to the greedy-switch heuristic as `GreedySwitch`.

Gansner et al. [GKNV93] used a median heuristic for crossing minimization and then applied a layer-by-layer sweep based on `GreedySwitch` as post-processing. They reported that the post-processing step "*typically provides an additional 20-50% reduction in edge crossings*". But on the other hand, some publications [JM97, EK86] reported that the performance of `GreedySwitch` on 1-sided and 2-sided 2-LCM instances is very poor.

Sifting: The sifting heuristic was originally introduced by Rudell for reducing nodes of an ordered binary decision diagrams (OBDDs) [Rud93]. Matuszewski, Schönfeld and Molitor adapt the approach for the 2-LCM problem [MSM99]. The heuristic computes a local crossing minimizing position of a node v on the free layer L_2 under the assumption that the current node order of L_2 is fixed. After node v is placed at the computed position, the heuristic continues with the next node of L_2 by fixing the current obtained node order. The algorithm stops when all nodes of L_2 are placed. As shown by the authors, the results of sifting on 1-sided 2-LCM instances is slightly better (2–2.5% less crossings) than the barycenter heuristic but it is significant slower with runtime $\mathcal{O}(|L_2|^2)$.

The authors also describe an extension of the sifting heuristic called

global sifting for k-level crossing minimization. Let $J = \langle v_1, \ldots, v_n \rangle$ be a list of the nodes of G sorted in descend order of their degree. A *sifting trial* is an application of the sifting heuristic to each node $v_1, \ldots, v_n$ on its corresponding layer. A trial is a failure if the number of crossings cannot be reduced thereafter. Global sifting consists of two steps: In the first step, sifting trials are performed on the nodes of J until failure. Then in the second step, the order of J is reversed and sifting trials are performed again on the nodes of J until failure. These steps are repeated alternately until a certain number of failures is detected.

As reported in [MSM99], the results of global sifting are better than that obtained by `Barycenter` (15–20% less crossings). However, the runtime is also significantly increased, but according to the authors still practical for digraph with small *density* $(=|A|/|V|)$, and size $|V| \leq 100$.

Based on the sifting idea, Bachmaier, Brandenburg, Brunner, and Hübner published a more sophisticated version of global sifting [BBBH10]. Instead of sifting a single node, they suggested to sift *blocks*, where a block is a single node or a maximum connected subgraph of long arc dummies. Compared to the original global sifting heuristic, their version achieves a further crossing reduction by $5\% - 10\%$. In the following we refer to their version of global sifting as `Global-Sifting`.

The idea published in [BBBH10] was extended by Bachmaier, Brunner, and Gleißner. They developed a sifting version called *grid sifting* [BBG11] (in the following referred to as `Grid-Sifting`). While the previous sifting approach allows a block only to change its position on the corresponding layer (horizontal sifting), `Grid-Sifting` allows it in addition to change the assigned layer if the upward property is not violated (vertical sifting). Furthermore, the approach can modify the given layering by introducing new layers, if it is required for vertical sifting, or by deleting unnecessary layers. Due to the fact that many positions are tried in order to find a local optimal position for a block, the runtime of this approach is accordingly high, but the results are very satisfying. As reported by the authors, grid sifting outmatched most k-LCM heuristics including `Barycenter`.

ILP/SDP: In [JM97], Jünger and Mutzel investigated the 2-LMC problem. They gave a branch-and-cut approach for the 1-sided 2-LCM problem that can solve instances with size $|L_2| \leq 60$ optimally. In addition, they also gave an approach for the more difficult 2-sided 2-LMC problem for computing exact solutions. Due to the runtime this approach is only practicable for small instances.

Buchheim, Wiegele and Zheng modeled the 2-sided 2-LCM problem as an quadratic linear ordering problem and use a semidefinite programming (SDP) approach to compute an exact solution [BWZ10]. On dense

or on large instances, the SDP approach is considerably faster than the approach by Jünger and Mutzel suggested in [JM97].

The aforementioned approaches of Jünger et al. and Buchheim et al. only consider the 2-LCM problem. An approach that is not limited to $k = 2$ was suggested by Jünger, Lee, Mutzel and Oldenthal in [JLMO97]. The authors extended an ILP approach for 2-LCM to an approach for k-LCM. However, due to the runtime, the algorithm is far away from practical use.

Healy and Kursik [HK99a] gave a solution for k-LCM based on the ILP formulation of [JLMO97] and on the idea of a so-called *vertex exchange graph*. As reported in [HK99b], they managed to solve some practical relevant instances optimally for the first time.

In [HR10], Hungerländer and Rendl investigated the quadratic linear ordering problem and suggested a solution based on SDP. This SDP formulation can also be used for optimally solving k-LCM instances, since k-LCM can be formulated as a linear ordering problem. As reported by the authors, on 2-LCM the SDP approach is significant faster than the approach published in [BWZ10].

k-LCM was also investigated by Chimani, Hungerländer, Jünger and Mutzel [CHJM11]. They suggested an SDP-based approach for computing optimal solutions and an SDP-based heuristic that gives in practice solutions which are nearly optimal. Furthermore, the authors also compared their SDP approach with a reimplemented ILP approach published by Jünger, Lee, Mutzel and Odenthal [JLMO97]. They reported that both the SDP and ILP approach managed to solve almost all instances of the Rome benchmark set. The experimental evaluations also showed that the SDP approach can solve more instances to optimality than the ILP approach within a given time limit. Besides these results, Chimani et al. evaluated the traditional barycenter crossing minimization heuristic and our new layer-free upward planarization approach introduced in Chapter 4. We will discuss their results in detail in Section 4.3.

k-level Planarization: Due to the fact that k-LCM is hard, Mutzel [Mut97] suggested to consider the k-level planarization problem instead, since level planarity testing of a layered DAG can be performed in linear time (Jünger, Leipert, and Mutzel [JLM98]; Di Battista and Nardelli [BN88]). For $k = 2$, she gave an ILP formulation derived from the 2-level planar graph characterization of Tomii, Kambayashi and Shuzo [TKS77] for computing a *maximum 2-level planar subgraph*. Furthermore, she also suggested a branch-and-cut algorithm for solving practical instances. However, even when solving the 2-level planar subgraph problem optimally, the final number of crossings may be far away from the optimum,

since reinserting the deleted edges into the subgraph may cause many crossings.

Based on the results of [Mut97], Gange, Stuckey and Marriot introduced in [GSM10] a hybrid approach, that is, a combination of k-level crossing minimization and planarization. The authors reported that drawings obtained by using the hybrid approach are considerably better than drawings obtained by either using crossing minimization or k-level planarization alone.

Counting crossings. Crossing counting is often needed for many crossing minimization heuristics. As reported by Waddle and Malhotra, this task can be a bottleneck in overall computational time of `Sugiyama` especially for large instances (Waddle and Malhotra [WM99]). Hence, they suggested a fast algorithm to solve this problem with runtime $\mathcal{O}(m \log m)$. There, m is the number of edges connecting the nodes on layer L_1 and L_2 (2-LCM scenario). However, the algorithm is relatively complex to implement. Due to this fact, Barth, Jünger and Mutzel [BJM02] published a simpler approach with runtime $\mathcal{O}(m \log n)$, where $n = \min\{|L_1|, |L_2|\}$.

3.1.4 Coordinate Assignment

The outcome of the previous step is a proper layering $\mathcal{L}$ with given node order of each layer. The task now is to assign the final coordinates to the regular nodes and the long arc dummies which are considered as bend-points of the corresponding arc. It is desirable that the coordinates are assigned such that the final drawing is compact, and the number of bend-points occurring on each polyline should be as small as possible. Moreover, the given constraints like minimal node distance, minimal layer distance, etc., must be respected, and besides these constraints, two crucial points have to be fulfilled: First, the results achieved by the previous steps must not be violated. Second, an arc a should be routed such that no line segment of the drawing of a overlaps a node or overlaps another line segment.

After the final coordinates are computed, each arc $a = (u, v)$ is represented by a polyline whose bend-points are the long arc dummies of a.

In the following, we give three methods for coordinate assignment which fulfill all the aforementioned constraints.

BJL: Buchheim, Jünger and Leipert gave a coordinate assignment approach which consists of three steps [BJL99]. First, x-coordinates of long arc dummies are determined such that line segments connecting two dummies are drawn vertically. Then, in the second step, the x-coordinates of the ordinary nodes are computed with respect to the already precomputed x-coordinates and the precomputed node ordering. Based on the observation that a long line segment requires a larger level distance

than short ones in order to obtain a better readability, the y-coordinate of each layer L_i is determined with respect to such long segments which connect the neighbored layer of L_i.

The approach has some nice properties: Due to the runtime $\mathcal{O}((n + m)(\log(n + m))^2)$ where n is the number of regular and dummy nodes and m the number of arcs/subarcs connecting two consecutive layers, it is quite fast. Furthermore, the algorithm computes a layout such that each drawing of an arc has at most two bend-points, and *inner subarcs*—subarcs whose end-points are dummies—are drawn vertically.

We refer to this layout algorithm as `BJL-Layout`.

GKNV: Let δ denote the minimal distance of two nodes, $\mathit{Width}(u)$ the width of the bounding box of node u and u' the immediate left neighbor of node u on the same layer (see Figure 3.2). Let $X(u)$ denote the x-coordinate of node u and let

$$\rho(w', w) = \frac{\mathrm{Width}(w') + \mathrm{Width}(w)}{2} + \delta.$$

In a nice drawing, the arcs should be short and as straight as possible. The latter condition is preferable in order to prevent the "spaghetti effect". To achieve this goal, Gansner, Koutsofios, North and Vo [GKNV93] suggested the following ILP-based solution:

$$\mathrm{X}(w) - \mathrm{X}(w') \geq \rho(w', w) \tag{3.5}$$

$$\min \sum_{a=(u,v)} \Theta(a)|\mathrm{X}(v) - \mathrm{X}(u)| \quad \text{subject to (3.5)} \tag{3.6}$$

where $\Theta(a)$ gives the priority to draw a vertically and $\rho(u', u)$ ensures the minimal distance between the bounding box of node u' and u. Further (3.5) must hold for each pair $\{w', w\}$ on the same layer with w' is the left neighbor of w.

Using standard techniques, this problem can be transformed into a linear program, and then it can be solved optimally using the simplex method. However the transformation increases the overall runtime. Nevertheless, the approach is still fast for practical instances and the computed layouts are very satisfying.

We denote this layout algorithm as `GKNV-Layout`.

BK: Brandes and Köpf presented in [BK02] a very fast layout algorithm with runtime $\mathcal{O}(n + m)$. Their approach minimizes the objective function of (3.6) heuristically and guarantees that each inner subarc is drawn vertically. As reported by the authors, the quality of the drawings is nearly as good as those produced by `GKNV-Layout`.

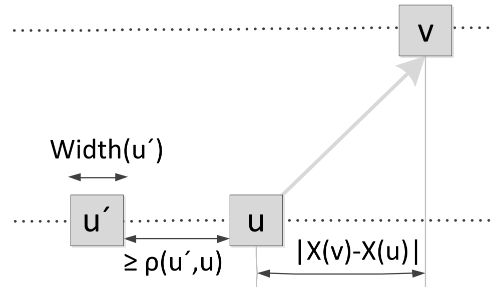


Figure 3.2: Coordinate assignment by Gansner et al.: The algorithm minimizes the distance $|X(v) - X(u)|$ for each arc/subarc (u, v) with respect to some given constraints.

3.1.5 Extensions

Besides the huge amount of suggested solutions for the individual steps, there exists several extensions for certain graph classes or for specific applications. We consider here two important extensions: the extension to directed hypergraphs and the extension for dealing with port constraints.

Hypergraphs

Many important applications such as data flow diagrams or electric schematics require the drawing of directed hypergraphs rather than traditional directed graphs. Furthermore, these application domains prefer drawings in orthogonal style, which has become a quasi drawing standard for visualizing electric schematics. However, only few approaches exist for drawing directed hypergraphs in orthogonal style. Most of them are based on Sugiyama's framework due to the fact that it can be easily modified for this task. Some of the few publications is due to Sander [San96, San99]. He investigated the problem of drawing directed hypergraphs with `Sugiyama`. We describe here some of his ideas:

The hyperarcs can be considered as the main difference between ordinary digraphs and directed hypergraphs. Sander suggested a solution where each hyperarc is replaced by a set of ordinary arcs (see Figure 3.3). A layout for the transformed graph is then computed by using a slightly modified `Sugiyama`. The hyperarcs are "simulated" by overlapping the start line segment of all regular arcs which correspond to the original hyperarc. However, since the simulated hyperarcs branch out very early, the drawings are unnecessarily complex. To overcome this problem, Sander developed a more sophisticated approach also based on `Sugiyama` where the directed hypergraph is first transformed into a suitable underlying digraph [San04].

In contrast to the ordinary digraphs, the crossings of the final layered drawing of a directed hypergraph not only depend on the node order on each

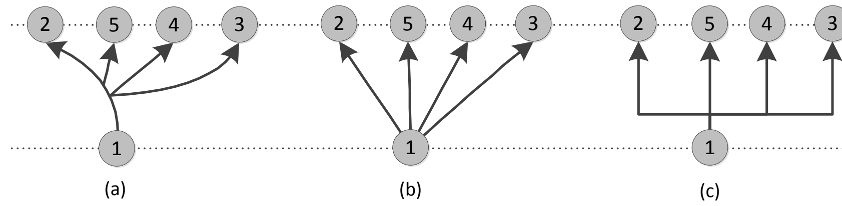


Figure 3.3: Simulating a hyperarc by ordinary arcs: (a) A drawing of a hyperarc. (b) The hyperarc of (a) is replaced by several ordinary arcs. (c) The ordinary arcs are routed such that they overlap each other in the start line segment. Thus, the final orthogonal drawing of the ordinary arcs looks like a drawing of a hyperarc.

layer, but also on the choice of the representation of the hyperarcs, that is, the underlying trees. As shown by Eschbach, Günter, and Becker [EGB06], a good hyperarc routing can also reduce the arc crossings. Furthermore, they proved that orthogonal routing of hyperarcs with the objective to minimized the number of arc crossings is NP-hard even if the node order on each layer is fixed [EGB06]. Thus they suggested a heuristic solution where an orthogonal routing for the arcs is computed with the objective to minimize the crossings. Then, `GreedySwitch` is applied to each layer to improve the results.

Port Constraints

Some applications not only require the drawing of a special graph class like the directed hypergraphs, they also come with special constraints. We consider here the port constraints which is a very common type of constraint in technical drawings.

The *ports* of a node v are prescribed positions that determine where the end-points of the arcs incident to v can connect to the drawing of v. In many applications the ports have a specific semantic interpretation, such as being *input-* or *output-channels* for data tokens in data flow diagrams, or *pins* of electric components in circuit schematics. Thus, in such applications the positioning of ports is not arbitrary, but may be subject to specific constraints (Spönemann, Fuhrmann, von Hanxleden, and Mutzel [SFvHM10]).

Given a set $\mathcal{P}$ of ports for the nodes of a graph G, $\mathcal{P}$ induces a set $\mathcal{C}$ of certain drawing constraints called *port constraints.* The strictest variant of port constraints is the *fixed-port scenario* where the exact position of each port, relative to the respective node, is prescribed.

First approaches to include ports in layer-based drawings were given by Gansner, Koutsofios, North, and Vo [GKNV93] and by Sander [San94]. Spönemann, Fuhrmann, von Hanxleden, and Mutzel suggested a more advanced adaption which considers different types of port constraints and in addition also hyperarcs, as they are required for the layout of data flow diagrams [SFvHM10]. The approach leads to quite acceptable results for different

types of hypergraphs with port constraints, but is still limited by the fact that a bad layering can lead to an unnecessarily high number of arc crossings.

Eiglsperger, Fößmeier, and Kaufmann [EFK00] proposed another approach based on ILP formulations to include constraints in the orthogonalization phase for the *topology-shape-metrics* (Di Battista, Eades, Tamassia, and Tollis [DETT99]) approach. However, due to the runtime, the approach is not practical.

3.2 Alternative Upward Drawing Algorithms

We first describe the mixed upward planarization approach by Eiglsperger, Eppinger, and Kaufmann [EKE03] which can be used to compute upward planar representations of DAGs. An *upward planar representation* (UPR) is an upward planar embedded DAG where crossings are modeled by *crossing dummy nodes* (also called dummies in the following). We call approaches following the idea of planarization but constructing upward representations instead of planar representations *upward planarization* approaches.

The mixed upward planarization is one of the first approaches which successfully adapts the idea of planarization. Before its publication, Di Battista, Pietrosanti, Tamassia and Tollis have already suggested an upward planarization approach based on the planarization of *st*-graphs [BPTT89]. Like the planarization approach, their approach first computes a planar subgraph and then the remaining arcs are reinserted one by one under the premise that an arc is reinserted such that the obtained intermediate graph is a planar *st*-graph. In particular, the intermediate graph must be acyclic. Unfortunately the authors do not clearly explain how this can be achieved.

In [DGL$^+$00], Di Battista et al. experimentally evaluated the upward planarization approach suggested in [BPTT89]. They found out that its performance is very poor, that is, the approach produces drawings with twice the number of arc crossings in comparison to the classical layered upward crossing minimization heuristic where `Barycenter` was used for the second step. Therefore we will only consider the mixed upward approach in this thesis.

Besides the mixed upward planarization approach, we also describe the dominance and the visibility drawing approaches that can construct upward drawings. Originally, the two layout algorithms require as input a planar *st*-graph. Since upward planar DAGs can be augmented to planar *st*-graphs by adding artificial arcs and nodes (Di Battista and Tamassia [BT88]), both approaches can also be used for drawing upward planar DAGs as the artificial auxiliary elements can be omitted in the final drawing. Moreover, they can also be used for drawing UPRs by augmenting them to a planar *st*-graph. In the final drawing the crossing dummies are replaced by crossings.

3.2.1 Mixed Upward Planarization

The mixed upward planarization approach was originally proposed for drawing mixed graphs, that is, graphs which contain both directed and undirected edges. This class of graphs arises for example in software engineering. We depict here the idea of upward planarization of DAGs by Eiglsperger et al. The approach is divided into two main phases:

Subgraph Computation: The idea of this phase follows the idea by Goldschmidt and Takvorian published in [GA94]. First, a planar subgraph is obtained by computing a node order π of the input graph $G = (V, E)$. Then, according to π the nodes are placed on a vertical line. Each edge e can now be added to the "node line" by drawing e left or right of the line when it does not cross an already added edge. Thus, the edges of G are partitioned into three sets: S_l which contains edges drawn left of the line, S_r which contains edges drawn right of the line, and S_b which contains edges causing crossings. The arcs of S_b are not added to the node line. Then a *conflict graph* is constructed for each crossing pair of edges. Since an induced bipartite subgraph of the conflict graph corresponds to a valid partition of E into the set S_l, S_r and S_b, a *maximum bipartite subgraph* corresponds to a maximum set of $S_l \cup S_r$. Due to the NP-hardness of the maximum bipartite subgraph problem (Garey and Johnson [GJ83]), Goldschmidt and Takvorian use heuristics to tackle this problem.

Eiglsperger et al. adapted the approach for DAGs by replacing the part of the algorithm responsible for computing the node order π with a variant of a standard topological sorting algorithm. This guarantees that the final subgraph U is upward planar.

Arc Reinsertion: The main problem arising here is to compute an insertion path for an arc $a \in S_b$ such that after its insertion, the obtained intermediate graph is upward planar, in particular acyclic. To achieve this goal, the upward embedded subgraph U is augmented to an *st*-graph U'. Then a layering $\mathcal{L}$ of U' is computed, and a routing network R with respect to a and U' is constructed. By the combination of $\mathcal{L}$ and R, a computed insertion path for a can be realized such that the obtained intermediate graph is upward planar. Particularly it can be used as starting point for the reinsertion of the next arc.

The approach can be extended to mixed graphs and by using layout algorithms described in the next subsection, a final upward drawing can be constructed. Regarding the quality (the number of crossings), the authors reported good results in comparison to multi-run versions of the classical layered approaches based on `Barycenter`. We refer to the mixed upward planarization approach as `MUP`.

3.2.2 Dominance Drawing

The dominance drawing method was suggested by Di Battista, Tamassia, and Tollis for upward drawing of planar *st*-graphs [DTT92]. Notice that planar *st*-graphs are upward planar (Garg and Tamassia [GT95]). A *dominance drawing* of G is a drawing $\mathcal{D}$ such that for any pair of nodes u and v the following holds: u dominates v in G if and only if $X(u) < X(v)$ and $Y(u) < Y(v)$.

There are two variants of the dominance drawing algorithm: the straight-line and the polyline variant. For both variants, the dominance drawing algorithm constructs in linear time a *dominance drawing* of the input *st*-graph. Besides the fast runtime, it constructs an upward drawing with few bend-points. Moreover, the straight-line variant always produces drawings with *minimum* area requirement regardless of the given resolution rules, and it is notable appropriate for displaying symmetries of G.

The straight-line version of dominance drawing can be depicted as follows: Let G be a planar embedded *reduced st*-graph, that is, G contains no transitive arcs. In a pre-processing step, a data structure is constructed in order to provide a quick access to the arcs; in particular, the left and the right in- and outgoing arc of each node. Then, in a preliminary layout phase, the coordinates are assigned to the nodes by traversing G starting from the source. In the first traversal, the order of the visited nodes π_1 are determined by the clockwise arc order. The x-coordinates are then assigned to the nodes with respect to π_1. The visited node order π_2 in the second traversal is determined by the counterclockwise arc order. Similar to the assignment of the x-coordinate, the y-coordinates are assigned to the nodes according to π_2. After obtaining a preliminary layout, a compaction algorithm is applied and the final coordinates of the nodes are determined. Finally, the arcs are drawn as straight-lines.

The straight-line version is restricted to *st*-graphs without transitive arcs, but it can be easily extended to general *st*-graphs by splitting each transitive arc by introducing a dummy node. A polyline drawing can be obtained by replacing the introduced dummy nodes with bend-points.

We refer to the polyline variant of the domination drawing approach as `Dominance`.

3.2.3 Visibility Representation

In a visibility representation of a planar embedded *st*-graph $G = (V, A)$, a node u is drawn as a horizontal node-segment μ and an arc $a = (u, v)$ is drawn as a vertical arc-segment ϵ such that no node-segment and no arc-segment overlaps each another. In addition, the end-points of the arc segment ϵ corresponding with a is located at $Y(u)$ and $Y(v)$ and ϵ does not intersect any other node-segment (see Figure 3.4).

The visibility drawing method has been introduced by Rosenstiehl and Tarjan [RT86] and independently by Tamassia and Tollis [TT86]. Similar to

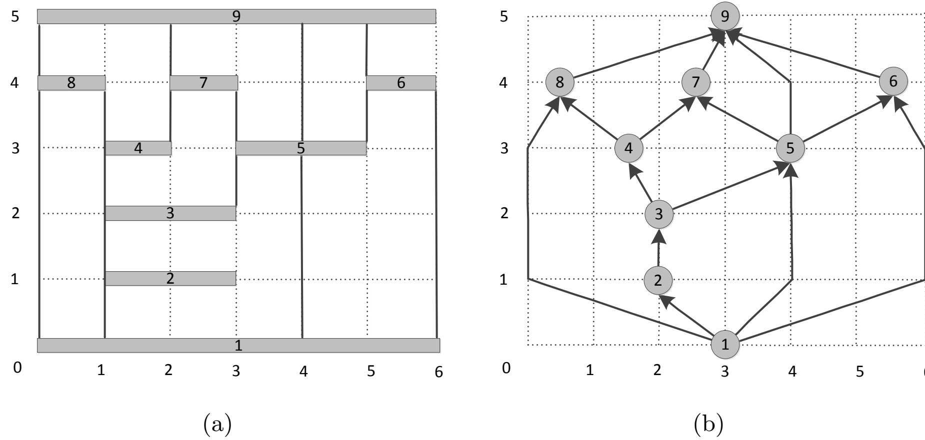


Figure 3.4: Construction of upward drawings based on visibility representations (Redrawn figures of a example in [DETT99]): (a) a visibility representation of digraph (b); (b) A drawing obtained from (a) by using arithmetic mean node positioning.

`Dominance`, the visibility approach requires a planar *st*-graph as input. The algorithm requires further the construction of a directed dual graph G^D of G which can be constructed similar to an undirected dual graph as defined in Definition 2. After the construction of G^D, two topological numberings of the nodes are computed. A *topological numbering* of G is a numbering $\#(\cdot)$ of the nodes of G such that for each arc $a = (u, v)$, $\#(v) \geq \#(u)$ holds. The first topological numbering of the nodes is considered as the x-coordinate. It is obtained by applying a standard topological sorting algorithm to G^D. The second topological numbering is considered as the y-coordinates of the nodes and is obtained by applying a topological sorting algorithm to G.

A visibility representation is then constructed by using the precomputed coordinates of the node-segments. As we are interested in a drawing but not in a visibility representation of G, the final task is to transform the visibility representation into a drawing of G. This can be done easily by choosing suitable coordinates for the nodes derived from the corresponding node-segments. For example: Suppose the x-coordinate of the node-segment μ corresponding to node u spans from x_a to x_b, then we can assign u to the coordinate x_a, x_b, or the arithmetic mean $\lfloor \frac{x_a+x_b}{2} \rfloor$. In a similar way, a suitable routing for an arc a can be derived from the coordinates of the corresponding arc-segment ϵ.

The visibility algorithm has some nice properties: It constructs a polyline upward drawing in linear time. For any given resolution rules, the area of the drawing is bounded by $\mathcal{O}(|V|^2)$ and every drawing of an arc has at most two bend-points. We denote the visibility drawing approach as `Visibility`.

Chapter 4

Upward Planarization

4.1 Introduction

Since for many applications the number of arc crossings arising in a drawing is considered as an important aesthetic criterion, finding crossing minimization algorithms is one of the main tasks in graph drawing. In the context of drawing DAGs, this task can be stated as follows:

Definition 3 (Upward Crossing Minimization). *Given a DAG G, we are interested in an upward drawing of G with the minimum number of arc crossings.*

This problem is closely related to the problem of computing the upward crossing number of G, where we are only interested in the minimum number of arc crossings:

Definition 4 (Upward Crossing Number $ucr(G)$). The *upward crossing number $ucr(G)$* of a DAG G is the minimum possible number of arc crossings with which G can be drawn in an upward fashion.

However, upward crossing minimization and hence also computing $ucr(G)$ is NP-hard since upward planarity testing is already NP-hard (Garg and Tamassia [GT01]). To the best of our knowledge, only trivial exponential algorithms exist for *exact* upward crossing minimization. Yet, there exist two main approaches which can be used for computing heuristic solutions:

The first main approach for upward crossing minimization does not directly tackle the upward crossing minimization problem, instead, the related k-LCM problem is considered. It consists of two steps: In the first step, a k-LCM instance—a layering $\mathcal{L}$ of the input DAG G—is computed, then in the second step, the obtained instance is solved by heuristics or solved exactly by an ILP or an SDP formulation. This approach is known as the first and the second step of the Sugiyama framework. We refer to it as *layered upward crossing minimization* approach.

The second main approach is upward planarization. Upward planarization is based on the idea of planarization which is recognized as a very successful heuristic for minimizing edge crossings of undirected graphs [GM04]. However there exist only few approaches, for example, Eiglsperger et al. [EKE03] or Di Battista et al. [BPTT89].

In this chapter, we propose a new approach referred to as *layer-free upward planarization (LFUP)* for upward planarization of DAGs.

Organization of this chapter. In the first section of this chapter, the motivation for a new upward planarization approach is given. Since the adaption of the planarization approach for DAGs is by far not trivial, Section 4.1.2 is dedicated to the problems that arise on the way toward to a new upward planarization algorithm. We then give an overview of LFUP (Section 4.2.1) and thereafter depict its theoretical backgrounds. After analyzing the theoretical runtime of LFUP in Section 4.2.4, we give some extensive experimental evaluations in Section 4.3. In Sections 4.4–4.5, we explain how to extend LFUP such that it can handle port constraints and compute UPRs for directed hypergraphs, and in Section 4.6, we give an illustrated example of LFUP.

4.1.1 Motivation

Since the publication of the Sugiyama drawing framework, the layered upward crossing minimization approach has received a lot of attention. Though various refinements and improvements were developed, one inherent drawback is not overcome by any of these modifications even when using optimal solutions for the k-LCM problem: assigning nodes to fixed layers in the first step can severely affect the subsequent crossing minimization step, requiring arc crossings that would be unnecessary if a "better" layer assignment has been chosen. Figure 4.1 gives an example: Obviously, when using the illustrated layering (a) for the crossing minimization step, the crossings can never be totally eliminated. However, the example DAG can be drawing without any arc crossings (b), since it is upward planar.

As mentioned before, planarization is considered as a very successful approach for crossing minimization of undirected graphs, but it cannot be used for drawing digraphs in an upward fashion, since the approach does not take the arc direction into account. But if we can develop an upward planarization approach based on the idea of planarization, then we may achieve similar success for DAGs.

The results published by Eiglperger et al. [EKE03] strengthen our assumption that upward planarization could be as successful as planarization for undirected graphs. However the technique used by Eiglsperger et al. for computing the upward planar subgraph is based on a simple idea. Moreover, their approach uses a layer-based algorithm for arc reinsertion which can reduce the number of possible insertion paths, in particular "good" inser-

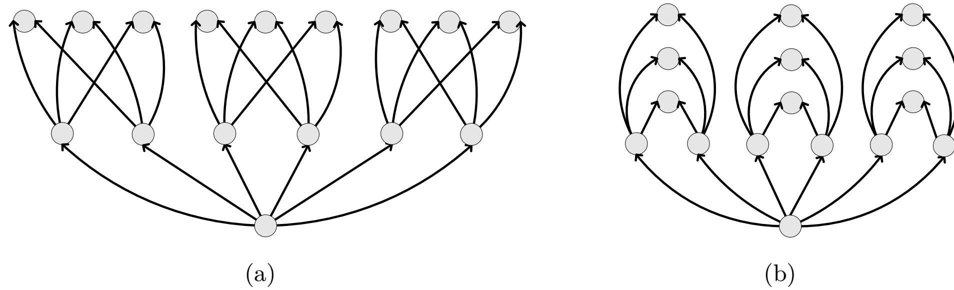


Figure 4.1: A bad layering (a) can force unnecessary crossings. In this example the graph is in fact upward planar (b).

tion paths may not be considered. Sophisticated techniques for the subgraph computation and the arc reinsertion step may lead to better results.

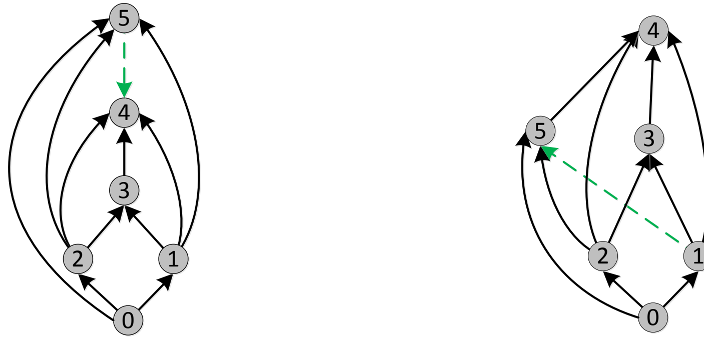
4.1.2 Challenges

Adapting the planarization approach for upward drawings is by no means straight-forward. Firstly, while planarity can be tested efficiently, testing upward planarity is NP-complete (Garey and Johnson [GT01]); on the other hand, upward planarity can efficiently be tested for digraphs with a single source (Hutton and Lubiw [HL96]; Bertolazzi, Di Battista, Mannino, and Tamassia [BDMT98]).

Secondly, while any planar subgraph is suitable as starting point for the edge reinsertion phase in the undirected case, further constraints are necessary for upward drawings. For example, assume we construct the upward planar subgraph straight-forwardly by adding arcs to an initially empty subgraph; after each arc we test for upward planarity. The process stops when no more arcs can be inserted without losing upward planarity.

As shown in Figure 4.2(a), the subgraph obtained by deleting the arc $(5, 4)$ has exactly two upward planar embeddings (Due to the fact that it is a 3-connected planar graph.). One is as illustrated and the second can be obtained by reversing the arc order on each node of (a). Obviously, in each upward planar drawing of the subgraph the node 5 has to be drawn higher than the node 4. Hence, we cannot insert the remaining arc $(5, 4)$ *at all*, no matter how many crossings we may use and which upward embedding we choose for the subgraph. So the first challenge is to find an approach to identify a *feasible upward planar subgraph (FUPS)*.

And finally, even if we have a FUPS, we cannot easily insert arcs iteratively into it in a crossing minimal fashion, without taking the not-yet inserted arcs into account. Figure 4.3(a) shows that, even though it is possible to insert both arcs into U, inserting one arc straight-forward may make inserting the other one impossible. Therefore, the second challenge is to find an approach to solve this problem without reducing the number of possible insertion paths.



(a) An infeasible upward planar subgraph: the arc $(5, 4)$ cannot be inserted.

(b) A feasible upward planar subgraph (FUPS).

Figure 4.2: Problems with simple arc insertion approaches: Not every upward planar subgraph (bold lines) is feasible with respect to the temporarily removed arcs (dashed lines) of the original graph that have to be inserted in the subsequent steps.

4.2 Upward Planarization Algorithm

We first give an overview of the layer-free upward planarization algorithm, then explain how to face the two main challenges: the *feasible subgraph computation* and the *arc reinsertion problem*.

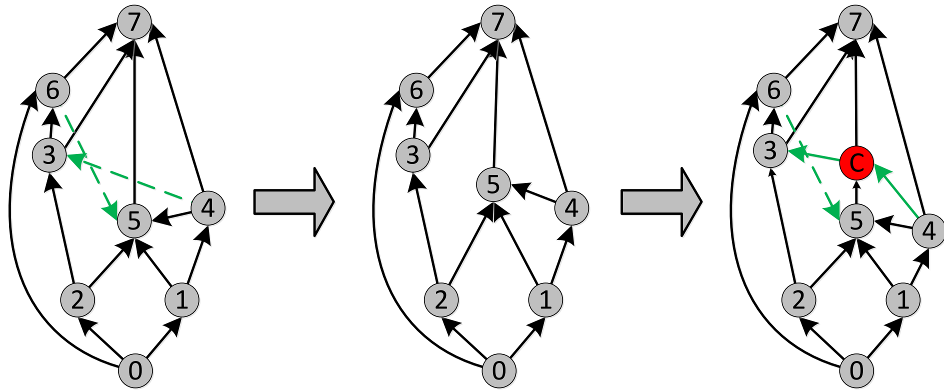
4.2.1 Algorithm Overview

Due to the NP-hardness of upward planarity testing of general DAGs, we focus on the class of sT-graphs. If the input graph has multiple sources, we can add a super source node $\hat{s}$, connect it to all original sources, and later set the costs for crossing these additional arcs to zero. The additional arcs and the new super source will be omitted in the final drawing.

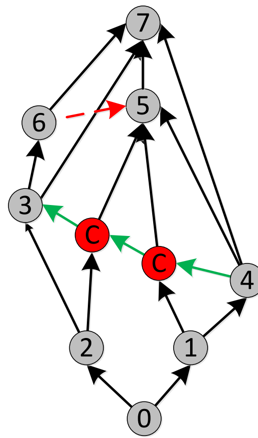
Like the traditional planarization approach for undirected graphs, our algorithm consists of two phases:

LFUP	
Phase 1	Feasible Upward Planar Subgraph Computation
Phase 2	(Feasible) Arc Reinsertion

Algorithm 1 offers more details regarding the layer-free upward planarization approach: In the first phase we construct a *feasible* upward planar subgraph (line 2–12). This can be done by first computing a spanning tree U (line 2) and add the arcs not in U one by one (line 4–12). Each time an arc $\tilde{a}$



(a) The given graph (left) and a FUPS (middle) obtained by removing the arcs $(4, 3)$ and $(6, 5)$. A crossing minimal insertion path for $(4, 3)$ is infeasible as $(6, 5)$ can no longer be inserted (right) due to the arising cycle $\langle 6, 5, C, 3, 6\rangle$.



(b) The arc $(4, 3)$ has to be inserted differently (and requiring one more crossing), such that $(6, 5)$ can be inserted.

Figure 4.3: Problems with simple arc insertion approaches: Not every upward insertion path is feasible with respect to the temporarily removed arcs of the original graph that have to be inserted in the subsequent steps.

is added, an upward planarity test (which also upward embeds $U' := U \cup \{\tilde{a}\}$ if possible) is performed (line 6). If U' is upward planar, then we obtain an upward planar embedding Γ' of U'. An auxiliary graph $\mathcal{M}(\Gamma')$—called merge graph—is then constructed with respect to Γ'. The special properties of $\mathcal{M}(\Gamma')$ allow us to use an acyclicity test to decide whether U' is suitable as starting point for the second phase or not (line 6). If U' is not upward planar or $\mathcal{M}(\Gamma')$ is cyclic, then U' is not suitable and we delete $\tilde{a}$ from U' and add it to the set B (line 10). The arcs of B will be reinserted in the second phase. We refer to the upward planarity testing of U' together with the acyclicity testing of $\mathcal{M}(\Gamma')$ (line 6) as *fixed-embedding FUPS testing.*

In the second phase (line 15–19), we iteratively insert the arcs not yet in the FUPS U so that few crossings arise; these crossings are replaced by dummy nodes so that the graph in which arcs are inserted can always be considered as upward planar (line 17). Inserting an arc $\tilde{a}$ into a planar graph thereby means that all arising crossings lie on $\tilde{a}$; we do not introduce additional crossings purely on the planar graph itself. The final result, the upward planar representation, can be turned into a drawing of the original graph by using appropriate layout algorithms like `Dominance`, `Visibility` or the new layout algorithms `UPL` introduced in Chapter 5. In the following, let G be an sT-graph.

4.2.2 Feasible Subgraph

Feasible subgraph testing, that is, testing an subgraph U whether it is suitable as starting point for the arc reinserting phase or not, is a fundamental step in upward planarization. We consider here a more restricted variant of feasible subgraph testing: the fixed-embedding FUPS testing.

Now, let $U = (V, A')$ be an upward planar subgraph of an sT-graph $G = (V, A)$ and $B = A \setminus A'$. We naturally extend Definition 1 (Insertion Path):

Definition 5 (Upward Insertion Path (UIP))**.** An *upward insertion path* p with respect to some arc $\tilde{a} = (x, y) \in B$ is an insertion path for $\tilde{a}$ such that the graph U' obtained from realizing p is upward planar.

Let Γ' be an upward planar embedding of U'. We say Γ' *induces* an upward planar embedding Γ of U that is obtained by reversing the realization procedure while maintaining the embedding. That is, we fix the embedding Γ' of U', replace the dummy nodes in U' by arc crossings and then delete the arc $\tilde{a}$. The result is the embedding Γ of U.

For example, the right drawing of Figure 4.3(a) shows the graph U' with embedding Γ' after arc $(4, 3)$ has been inserted. Notice, arc $(6, 5)$ is not an arc of U'. By reversing the realization of the insertion path for $(4, 3)$, we obtain U (middle) and its embedding Γ.

We will see that inserting the arcs of B one by one cannot be done without taking the remaining arcs into account. Thus we define:

Algorithm 1 Upward planarization algorithm `LFUP`

Require: sT-Graph $G = (V, A)$
Ensure: Upward planar representation of G

1: ▷ Phase 1: FUPS Computation
2: Identify spanning tree $U = (V, A_T)$ of G, routed at source s.
3: $B := \emptyset$
4: **for** each $\tilde{a} \in A \setminus A_T$ **do**
5: $U' := U + \tilde{a}$
6: **if** $\exists$ upward planar embedding Γ' of U' **and** $\mathcal{M}(\Gamma')$ is acyclic **then**
7: $U := U'$, $\Gamma := \Gamma'$
8: **continue**
9: **else**
10: $B := B \cup \{\tilde{a}\}$
11: **end if**
12: **end for**
13: ▷ We have a FUPS $U = (V, A')$, and arcs $B = A \setminus A'$ to reinsert into U.
14: ▷ Phase 2: Arc Reinsertion
15: **for** each $\tilde{a} \in B$ **do**
16: Compute insertion path p for $\tilde{a}$ into Γ that will ensure property **(M)**
17: Insert $\tilde{a}$ along p, replacing crossings by dummy nodes $\rightarrow$ new U, Γ.
18: **Property (M):** $\mathcal{M}(\Gamma)$ is acyclic
19: **end for**

Definition 6 (Upward Insertion Sequence)**.** An *upward insertion sequence* is a sequence of k UIPs for all arcs of B. The first arc in the sequence is inserted into U—introducing dummy nodes—which results in an upward planar embedded graph $U' = U_1$ called *intermediate UPR*. The second arc is then inserted into U_1 which results in an intermediate UPR U_2, etc. After realizing all insertion paths, we hence obtain a final upward planar graph U_k, which is a UPR of G.

Our approach is restricted to the *fixed-embedding scenario*, that is, we start with a subgraph U with upward planar embedding Γ in the arc reinsertion phase and insert the arcs of B one by one. The upward planar embedding Γ_i of the intermediate UPR U_i obtained after realizing the i-th arc $\tilde{a}_i \in B$ induces the embedding Γ_{i-1} of the previous intermediate UPR U_{i-1} for $0 < i \leq k$. Thus, the final embedding Γ_i induces Γ.

Definition 7 ((Constraint) Feasible Upward Insertion Path)**.** The UIP p is *feasible* with respect to Γ, if there exists an upward planar embedding Γ' of the graph U', obtained after realizing p, which induces Γ. It is *constraint feasible* with respect to Γ, if it furthermore allows an upward insertion sequence for the arcs $B \setminus \{\tilde{a}\}$ into U' such that there exists an upward planar embedding

Γ_k of the final graph U_k that induces Γ. We say the path p is *(constraint) minimal* if it requires the fewest crossings over all (constraint) feasible UIPs.

Obviously, a constraint feasible UIP is a feasible UIP and a feasible UIP is a UIP. Moreover, in the fixed-embedding scenario, the insertion paths of an upward planar insertion sequence are all constraint feasible.

Regarding the definition of constraint feasible UIP, Figure 4.3(a) gives an example of its necessity: The right-most graph in Figure 4.3(a) illustrates a minimal feasible UIP p_1 for $\tilde{a}_1 = (4, 3)$. We realize p_1 by splitting arc $(5, 7)$ (introducing the crossing dummy C) and adding the arcs $(4, C)$ and $(C, 3)$ to represent $\tilde{a_1}$ in the intermediate UPR U_1. Since U_1 is upward planar, p_1 is a feasible UIP. However, inserting $\tilde{a}_2 = (6, 5)$ in U_1 would inevitably cause a cycle; hence p_1 is not a constraint feasible UIP. An alternative, constraint feasible UIP for $\tilde{a}_1$ is illustrated in Figure 4.3(b).

Definition 8 (Feasible Upward Planar Subgraph (FUPS) and Embedding)**.** An upward planar subgraph U of G is *feasible* if there exists an upward insertion sequence. An upward planar embedding Γ of a FUPS U is *feasible* if there exists an upward insertion sequence such that the final embedding Γ_k induces Γ.

Obtaining the formal definition of a FUPS, the goal now is to find an approach for its construction; in particular finding a subgraph feasibility test. For this purpose we introduce the central idea based on an auxiliary graph called merge graph:

Definition 9 (Merge Graph)**.** The *merge graph* $\mathcal{M}(\Gamma)$ of U with respect to an upward planar embedding Γ of U is constructed as follows:

1. We start with $\mathcal{M}(\Gamma)$ being a copy of G.
2. For each internal face f of Γ, we add an arc from each non-top sink-switch of f to the top sink-switch of f. We call these arcs *sink-arcs*.

A merge graph $\mathcal{M}(\Gamma)$ models the hierarchical information of the nodes with respect to an embedded subgraph U and the set B. The arcs of $\mathcal{M}(\Gamma)$ can be partitioned into the sink-arcs, the arcs of U, and the arcs of B. The sink-arcs model the hierarchies between the top-sink-switches and non-top-sink-switches on each internal face of Γ, therefore mapping the hierarchical information of Γ to $\mathcal{M}(\Gamma)$. The arcs of U and the arcs of B reflect the node hierarchy induced by G.

Let Y be the nodes of U dominated by y including y and let X be the nodes of U dominating x including x. The *dominated subgraph* in G of y is the subgraph induced by the nodes of Y and the *dominating subgraph* in U of x is the subgraph induced by the nodes of X.

Let v_1 and v_2 be two nodes of G. Considering some specific upward drawing of G, we denote by $v_1 \prec v_2$ that v_1 is drawn lower than v_2. An upward planar drawing clearly requires $v_1 \prec v_2$ if $v_1 \rightsquigarrow v_2$.

Due to the fact that the hierarchy of the nodes is mapped to merge graph, we observe:

Observation 1. *For any fixed upward planar embedding* Γ*, there always exists an upward planar drawing with respect to* Γ *with* $v_1 \prec v_2$ *if* $v_2 \not\rightsquigarrow v_1$ *in* $\mathcal{M}(\Gamma)$*.*

The merge graph with respect to an embedded upward planar subgraph has one crucial property:

Lemma 1 (Feasibility Lemma)**.** *The merge graph* $\mathcal{M}(\Gamma)$ *is acyclic if and only if there exists an upward insertion sequence* S *such that the final graph* U_k *obtained after inserting all arcs of* B *according to* S *is upward planar (with embedding* Γ_k*).*

Proof. $\exists$ Sequence $\Longrightarrow$ $\mathcal{M}(\Gamma)$ is acyclic: Let U_k be the UPR of a DAG G obtained after inserting all arcs according to the upward insertion sequence S. Let Γ_k be an upward planar embedding of U_k. Considering an upward planar drawing $\mathcal{D}_k$ of U_k inducing Γ_k, we replace all dummy nodes with crossings and delete the arcs of B in $\mathcal{D}_k$. The graph associated with the new drawing $\mathcal{D}$ is the upward planar subgraph U and the embedding induced by $\mathcal{D}$ is Γ. For each inner face $f \in \Gamma$, we draw a sink-arc from each non-top sink-switch to its corresponding top sink-switch. As Γ is upward planar, these arcs are clearly oriented upwards in the drawing. Finally, we reintroduce the arcs of B into $\mathcal{D}$, drawing them exactly as in $\mathcal{D}_k$. By these operations we obtain a drawing of the merge graph $\mathcal{M}(\Gamma)$ and as all arcs in the drawing are oriented upwards, $\mathcal{M}(\Gamma)$ is acyclic.

$\mathcal{M}(\Gamma)$ is acyclic $\Longrightarrow$ $\exists$ Sequence: Let U be a FUPS with an upward embedding Γ and $\mathcal{M}(\Gamma)$ its acyclic merge graph. Let $\mathcal{M}(\Gamma)^-$ be the graph $\mathcal{M}(\Gamma)$ without the arcs of B. Ignoring the sink-arcs of $\mathcal{M}(\Gamma)^-$ first, we embed $\mathcal{M}(\Gamma)^-$ like Γ, thereafter we embed the sink-arcs without crossings within their corresponding faces. Let $\Gamma_{\mathcal{M}(\Gamma)^-}$ be the resulting embedding. Notice that $\Gamma_{\mathcal{M}(\Gamma)^-}$ is an upward planar embedding. Let $\tilde{a}_i = (x_i, y_i)$, $1 \leq i \leq k$, be the arcs not in U. Since $\mathcal{M}(\Gamma)$ contains these arcs and is acyclic, we know that $y_i \not\rightsquigarrow x_i$ in $\mathcal{M}(\Gamma)^-$ and $\mathcal{M}(\Gamma)$, for all $1 \leq i \leq k$. Hence, considering any arc $\tilde{a}_i$ individually, we can find a drawing of the graph $\mathcal{M}(\Gamma)^-$ with respect to $\Gamma_{\mathcal{M}(\Gamma)^-}$ where $x_i \prec y_i$ (Observation 1).

We show that this holds for all arcs together by induction over the number of arcs to be inserted:

Induction basic: We start with an arbitrary upward drawing $\mathcal{D}_1$ of $\mathcal{M}(\Gamma)^-$ inducing $\Gamma_{\mathcal{M}(\Gamma)^-}$ which is stepwise modified. If $y_1 \prec x_1$, then we modify $\mathcal{D}_1$ without violating the embedding $\Gamma_{\mathcal{M}(\Gamma)^-}$ such that $x_1 \prec y_1$ holds. This is always possible since $y_1 \not\rightsquigarrow x_1$ in $\mathcal{M}(\Gamma)$ (Observation 1).

Induction conclusion: Now consider the drawing $\mathcal{D}_k$ inducing $\Gamma_{\mathcal{M}(\Gamma)^-}$ in the k-th step where by induction hypothesis, we have $x_i \prec y_i$ for all $1 \leq i < k$.

If $x_k \prec y_k$, then there is nothing to do in this step, so assume $y_k \prec x_k$. Since $\mathcal{M}(\Gamma)$ is acyclic, we know that $y_k \not\rightsquigarrow x_k$ in $\mathcal{M}(\Gamma)$ and hence there exists a drawing with $x_k \prec y_k$ (Observation 1). We show that we can fulfill the latter condition without violating the respective embedding $\Gamma_{\mathcal{M}(\Gamma)^-}$ and the order $x_i \prec y_i$ in $\mathcal{D}_k$ for each $i < k$.

Considering $\mathcal{M}(\Gamma)$, let Y_k be the dominated subgraph in $\mathcal{M}(\Gamma)$ of y_k without any arcs of B. So Y_k contains the subgraph of $\mathcal{M}(\Gamma)^-$ which is dominated by y_k in $\mathcal{M}(\Gamma)$. Let $\overline{Y_k}$ be the graph obtained from $\mathcal{M}(\Gamma)^-$ after deleting the nodes and arcs of Y_k. An arc that connects a node of $\overline{Y_k}$ and a node of Y_k has its source node in $\overline{Y_k}$ and its target node in Y_k, since Y_k is a dominated graph of y_k (see Figure 4.4). We know that $x_k \notin Y_k$ as $\mathcal{M}(\Gamma)$ is acyclic. In $\mathcal{D}_k$, we perform an *upward shift* of the subgraph Y_k: we move y_k and all nodes of Y_k such that y_k is above x_k. Due to the sink-arcs, $\mathcal{M}(\Gamma)^-$ does not have any maximal bi-connected component C that is within another such component. Hence the upward shift will not result in any crossings. Moreover, it preserves the induced embedding $\Gamma_{\mathcal{M}(\Gamma)^-}$.

Assume that the shift would invalidate $x_i \prec y_i$ for some i. This would mean that x_i would be in Y_k but y_k would not. But since (x_i, y_i) is an arc in the merge graph $\mathcal{M}(\Gamma)$ this cannot be the case. Therefore the upward shift in the k-th step ensures an upward planar drawing of $\mathcal{M}(\Gamma)^-$ where we have $x_i \prec y_i$ for $1 \leq i \leq k$. We now can draw the arcs $\tilde{a}_1, \ldots, \tilde{a}_k$ into $\mathcal{D}_k$ in an upward fashion. Considering $\tilde{a}_1, \ldots, \tilde{a}_k$ as a sequence of arcs to be inserted, we can generate a constraint feasible UIP p_i for each $\tilde{a}_i$ just by tracking a_i from x_i to y_i in the final drawing. p_i can be derived by the order of the crossed arcs along $\tilde{a}_i$. □

The next corollary follows immediately from Lemma 1:

Corollary 1. *An upward planar embedding* Γ *of a FUPS* U *is feasible if and only if the corresponding merge graph* $\mathcal{M}(\Gamma)$ *is acyclic.*

Hence line 6 of Algorithm 1 realizes a fixed-embedding FUPS test.
In Figure 4.5 the merge graph of the upward planar subgraph of Figure 4.2(a) is illustrated. As it contains a cycle, the subgraph is not feasible.

Algorithm 1 computes a FUPS $U = (V, A')$ with embedding Γ which is not necessarily *maximal*, that is, there may be a FUPS $U' = (V, A'')$ with embedding Γ' such that $|A'| < |A''|$ and Γ' induces Γ. So after obtaining U, we can improve the results by fixing the embedding Γ, and try to reinsert the arcs in B. The result is a maximal embedded FUPS.

4.2.3 Arc Reinsertion

Having solved the feasible subgraph problem, we now consider the problem of inserting arcs into a FUPS with few arc crossings by iteratively adding the arcs not in the FUPS. In the following we are again given an sT-graph

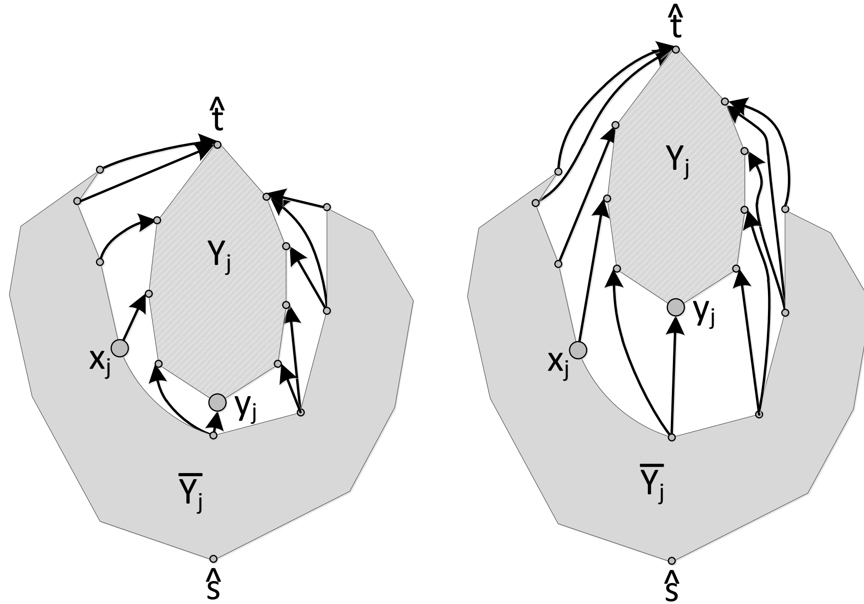


Figure 4.4: Illustration for proof of Lemma 1: (left) An upward planar drawing of $\mathcal{M}(\Gamma)^-$. Before the upward shift it is $Y(y_j) < Y(x_j)$. (right) The drawing after the upward shift of the subgraph Y_j. Observe that arcs which connect the nodes of Y_i with the nodes of $\overline{Y_i}$ are all pointing to Y_i, that is, the target nodes of the arcs are nodes of Y_i, hence we can shift Y_i without violating the hierarchies of the nodes in Y_i and in $\overline{Y_i}$.

$G = (V, A)$, a FUPS $U = (V, A')$, and an upward planar embedding Γ of U. Let $B = A \setminus A'$ and let $\tilde{a} = (x, y) \in B$. Formally, we can define the arising problem per arc as follows:

Definition 10 (Constraint Upward Arc Insertion with Fixed-embedding)**.** The *constraint upward arc insertion problem with fixed-embedding* is to find a constraint feasible minimal UIP for $\tilde{a}$ into U with respect to Γ and the arcs $B \setminus \{\tilde{a}\}$.

Let $p = \langle a_1, \ldots, a_n \rangle$ be an insertion path for (x, y). Associated with p is a sequence of faces $F = \langle f_1, \ldots, f_k \rangle$ such that when p is realized, each face f_i of F is split into two or more faces (see Figure 4.6). The latter case occurs if p contains more then two arcs of f_i.

Let $a_j, a_{j+1} \in f_i$. Assume p is realized by splitting a_j and a_{j+1} by introducing the dummy nodes d_j and d_{j+1}, respectively, and f_i is split by the arc (d_j, d_{j+1}) into two faces. We say p *enters* f_i through a_j and *leaves* f_i through a_{j+1} and f_i is traversed by the insertion p.

Let now $q = \langle a_1, \ldots, a_m \rangle$ and $r = \langle a_{m+1}, \ldots, a_n \rangle$. We denote a concatenation of two (insertion) paths q and r with $q + r$. Thus $p = q + r$. A realization

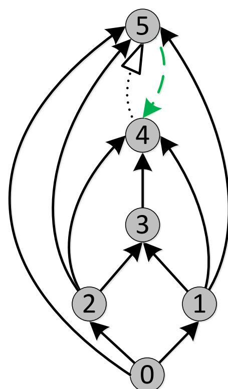


Figure 4.5: The merge graph of the upward planar subgraph of Figure 4.2(a). The deleted arc $(5, 4)$ is drawn as green dashed arc and the only sink-arc $(4, 5)$ is drawn as dotted arc with hollow arrow head. Since the merge graph is not acyclic, the upward planar subgraph is not feasible.

of q is a *partial realization* of p. The path q ends at the arc a_m and we say q is an insertion path from x to the arc a_m.

Since we will augment the subgraph U into a bi-connected graph, we can assume that a_m is embedded on the boundary of two faces f_l and f_r with $f_l \neq f_r$. Without loss of generality, let f_l be the last face in the sequence of traversed faces of q. Then, a_m is a *dynamic entrance* to the face f_r of the insertion path q. Referring to the example of Figure 4.6, the subpath $q = \langle a_1 \rangle$ of p has traversed the face f_1 leaving f_1 through arc a_1 and enters the face f_2 through arc a_1, so a_1 is the dynamic entrance of q to face f_2.

Routing Network

In order to compute a UIP for $\tilde{a} = (x, y)$ we use a *routing network* R. For its construction, we first augment U and Γ with the sink-arcs as we did for the merge graph. Furthermore, we add a super sink node $\hat{t}$ and sink-arcs $(t, \hat{t})$, for each sink t on the external face of Γ. We denote the augmented graph with $\hat{U}$ and the upward planar embedding of $\hat{U}$ with $\hat{\Gamma}$. The augmentation guarantees that all faces in $\hat{\Gamma}$ are *simple*, that is, they have exactly one source- and one sink-switch, hence alleviates the construction of the routing network.

Each simple face $f \in \hat{\Gamma}$ consists of two directed paths—one on its left and one on its right hand side—from the source-switch to the sink-switch of f. We call the left/right directed path p_ℓ/p_r just the *left/right side of f*. The face f is called the *right face* of the arcs of p_ℓ, and the *left face* of the arcs of p_r.

For the traditional edge insertion problem, that is, without considering

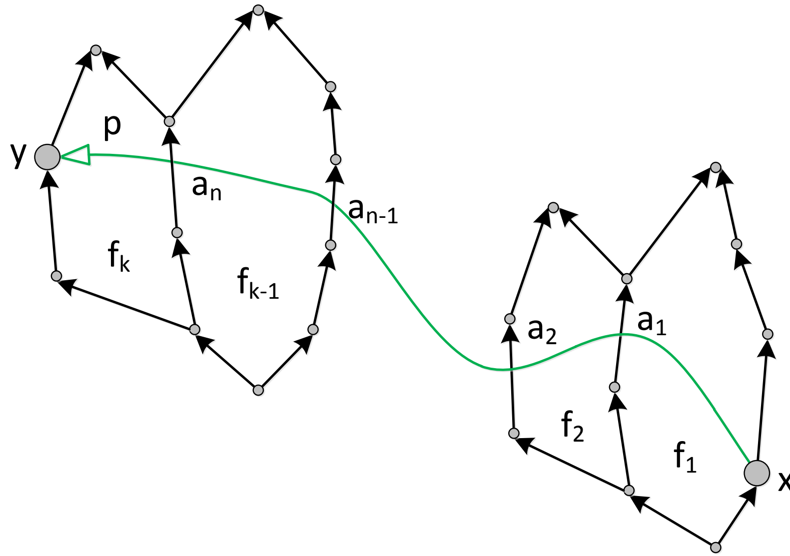


Figure 4.6: The insertion path $p = \langle a_1, \ldots, a_n \rangle$ traversing a sequence of faces $\langle f_1 \ldots, f_k \rangle$. After realizing p, each face of sequence is split by subarcs which represent the arc (x, y).

upward planarity, we simply use the bi-directed dual graph of U with respect to Γ, augmented with the start and end nodes of $\tilde{a}$. Due to the required upward planarity, we have to use a more heavily augmented routing network. We do not represent single faces as single nodes in R but as well-structured sub-networks, as shown in Figure 4.7(a) schematically for an internal face and (b) for the external face. Such a sub-network guarantees that when the currently considered insertion path enters a face f through some arc, it can only leave f either above that arc or on the other side of f. We call the arcs that are the dual of some arc of $\hat{U}$ the *crossing-arc*, as a path over these arcs crosses an arc of $\hat{U}$. Arcs of R which are not crossing-arcs are called *auxiliary routing arcs*.

Finally, we add the nodes x^* and y^* corresponding to x and y to R which will be the start and end node of the insertion path, respectively.

Let A_x and A_y be the crossing-arcs corresponding to arcs starting at x or ending at y, respectively. We add arcs from x^* to each target node of the arcs A_x, and arcs from each source node of the arcs A_y to y^*. We have:

Lemma 2. *The routing network R has $\mathcal{O}(|V|)$ nodes and arcs.*

Proof. Let Γ be an upward planar embedding of $G = (V, A)$ and $\hat{G} = (V, A \cup F)$ the augmented digraph of G with respect to Γ, where F is the set of sink-arcs. Since $\hat{G}$ is planar we know $|A \cup F|$ is bounded by $\mathcal{O}(|V|)$. As shown

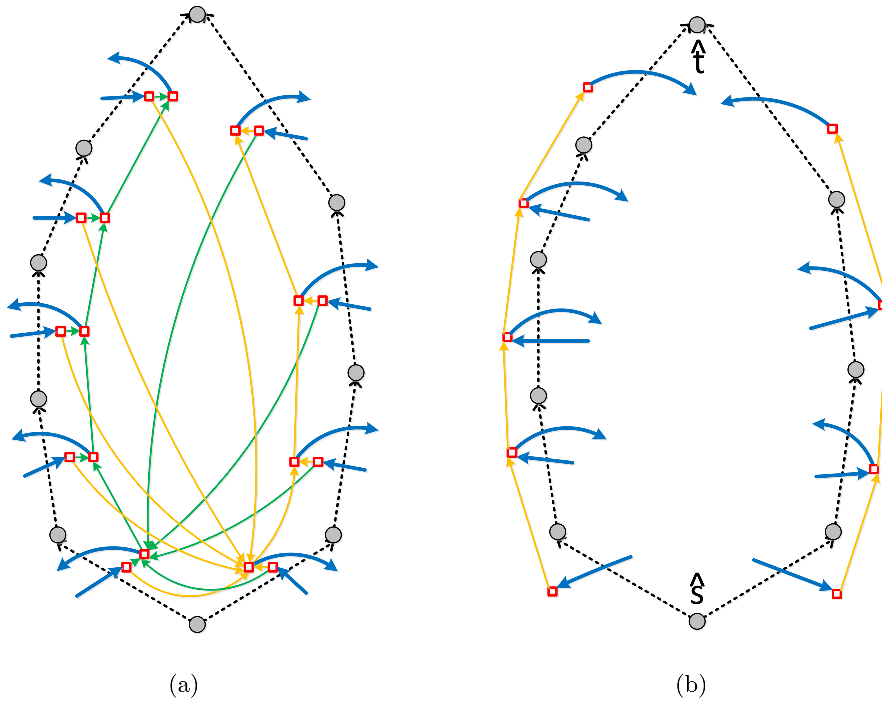


Figure 4.7: Finding feasible UIPs using the routing network R: The routing sub-network of a single internal face (a) and the external face (b). The dotted lines are the arcs of the underlying graph $\hat{U}$, the bold blue arcs are *crossing-arcs*, and the yellow and green arcs are the auxiliary routing arcs of R. Due to the augmentation, each face has exactly one sink- and one source-switch.

schematically in Figure 4.7, the network contains at most four nodes for each arc of $\hat{G}$ and the two additional nodes x^*, y^* for x, y, respectively. Hence R consists of $\mathcal{O}(|V|)$ nodes.

Every node of R, except for x^* and y^*, has at most out-degree 4. The arcs incident to x^* and y^* are limited by $\mathcal{O}(|V|)$ since the nodes of R are bounded by $\mathcal{O}(|V|)$. Hence R contains $\mathcal{O}(|V|)$ arcs. □

We assign length one to each crossing-arc which corresponds to the original arc of the input graph, that is, not to sink-arcs in $\hat{U}$ and not to arcs which are added in order to obtain an sT-graph. All other arcs in R have length zero. Notice that the assigned lengths are not final. We may change them later due to the so-called static and dynamic locks. By construction of R we can observe:

Observation 2. *Let p be a feasible UIP for $\tilde{a} = (x, y) \in B$.*

(a) There exists a path $p' = x^* \rightsquigarrow y^*$ in R corresponding to p.

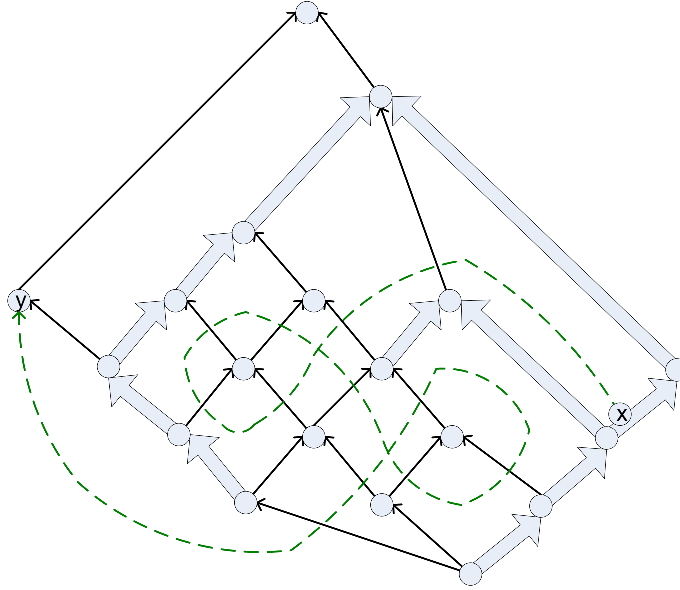


Figure 4.8: Finding feasible UIPs using the routing network: The routing in R without locking can result in infeasible insertion paths. The thick gray arcs denote substructures of $\hat{U}$ which are expensive to cross. The dashed line denotes the arc sequence p, which corresponds to a shortest path p' in the underlying routing network R, that makes loops. Thus p is an infeasible insertion path.

(b) The length of p' in R gives the number of the required crossings when realizing the corresponding insertion path p.

Locking Arcs

The routing network by itself is not strong enough to guarantee that the shortest path between x^* and y^* corresponds to a feasible UIP for (x, y). A shortest path p' in R may contain *loops*, that is, after realizing the corresponding arc sequence p of p', there exists at least two subarcs of (x, y) in the resulting intermediate UPR that cross each other; see Figure 4.8. This clearly violates the upward property of the drawing. Therefore we will introduce static and dynamic *locks*, that is, we prohibit arcs to be considered during the shortest path computation. While the static locking by itself does not directly ensures feasibility for the UIPs, it is necessary to make the dynamic locking strategy valid. So whenever we say an arc a is locked, we also mean implicit that the corresponding crossing-arcs in R are also locked. This can be achieved by setting its length to ∞.

Let Γ be a feasible upward planar embedding of FUPS U and let $\hat{\Gamma}$ be the upward planar embedding of the augmented subgraph $\hat{U}$ inducing Γ.

Lemma 3 (Static Locking)**.** *Considering the merge graph $\mathcal{M}(\hat{\Gamma})$, let Y be the dominated subgraph in $\mathcal{M}(\hat{\Gamma})$ of y; let X be the dominating subgraph in*

$\mathcal{M}(\hat{\Gamma})$ of x and let $\tilde{a} = (x, y)$ be the arc to be reinserted. A constraint feasible UIP for $\tilde{a}$ will not cross arcs of $\hat{U}$ that are also arcs of X or Y.

Proof. Assume a constraint feasible UIP p would cross an arc that connects two nodes $v_1, v_2 \in Y$. Let C be the crossing dummy node created by this crossing; it is dominated by either v_1 or v_2, hence also by y. Then through the insertion of $\tilde{a}$, we would have the path $x \rightsquigarrow C \rightsquigarrow y$ in the resulting UPR and since y dominates C, the cycle $y \rightsquigarrow C \rightsquigarrow y$ would arise. This contradicts the constraint feasibility of p. The analogous holds if $v_1, v_2 \in X$. □

We call the arcs of X and Y the arcs of the static locks. These arcs can be a priori locked before we start to compute a UIP for (x, y).

The static locks are necessary but not sufficient to prevent insertion paths containing double-loops as illustrated in Figure 4.8. We require in addition the dynamically locking of arcs during the insertion path computing phase.

Definition 11 (Face-Lock $\mathcal{L}(e)$)**.** Let e be a dynamic entrance of an insertion path p to face $f \in \hat{\Gamma}$. We denote the set of arcs of f that connect the source-switch of f to the dynamic entrance e as the *face-lock* $\mathcal{L}(e)$ with respect to p.

We observe: If an arc e is an arc of a feasible UIP p, then a successor arc of e in the arc sequence p must not be an arc of face-lock $\mathcal{L}(e)$.

Definition 12 (Dynamic Locking)**.** Let $\mathcal{A}$ be a shortest path algorithm that visits the nodes of R in the order of their distance from the start node x^*. Whenever $\mathcal{A}$ considers a path p that enters a face f through a dynamic entrance e, $\mathcal{A}$ can lock (that is, forbid) all arcs of $\mathcal{L}(e)$ after p leaves f. We refer to this procedure as *dynamic locking*.

From now on, we assume that $\mathcal{A}$ is a shortest path algorithm that utilizes the procedure of dynamic locking and we further assume that the length of the arcs of R corresponding to the static locking are set to ∞, thus in fact not used for computing UIPs.

A priori we may fear that dynamically locking arcs may prevent a shortest-path algorithm to find a feasible minimal UIP or even worse, all important arcs are dynamically locked and no feasible UIP for inserting (x, y) can be found. We will show that this is not the case. Moreover, we show that a path computed by $\mathcal{A}$ is a feasible minimal UIP for (x, y). We first show that if p is computed by $\mathcal{A}$, then it is feasible. Observe:

Theorem 1 (Di Battista and Tamassia [BT88]; Kelly [Kel87])**.** *A digraph is upward planar if and only if it is a subgraph of a planar st-graph.*

So, according to Theorem 1 we only need to prove that the graph obtained after realizing p can be augmented to a planar *st*-graph, and hence conclude p is feasible.

Lemma 4. *Let $p' = x^* \rightsquigarrow y^*$ be a path in R computed by $\mathcal{A}$. Let p be a sequence of arcs of $\hat{U}$ corresponding to p'. Then, p is an insertion path for (x, y), that is, the graph obtained after realizing p into $\hat{U}$ with respect to $\hat{\Gamma}$ is planar.*

Proof. Let $F = \langle f_1, \ldots, f_k \rangle$ be the face sequence traversed by p. Let H be the graph obtained after realizing p. Assume H would be non-planar. Since p is realized by inserting subarcs that split the faces of F, there would be a face $f \in F$ that cannot be split in a planar fashion. (p contains at least one loop.) A single line segment of p crossing through f and split it into two faces cannot lead to such a situation. Hence there have to be at least two segments of p going through f, which together contradict the planarity of the drawing and therefore cross each other (see Figure 4.9). Without loss of generality, we assume that the first segment (in traversal order of p) $\epsilon = (z_l, z_r)$ goes from point z_l on the left side of f to point z_r on the right side of f. In order to conflict with ϵ, the second crossing segment $\epsilon' = (z'_l, z'_r)$ occurring one of the following cases:

- It starts at z'_l on the left side of f above z_l and ends on the right side of f at z'_r below z_r (see Figure 4.9). But then $\mathcal{A}$ could find a shorter path which goes directly from z_l to the exit point z'_r of ϵ'.
- It starts on the right side of f above z_r and ends on the left side of f below z_l. Due to the dynamic locking, $\mathcal{A}$ has forbidden all arc crossings that can occur over arcs below z_l, hence ϵ' cannot exist.

□

Lemma 5. *Let p be an insertion path for (x, y) computed by $\mathcal{A}$. Then the graph obtained after realizing p into $\hat{U}$ with respect to $\hat{\Gamma}$ is acyclic.*

Proof. Let H be the graph obtained after realizing p into $\hat{U}$ with respect to $\hat{\Gamma}$. Assume H would be cyclic. We know from Lemma 4 that H is planar, hence the path p cannot contain loops. Due to the static locks, we know that the arcs of the dominated subgraph of y and the arcs of the dominating subgraph of x cannot be crossed. So a cycle C in H can only arise if a subpath q of p crosses a directed path $r = u \rightsquigarrow v$ in $\hat{U}$ such that q first crosses an arc b and then an arc a of r with $a \rightsquigarrow b$ (see Figure 4.10).

Let C be a smallest cycle (in terms of number of arcs) in H. Considering any upward planar drawing of $\hat{U}$, r is drawn as an upward path. Without loss of generality, we assume that p comes from the left side of r, then crosses through b, and arrives at the right side of r. Thereafter, p crosses r again at the arc a and arrives at the left side of r. Let $\langle f_1, \ldots, f_k \rangle$ denote the sequence of traversed faces when following p from x to y. Let f_l denote the right face of b and f_m the right face of a. If $f_l = f_m$, then a and b are embedded on the

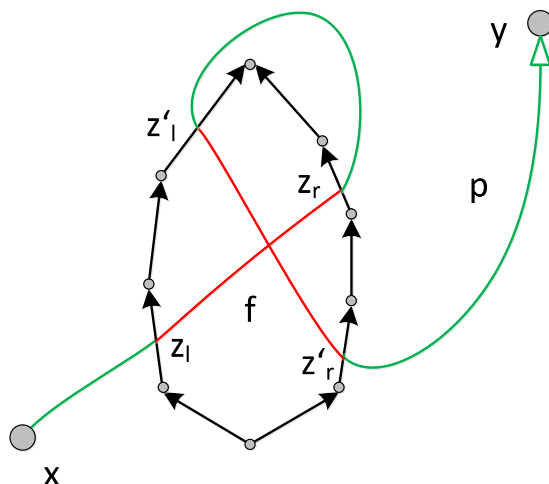


Figure 4.9: Illustration for proof of Lemma 4. The segments $\epsilon = (z_l, z_r)$ and $\epsilon' = (z'_l, z'_r)$ of p are drawn in red. The shown situation would violate the planarity of the drawing. But then $\mathcal{A}$ would have found a shorter path $x \rightsquigarrow z_l \to z'_r \rightsquigarrow y$, and hence this situation cannot occur.

same face and since a dominates b, both arcs are embedded on the same face side such that $a \in \mathcal{L}(b)$, that is, a is below b. Due to the dynamic locking, arc a cannot be an arc of p. Hence this case cannot occur, hence assume $f_l \neq f_m$.

Let f_i be a face with $l < i \leq m$. Let s be the source-switch of f_i and let p leave f_i through c. Since s is source-switch, there is a directed path $s \rightsquigarrow c$. Furthermore, since $\hat{U}$ is a single source graph, there is a path from the super source $\hat{s}$ to s. We know that s must be within the subgraph bounded by C, hence the path $\hat{s} \rightsquigarrow s$ has to cross C. Let d be the crossed arc. We can construct a cycle C^* that consists of the subpath of p which starts at c and ends at the arc d[1], and the subpath $d \rightsquigarrow s \rightsquigarrow c$. Since $f_i \neq f_l$, the cycle C^* consists of at least one arc less than C which contradicts the minimality of C. Hence H is acyclic. □

Lemma 6. *Let p be an insertion path for (x, y) computed by $\mathcal{A}$. Then p is a feasible UIP.*

Proof. Since $\hat{U}$ is an upward planar single sink sT-graph, the graph U' (with upward embedding $\hat{\Gamma}$) obtained after realizing p is also a single sink sT-graph. Further, we know from Lemma 4 and Lemma 5 that U' is planar and acyclic. We can augment U' to an st-graph by adding the arc $(\hat{s}, \hat{t})$

[1] More precisely, the subpath starts and ends at the crossing occurring in c and d, respectively.

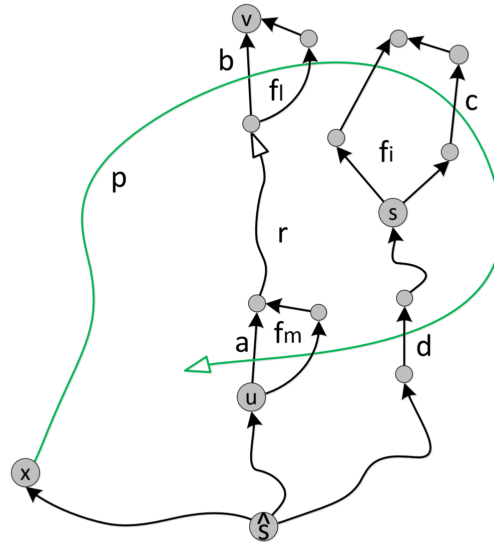


Figure 4.10: Illustration for proof of Lemma 5: The path r starts at u and ends at v. Furthermore, we have $a, b \in r$. The green line denotes the insertion path p. The subpath q of p starts at b and ends at a. By crossing b and a through p a cycle C occurs. The path p also crosses the path $\hat{s} \rightsquigarrow s$ such that a new cycle $C^* = d \rightsquigarrow s \rightsquigarrow c \rightsquigarrow d$ arises.

connecting the source $\hat{s}$ and the sink $\hat{t}$ without violating the planarity and the acyclicity of U', since they are embedded on the external face of $\hat{\Gamma}$. Thus, according to Theorem 1, U' is upward planar, therefore p is a feasible UIP by Definition 7. $\square$

If p is computed by $\mathcal{A}$, then we may fear that p it is not necessarily minimal: because of the dynamic locking, the shortest path algorithm may have be forced to use a detour. We now prove that this cannot happen.

Lemma 7. *Let p be a feasible minimal UIP for (x, y). Let F be the sequence of traversed faces when following p from x to y. Then each face in F is only traversed once by p.*

Proof. Assume there is a face $f \in F$ which is traversed more than once by p. Let p enter f the first time through the dynamic entrance e, then re-enter f one or more times, and leave f through the arc b. Since p is feasible, b is not an arc of the face-locked $\mathcal{L}(e)$. So we can construct a feasible UIP $x \rightsquigarrow a + b \rightsquigarrow y$ that is shorter than p. This contradicts the minimality of p. $\square$

Two insertion paths p and q for (x, y) cross each other if they have an arc in common or if there is a face f traversed by p and q such that a crossing

occurs in f when realizing p and q simultaneously. In the first case, we can always construct a new insertion path by concatenating appropriate subpaths of p and q. We now show, that we also can construct a new insertion path for (x, y) if q and p do not have any arc in common:

Lemma 8. *Let p and q be two feasible minimal UIPs for inserting (x, y) into $\hat{U}$ with upward planar embedding $\hat{\Gamma}$. Furthermore, let p and q cross and there exists no arc e with $e \in p$ and $e \in q$. Then, there exists a face f, where p and q enter through an arc a and c and leave through an arc b and d, respectively, such that a feasible minimal UIP $r = x \rightsquigarrow a + d \rightsquigarrow y$ can be concatenated.*

Proof. If p crosses q more than once, then the crossings can occur in several faces. Let f be the first face in the sequence of traversed faces of p where such a crossing occurs. The subpaths $x \rightsquigarrow a$ of p and $d \rightsquigarrow y$ of q are feasible minimal subpaths since p and q are feasible minimal. We know from Lemma 7 that p and q enter f only once, hence we have only the face locks $\mathcal{L}(a)$ and $\mathcal{L}(c)$. If r has to be feasible, then it must not contain any arcs of $\mathcal{L}(a)$. Considering any upward planar drawing of $\hat{U}$, no UIP for (x, y) can be routed beneath x or above y due to the static locking (Lemma 3). Each UIP that leaves f through an arc of $d \in \mathcal{L}(a)$ is routed underneath the subpath $x \rightsquigarrow a$; therefore it must cross p in order to reach y (see also proof of Lemma 9 and Figure 4.11). Assume r is infeasible, then $d \in \mathcal{L}(a)$. So q has to cross p somewhere in a face f' of $\hat{\Gamma}$. Then f' would be the first face in the sequence of traversed faces of p where p crosses q, which contradicts the assumption of f. If p crosses q only once, then f' does not exist which also leads to a contradiction. Thus r is feasible. □

Implicitly the proof reveals that if p and q cross each other more than once at some faces, then there may be a face where we cannot find appropriate subpaths of p and q in order to concatenate a new insertion path for (x, y).

We know from Lemma 6 that a path computed by $\mathcal{A}$ is feasible. We now proof that p is also minimal:

Lemma 9. *Let p be a feasible UIP for (x, y) computed by $\mathcal{A}$. Then p is a feasible minimal UIP.*

Proof. Assume p is feasible but not minimal. Then there exists a face f traversed by p such that an arc $a \in f$ is dynamically locked or cannot be used for further path computation (see Figure 4.11). Therefore, the shortest path algorithm $\mathcal{A}$ has to choose an alternative arc d that causes a detour such that p is not minimal. Let f be the first such face in the sequence of the traversed faces when following p from x to y. Let e be the entrance to f of the path p. Thus, the subpath $x \rightsquigarrow e$ of p contains no arcs which cause a detour and since it is computed by $\mathcal{A}$, the subpath is also a shortest path from x to e. Furthermore, it is feasible since p is feasible. Thus, $x \rightsquigarrow e$ is a feasible minimal subpath of p. According to Lemma 7, f is visited the first time by $\mathcal{A}$

and since dynamic locking is only applied after p leaves f (see Definition 12), no arcs of f are dynamically locked. Furthermore, the subpath $d \rightsquigarrow y$ of p is not minimal since p is by assumption not minimal.

The path p enters f through e and by the construction of R, it cannot leave f through an arc of $\mathcal{L}(e)$. Since no arcs of f are dynamically locked and since p leaves f through d and therefore make a detour, we can conclude, that a is an arc of $\mathcal{L}(e)$. Let r denote a feasible minimal UIP for (x, y) that uses a. We have two cases:

(a) The arc a is the dynamic entrance of r into f and r leaves f through an arc $b \notin \mathcal{L}(e)$. Then $b \rightsquigarrow y$ is a feasible minimal subpath of r. Since $b \notin \mathcal{L}(e)$, $\mathcal{A}$ would not take a detour using the non-minimal path $d \rightsquigarrow y$; instead it would compute the path $x \rightsquigarrow e + b \rightsquigarrow x$ which is shorter than p. Therefore this case does not arise.

(b) The path r leaves f through an arc of $\mathcal{L}(e)$. We have two subcases:

   (1) r enters f through a and leaves f through an arc of $\mathcal{L}(e)$ or

   (2) r enters f through an arc not identical to a and leaves f through a.

   In both subcases, the subpath $x \rightsquigarrow a$ of r is not shorter than the minimal subpath $x \rightsquigarrow e$ of p. Since r leaves f through an arc of $\mathcal{L}(e)$, it is routed underneath the subpath $x \rightsquigarrow e$ of p. We know from Lemma 3 that no UIPs can be routed above y and below x, so the subpath $a \rightsquigarrow y$ of r must cross the subpath $x \rightsquigarrow e$ of p. Let f' be the face where the two paths cross each other. According to Lemma 8, a feasible minimal UIP can be concatenated which consists of the minimal subpath of p from x to f' and the minimal subpath of r from f' to y. This concatenated path is shorter than r since f' is visited by p before f and the subpath $x \rightsquigarrow a$ of r is not shorter than the minimal subpath $x \rightsquigarrow e$ of p. This facts contradicts the assumption that r is minimal. So this case also not arise.

So non of the two possible cases arise, hence the assumption that p is not minimal is wrong.

□

Lemma 9 assumes that a feasible minimal UIP p can be found by $\mathcal{A}$. But due to the dynamic locking, we may fear that $\mathcal{A}$ may not always find such a UIP for (x, y). Or even worse, all important arcs may be dynamically locked and there exists no path in R that corresponds to a feasible UIP for (x, y). We now show that $\mathcal{A}$ can always find a path $x^* \rightsquigarrow y^*$ in R:

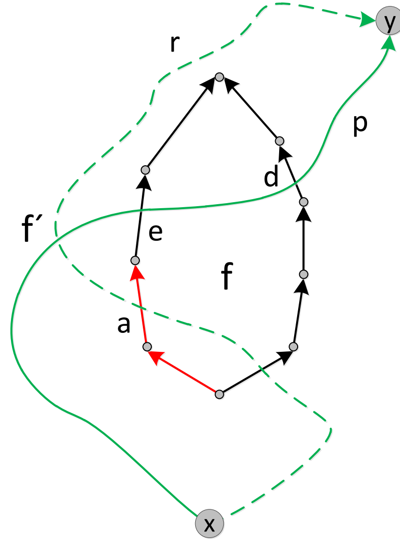


Figure 4.11: Illustration for the proof of Lemma 9: The arcs of $\mathcal{L}(e)$ are drawn in red. Due to the static locking, no UIP can be routed below x and above y, hence the path r has to cross p in order to reach y. We assume that r is minimal but not p. Since p arrives at f through e and before r, the arc a cannot be used. p is forced to make a detour by crossing d.

Lemma 10. *During the path computation of $\mathcal{A}$ from x^* to y^* there exists—at any time—a path $p' = x^* \rightsquigarrow y^*$ in R which contains no crossing-arcs corresponding to any dynamically locked arcs and the corresponding path p of p' is a feasible UIP for (x, y). Furthermore, $\mathcal{A}$ always computes a path from x^* to y^*.*

Proof. Let $\mathcal{A}$ currently consider the path $r' = x^* \rightsquigarrow a'$ in R where a' is a crossing arc of a. Let f be the face in which the corresponding path r of r' enters it through a. (Or in other words, a is the dynamic entrance of r to f.) Due to the dynamic locking procedure, the path r does not contain any locked arcs. According to Lemma 6 and Lemma 9, r is a feasible minimal UIP from x to arc a. We know from Lemma 7 that f cannot have been traversed before by r, so r enters the face f through a for the first time. Therefore no arcs of f are dynamically locked due to the path computation of r. But there maybe dynamically locked arcs of f due to other alternative paths that traverse f. We show now that this is not the case: The dynamic locking is only performed when the currently considered path leaves f (see Definition 12). So if some arcs of f are dynamic locked, then there must be an insertion path that enters f through an arc and leaves f through another

arc. But since $\mathcal{A}$ visits the nodes of R in the order of their distance to x^*, the corresponding path in R of such a path would be considered after the path r' has been considered by $\mathcal{A}$. Hence we can assume that no arcs of f are dynamically locked.

The shortest path algorithm $\mathcal{A}$ can add an arc $e \in f$ that is not in $\mathcal{L}(a)$ to extend r, leave f through e and dynamically lock the arcs of $\mathcal{L}(a)$. We know that due to the static locking $y \not\leadsto e$ in $\hat{U}$ and $\mathcal{M}(\hat{\Gamma})$ and hence by Observation 1, there exists an upward drawing with $e \prec y$ which preserves the embedding $\hat{\Gamma}$ of $\hat{U}$. Therefore there also exists a feasible minimal UIP q from e to y and by Observation 2, there exists a path $q' = e' \leadsto y^*$ in R. Since $\mathcal{A}$ visits the node of R in the order of the their distances to x^*, non of the crossing-arcs of q' have been considered yet. So no arcs of q corresponding to q' are dynamically locked. Thus $r + q = x \leadsto a + e \leadsto y$ does not contain any dynamically locked arcs and since $e \notin \mathcal{L}(a)$ by the construction of R, the concatenated path $r + q$ is also feasible. Notice that $r + q$ is not necessarily minimal since an insertion path for (x, y) that first visits f and then from there the node y^* can be a detour. Since there exists—at any time—a path $p' = x^* \leadsto y^*$ in R, $\mathcal{A}$ always computes a path from x^* to y^*. □

We summarize our result in the following theorem:

Theorem 2. *Let R be a routing network constructed with respect to $\hat{\Gamma}$ to insert (x, y) into $\hat{U}$. Let $\mathcal{A}$ be a shortest path algorithm that uses static and dynamic lockings and visits the nodes of R in the order of their distance from the start node x^*. Then $\mathcal{A}$ computes a path $p' = x^* \leadsto y^*$ in R that corresponds to a feasible minimal UIP p for (x, y).*

Proof. According to Lemma 10, a path $p' = x^* \leadsto y^*$ can always be found by $\mathcal{A}$ and according to Lemma 9, the corresponding path p is a feasible minimal UIP for (x, y). □

Upward Arc Insertion Algorithm

Feasible minimal UIP computation. Based on the above routing network R and Theorem 2, we can compute a feasible minimal UIP using a BFS algorithm, where arcs of R are assigned zero or one and arcs corresponding to locked arcs are assigned ∞ length. We called the algorithm `FeasibleMinUIP`. Its pseudo-code is given in Algorithm 3. The algorithm starts with the construction of R (line 2-3) and locks arcs specified by the static locks (line 10-12). Furthermore, it dynamically forbids additional arcs: Whenever a crossing-arc e' is added to the currently considered path, the corresponding arc $e \in \hat{U}$ is a dynamic entrance to a corresponding face f with $e \in f$, and hence all arcs of face-lock $\mathcal{L}(e)$ are locked (line 28-30) after f is left.

The procedure `ExtractIP` (line 37) constructs the feasible minimal UIP p by reconstructing the computed path $p' = x^* \leadsto y^*$ using the predecessor

Algorithm 2 `Relax`: subprocedure of `FeasibleMinUIP` (Algorithm 3) and `ConstraintFeasibleUIP` (Algorithm 4)

Require: node u, v
1: **if** $d[v] + \ell((u,v)) < d[v]$ **then**
2: $d[v] := d[v] + \ell((u,v))$
3: $\text{Pred}[v] = u$
4: **return true**;
5: **end if**
6: **return false**;

matrix *Pred* and then deletes the auxiliary routing arcs in p'. Thereafter p' consists of a sequence of crossing-arcs which corresponds to a sequence p of arcs of $\hat{U}$.

Corollary 2. *The algorithm* `FeasibleMinUIP` *computes a feasible minimal UIP p for (x, y).*

Corollary 3. *Let p be a feasible minimal UIP obtained by* `FeasibleMinUIP`*, and Γ' the upward planar embedding arising from realizing p. If the merge graph $\mathcal{M}(\Gamma')$ is acyclic, p is a constraint minimal UIP.*

Constraint feasible minimal UIP computation. There may be a situation when the computed minimal insertion path is not constraint feasible. In such cases we have to resort to a heuristic for finding a constraint feasible UIP:

The algorithm `ConstraintFeasibleUIP` (Algorithm 4) works similar to `FeasibleMinUIP` but uses no dynamic locking and instead whenever the BFS algorithm relaxes some crossing-arc b', it tests each time whether the corresponding arc b can be added to the current considered UIP q without violates the constraint feasibility (line 8). This test is done by constructing an *intermediate merge graph* $\mathcal{M}(\Gamma_C)$ (line 4-7) by partially realizing of $\tilde{a}$ along the UIP q up to b such that the inserted path ends at a new dummy node ξ (line 6). Then the merge graph $\mathcal{M}(\Gamma_C)$ for this graph is built by adding the arc (ξ, y) instead of (x, y) to $\mathcal{M}(\Gamma_C)$ (line 7). If the test is positive, that is, $\mathcal{M}(\Gamma_C)$ is acyclic, then q is extended by adding b to it (line 9), otherwise the considered arc b is forbidden in the BFS enumeration (line 11).

`ConstraintFeasibleUIP`—though always terminating—will in general not give an optimal solution, as an alternative path up to the rejected arc b might have allowed us to use b and find an overall shorter path.

Lemma 11. *The algorithm* `ConstraintFeasibleUIP` *computes a constraint feasible UIP p for (x, y).*

Algorithm 3 `FeasibleMinUIP`

Require: $\tilde{a} = (x, y)$, U, Γ
Ensure: Feasible minimal UIP for (x, y)

```
 1: ▷ routing network construction
 2: augment U → result: Û, Γ̂
 3: construct R with respect to Γ̂
 4: ▷ Initialization
 5: for each node v of R do
 6:     d[v] := ∞ ▷ distance matrice
 7:     Pred[v] := ∅ ▷ predecessor
 8:     Visited[v] :=false
 9: end for
10: for each statically locked arc a do
11:     set the length ℓ of the crossing-arcs of a to ∞
12: end for
13: Queue Q:= ∅;
14: Q.enqueue(x*)
15: Visited[x*] :=true
16: d[x*] := 0
17: ▷ BFS traversing of R
18: while Q ≠ ∅ do
19:     node u :=Q.dequeue()
20:     for each target node v of u do
21:         if not Visited[v] then
22:             arc b' := (u, v)
23:             if b' is not a crossing-arc then
24:                 Relax(u, v)
25:             else
26:                 if Relax(u, v) then
27:                     arc e := predecessor arc of b in the current path
28:                     for each a ∈ ℒ(e) do
29:                         set the lengths of the crossing-arcs of a to ∞
30:                     end for
31:                 end if
32:             end if
33:             Visited[v]:=true
34:             Q.enqueue(v)
35:         end if
36:     end for
37: end while
38: p:=ExtractIP(Pred[y*]) ▷ extract insertion path
39: return p
```

Algorithm 4 `ConstraintsFeasibleUIP` (fragment); This pseudocode replaces the *else*-block (line 25–31) of Algorithm 3 (`FeasibleMinUIP`)

Require: $\tilde{a} = (x, y)$, U, Γ
Ensure: ConstraintFeasible UIP for (x, y)

1: ...
2: **else**
3: arc b :=corresponding arc of b' ▷ recall: $b \in \hat{U}$ and $b' \in R$
4: q:=`ExtractIP`(Pred[v]) ▷ current insertion path to b
5: C :=copy of G; embed C like $G \rightarrow \Gamma_C$
6: realize q in C with respect to $\Gamma_C \rightarrow$ dummy ξ
7: replace (x, y) by (ξ, y) and construct $\mathcal{M}(\Gamma_C)$
8: **if** $\mathcal{M}(\Gamma_C)$ is acyclic **then**
9: `Relax`(u, v) ▷ q is a constraint feasible UIP
10: **else**
11: $\ell(b) := \infty$ ▷ lock b
12: **end if**
13: **end if**
14: ...

Arc reinsertion strategy. Algorithm 5 gives an overview on the overall arc reinsertion strategy: we try to add all arcs using the minimal path computed by `FeasibleMinUIP`. Only if there is no more arc insertable by it (no constraints feasible minimal UIP can be found for the remaining arcs), we insert a not-yet inserted arc using `ConstraintFeasibleUIP`. Afterwards we again try to use `FeasibleMinUIP` for the remaining arcs. We iterate that process until all arcs are inserted. As we will see in the experimental evaluations, the heuristic procedure `ConstraintFeasibleUIP` is only used very rarely; hence in most cases all arcs are inserted according to their corresponding feasible minimal UIP.

4.2.4 Runtime Analysis

We conclude the theoretic description by analyzing the algorithms' runtimes.

Lemma 12. *Let $G = (V, A)$ be an sT-graph. Algorithm 1 computes a FUPS U of G with a feasible embedding Γ in $\mathcal{O}(|A|^2)$ time.*

Proof. Since we only have to consider connected digraphs, we have $|V| \leq |A| + 1$. The computation of the spanning tree of G can be done in $\mathcal{O}(|A|)$, and upward planarity testing of sT-graphs requires $\mathcal{O}(|V|)$ time (Bertolazzi et al. [BDMT98]). The construction and cycle testing of the merge graph $\mathcal{M}(\Gamma)$ within the *for*-loop can be done in $\mathcal{O}(|A|)$. In the worst case we have to test $\mathcal{O}(|A|)$ arcs and hence obtain the above lemma. □

Algorithm 5 Reinsert all arcs (cf. Algorithm 1 (`LFUP`) line 15-19)

Require: sT-graph $G = (V, A)$, FUPS $U = (V, A')$ with feasible upward embedding Γ
Ensure: UPR U^* of G with embedding Γ^*, inducing Γ

```
 1: List L := A \ A'
 2: U* := U, Γ* := Γ
 3: while L not empty do
 4:     boolean success:=false
 5:     for each ã ∈ L do
 6:         p := FeasibleMinUIP (ã, U*, Γ*)
 7:         U°, Γ° := realizePath(p, U*, Γ*)
 8:         if M(Γ°) acyclic then                ▷ p was constraint feasible
 9:             U* := U°, Γ* := Γ°
10:             success:=true
11:             L.remove(ã)
12:         end if
13:     end for
14:     if not success then
15:         ã := L.extractRandomElement()
16:         p :=ConstraintFeasibleUIP(ã, U*, Γ*)
17:         U*, Γ* :=realizePath(p, U*, Γ*)
18:     end if
19: end while
```

Lemma 13. *Let $\bar{U} = (\bar{V}, \bar{A})$—with upward planar embedding $\bar{\Gamma}$—be the intermediate UPR obtained after inserting some arcs into the original FUPS U of G. Let (x, y) be the next arc to be inserted, and let r be the number of arcs to be inserted afterwards.*

(a) Computing a feasible minimal UIP for (x, y) via `FeasibleMinUIP` requires $\mathcal{O}(|\bar{V}| + r)$ time.

(b) `ConstraintFeasibleUIP` computes a constraint feasible UIP for (x, y) in $\mathcal{O}(|\bar{V}|^2 + r|\bar{V}|)$ time.

Proof. By Lemma 2, the routing network for $\bar{U}$ with respect to $\bar{\Gamma}$ can be constructed in $\mathcal{O}(|\bar{V}|)$ time. To compute the static locks we have to construct the merge graph $\mathcal{M}(\bar{\Gamma})$, which requires $\mathcal{O}(|\bar{V}| + r)$ time. The runtime of the BFS algorithm, including the computation of the dynamic locks, is bounded by $\mathcal{O}(|\bar{V}|)$. For the constraint feasibility checking, we have to (temporarily) insert the insertion path into $\bar{U}$, construct the corresponding merge graph, and test for acyclicity. This can be done in $\mathcal{O}(|\bar{V}| + r)$ time. Thus the total runtime of *(a)* is dominated by $\mathcal{O}(|V| + r)$.

The runtime analysis of ConstraintFeasibleUIP is similar. Instead of computing the dynamic locks, we temporarily insert the current insertion path into $\bar{U}$ after each arc relaxation, and check for acyclicity of the corresponding merge graph. Hence the runtime is $\mathcal{O}\left(|\bar{V}| \cdot (|\bar{V}| + r)\right)$. □

Using these two lemmata, and since each arc insertion step generates a linear number of additional dummy nodes, we can directly obtain a generous runtime bound:

Theorem 3. *Let $G = (V, A)$ be any connected DAG.* `LFUP` *computes a UPR of G in $\mathcal{O}(|A|^5)$ time.*

Proof. Clearly, $|V| = \mathcal{O}(|A|)$. Starting with a FUPS of size $\mathcal{O}(|V|)$, we have to insert at most $\mathcal{O}(|A|)$ arcs. After each insertion step, the digraph into which we insert the arcs grows by at most $\mathcal{O}(|A|)$ dummy nodes. Hence after all arc insertions the size of the planarization can be bounded by $\mathcal{O}(|A|^2)$.

For each arc insertion step we may have to run `FeasibleMinUIP` for each not-yet-inserted arc but in the end resort to `ConstraintFeasibleUIP`. The latter dominates the runtime of the insertion step. A single arc insertion might hence require up to $\mathcal{O}(|\bar{V}|^2 + |A| \cdot |\bar{V}|)$ time. With $|\bar{V}| = \mathcal{O}(|A|^2)$ we obtain $\mathcal{O}(|A|^4)$ for a single insertion step. As the computation of the FUPS is dominated by the $\mathcal{O}(|A|)$ insertion steps, we obtain an overall running time of at most $\mathcal{O}(|A|^5)$. □

Although the overall runtime of $\mathcal{O}(|A|^5)$ of `LFUP` seems too high for a practical algorithm, the experimental evaluations in the upcoming subsection exhibit that the runtime bound of $\mathcal{O}(|A|^5)$ is a rough estimate. It turns out that `LFUP` is very practical even for digraphs with size $|V| = 100$.

4.3 Experimental Evaluation

We have implemented the new layer-free upward planarization approach using the open-source C++-library *Open Graph Drawing Framework (OGDF)* [ogd] which is available under the general public license (GPL) and have compared its performance with state-of-the-art algorithms based on four benchmark sets.

4.3.1 Benchmark Sets

Rome graphs. The Rome graphs (Di Battista et al. [DGL+97]) are a widely used benchmark set in graph drawing, obtained from a basic set of 112 real-world graphs. It contains 11,528 instances with 10–100 nodes and 9–158 edges and with density 0.9–2 and average density of 1.29. Although the graphs are originally undirected, they have been used as directed graphs by artificially directing the edges according to the node order given in the input files. Hence all edges are directed and the graphs are acyclic.

Algorithm 6 Random DAG

Require: number of nodes n, probability p
1: $G = (V = \{v_1, \ldots, v_n\}, A = \emptyset)$
2: **for** $i = 1, \ldots, n$ **do**
3:     **for** $j = 1, \ldots, i-1$ **do**
4:         **if** UniformRandom[0..1[$< p$ **then**
5:             $A := A \cup \{(v_j, v_i)\}$
6:         **end if**
7:     **end for**
8: **end for**

North DAGs. The North DAGs have been introduced in an experimental comparison of algorithms for drawing DAGs by Di Battista et al. [DGL+00]. The benchmark set contains 1,277 DAGs collected by Stephen North which were slightly modified by Di Battista et al. Since the North DAGs are a collection of heterogeneous digraphs, that is, the density of the digraphs with same number of nodes may vary from very dense to very sparse, the instances are grouped into 9 sets, where set i contains graphs with $10i$ to $10i + 9$ arcs for $i = 1, \ldots, 9$. Hence we do not consider all 1,277 but 1,158 DAGs.

Random DAGs The real-world origin of the above benchmarks results in sets where, for example, the relative graph densities are not uniformly distributed over the different graph sizes; in particular larger graphs tend to have lower density. This can make it hard to interpret the algorithm's performance with respect to graph density. Therefore we also consider a set of random DAGs: All graphs have 100 nodes and each potential arc occurs with uniform probability p: Each DAG $G = (V, A)$ is generated using Algorithm 6 and suitable p, such that we obtain the expected density $\varrho = |A|/|V| = \{1.5, 2, 2.5, \ldots, 6\}$. We generate 20 random DAGs (ignoring runs where the algorithm gives a disconnected graph) for each ϱ, resulting in 200 connected random DAGs overall. Note that the density of the Rome and the North instances are below 2–3.

DAGs with known upward crossing number $ucr(G)$. To the best of our knowledge, only trivial exponential approaches for computing the upward crossing number of a DAG are known, but there exist ILP-based approaches for computing the crossing number $cr(G)$ of undirected graphs, and for many instances of the Rome graphs, the crossing number is already known (Chimani [Chi08]). We derive a set of DAGs with known upward crossing number from these Rome instances as follows: We first compute a drawing $\mathcal{D}$ of an instance G such that the number of edge crossings occurring in $\mathcal{D}$ is equal to $cr(G)$. Then we direct each edge e with end node u and v such that (u, v) if $Y(u) < Y(v)$ and (v, u) otherwise. In the new drawing there may be arcs

which are not drawn in an upward fashion. For example, a drawing of e may contain a segment ϵ_1 which is drawn upward firstly then makes a downward turn or a segment ϵ_2 which is drawn downward firstly then makes an upward turn. In this case, we manually replace ϵ_1 by a "peak" $\langle (u,w),(v,w) \rangle$ and ϵ_2 by a "V" $\langle (w,u),(w,v) \rangle$ by introducing new a node w. By these manual modifications, the number of crossings occurring in $\mathcal{D}$ is not violated and $\mathcal{D}$ becomes an upward drawing. However, the number of nodes can increase.

The set of DAGs with known upward crossing number consists of 20 DAGs partitioned into five groups. Each group consists of four instances with 20, 40, 60–61, 80–81, and 100–101 nodes, respectively. The instances of the groups are derived from the Rome graphs with $|V| = 20, |V| = 40, |V| = 60, |V| = 80$, and $|V| = 100$ respectively. For the latter three groups, we have added a new node to some instances in order to preserve the upward property. The upward crossing numbers vary from 1 to 22.

4.3.2 Evaluated Algorithms

We have considered the following algorithms:

`LFUP`: In our implementation of `LFUP`, we randomize the order of the arcs considered in the for-each loop (line 4 of Algorithm 1) of the FUPS computation, and the order in which arcs are reinserted (*for-each* loop in line 5 of Algorithm 5). We denote by `LFUP`i the best result obtained after i independent random runs of the algorithm.

`Sugiyama`: The graph drawing library OGDF contains an implementation of the traditional `Sugiyama`. The chosen settings are: `GKNV-Layering`, `Barycenter` with `GreedySwitch` post-processing. A randomized run start with a random order of the nodes on each layer. After each up- and down-sweep `GreedySwitch` is applied to each layer in order to improve the results. A randomized run is ended when no further crossing reduction can be achieved. We denote by `Sugiyama`i the best results obtained after i independent random runs.

`Dot and Layers`: In the experimental study of Di Battista et al. [DGL+00], the algorithms `Layers` and `Dot` turned out to be the most successful ones. `Layers` is an implementation of Sugiyama's algorithm according to the original paper [STT81] and `Dot` is a highly-optimized version of this algorithm developed by Koutsofios and North (see also [GKNV93]). We report on their results as they were published in [DGL+00], hence the evaluation of `Dot` and `Layers` refers only to the North DAGs. In [DGL+00], two further algorithms where considered, using a simple method based on planarization of *st*-graphs. (The planarization algorithms follow the idea published in [BPTT89].) We omit these in the diagrams, as they perform very poorly, achieving roughly 300 crossings on average for the largest North instances, that is, $90 \leq |A| \leq 99$.

MUP: We also compare LFUPi with the mixed-upward planarization approach as presented in [EKE03]. Again, due to the code's unavailability, we report on the published results for the comparison, hence the evaluation of MUP only refers to the Rome graphs.

Global-Sifting and Grid-Sifting: The authors of these algorithms have provided us with their experimental data as reported in the publication [BBG11]. Although their experiments are applied to two sets of benchmark graphs—the Rome graphs and a random DAG set—, we ignore the data of the latter set which contains very dense DAGs with $|V| \leq 400$ and $|A| \leq 4 \cdot |V|$, since even using state-of-the-art algorithms for drawing such instances, the drawing quality not significantly improves. (As reported in the technical report [BBG11], a drawing of a such graph can contain up to 100,000 crossings.)

The authors use the following settings: Global-Sifting always start with an initial layering where each node is assigned to one layer, hence the number of layers is $|V|$. It uses the best results of 400 sifting rounds. A sifting round computes for each block[2] a local optimal position. After a round, the obtained order is fixed and the algorithm again start to compute for each block a local optimal position and so on. The results obtained by i sifting rounds is denoted by Global-Sifting$i$.

Grid-Sifting allows the user to control the runtime by a parameter called *radius*. When setting the radius to r, then a node/block b is vertically sifted only between the layer L_{i+r} and L_{i-r} where L_i is the layer where b is assigned to. Due to the long runtime, the number of sifting rounds is set to 8. Furthermore, as reported by the authors, increasing the sifting round beyond 8 does not significantly improve the results [BBG11]. We denote with Grid-Siftingr/i the results obtained after i sifting rounds with radius r. In the case when the radius is not limited, we set $r := *$. In the evaluation we consider Grid-Sifting$10/8$ and Grid-Sifting$*/8$.

So we can hence only apply LFUPi and Sugiyama to all the benchmark sets, but we can, for example, assume that Sugiyama and Dot would behave roughly equivalently, as they are virtually indistinguishable on the known common benchmark set (see Figure 4.14).

4.3.3 Comparison

Due to the code's unavailability of Dot, Layers, MUP, Global-Sifting, and Grid-Sifting, the plots are presented with respect to the corresponding publications. As results, the Rome graph are grouped according to the number

[2]Recall, a block is either a regular node or a maximum connected subgraph of long arc dummies in the layering.

of nodes and the North DAGs according to the number of arcs. Each data point of the plots refers to the corresponding average value of all the graphs within the same group.

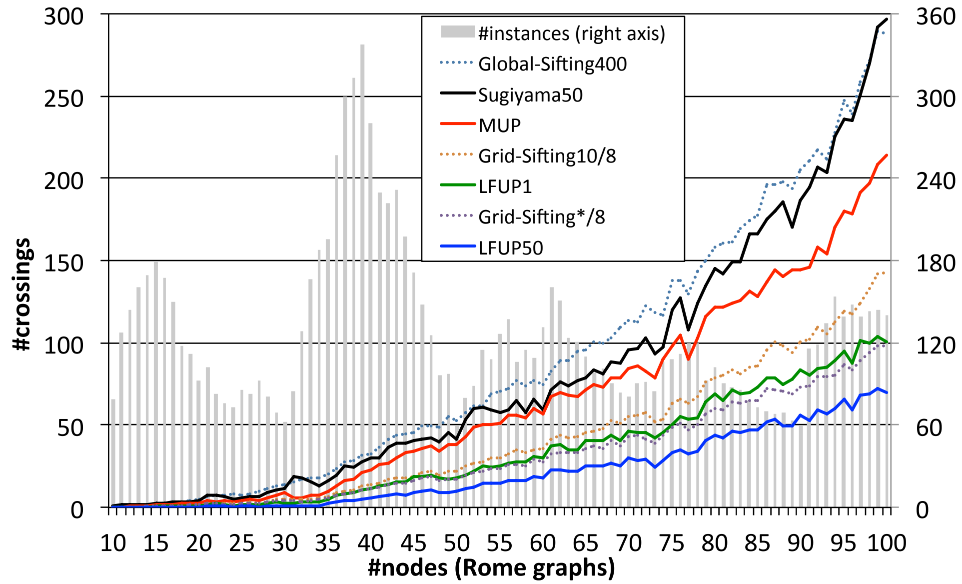
Rome graphs. Considering the Rome graphs, Figure 4.12 shows the results for MUP, OGDF's Sugiyama, Global-Sifting and Grid-Sifting, and the new approach LFUP. We observe: the classical Sugiyama50 clearly outmatched Global-Sifting400 on most of the Rome instances. This is quite surprisingly, since in the publications [MSM99] and [BBBH10] the authors reported that Global-Sifting performs quite well. On the considered benchmark sets of their experimental evaluations Global-Sifting clearly outmatched Sugiyama.

As expected, the quality of the results of Grid-Sifting increases with increasing radius. Though MUP is already considerably better than Sugiyama50, LFUP1 and Grid-Sifting*/8 obtain solutions with only half as many crossings as MUP. The performance of Grid-Sifting*/8 is quite similar to LFUP1; while LFUP1 produces slightly better results for small graphs ($|V| \leq 50$), Grid-Sifting*/8 produces better results for graphs with $|V| > 50$. The best results is obtained by LFUP50. In comparison to Sugiyama50 on the graphs with $|V| < 40$ and $|V| \geq 40$, the average number of crossings can be reduced by 90% and 70% on average, respectively.

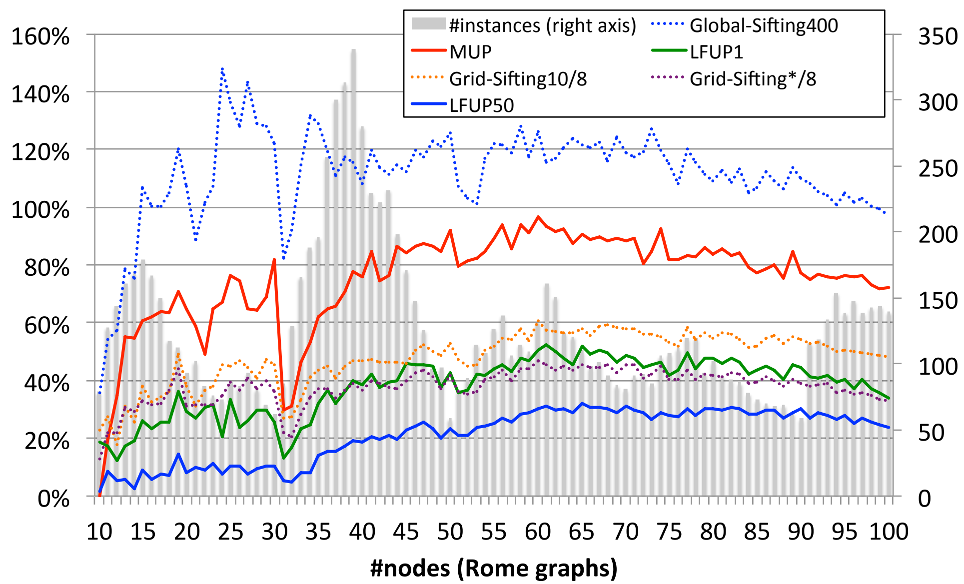
Figure 4.13 gives a comparison of the two best algorithms: LFUP50 and Grid-Sifting*/8. Again, we can see that LFUP50 achieves the most improvement on small graphs. With increasing number of nodes, the relative gap between LFUP50 and Grid-Sifting*/8 is getting smaller. We can observe that LFUP50 has clearly outmatched Grid-Sifting*/8.

North DAGs. Figure 4.14 illustrates the central results for the North DAGs. Again, the new algorithm LFUP clearly outperforms the three layer-based algorithms Layers, Dot, and Sugiyama, leaving them far behind. While OGDF's Sugiyama50 achieves virtually the same results as Dot, LFUP1 obtains solutions with roughly half as many crossings; the randomized version with multiple runs LFUP50 again yields significant improvements, especially for larger graphs. LFUP50 achieves results which have on average only 40% of the crossings of Sugiyama50.

Random DAGs. The results of the random DAGs are illustrated in Figure 4.15. First of all, LFUP clearly outperforms Sugiyama. Furthermore, with increasing density of the instances, the number of crossings generally increases which was expected. We can observe that on the one hand, the absolute discrepancy between Sugiyama and LFUP increases when the instances becoming denser (Figure 4.15(a)). On the other hand, the relative gap between LFUP and Sugiyama50 decreases (Figure 4.15(b)). It converges to 22% and 28% for LFUP50 and LFUP, respectively. Regarding the impact of random runs on



(a) Average number of crossings vs. number of nodes.



(b) Average number of crossings in relation to `Sugiyama50` (=100%).

Figure 4.12: Rome graphs.

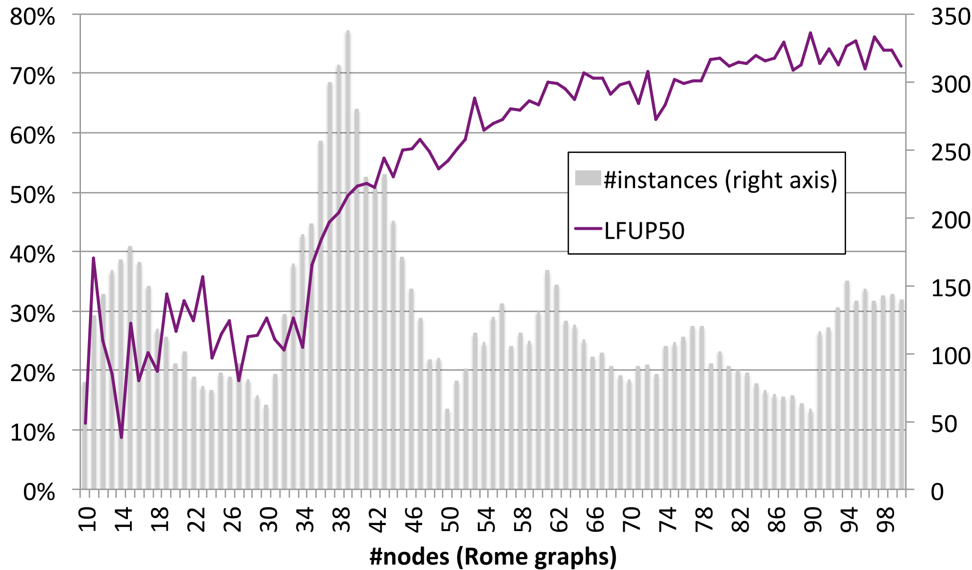


Figure 4.13: Average number of crossing of `LFUP50` in relation to `Grid-Sifting`*/8 (=100%).

the quality of the results, with increasing density, it decreases, but the relative gap between the results of one simple run and 50 random runs is still significant. It converges to 5% with increasing density.

GD 2008 Challenge Graphs

The topic of the 2008 Graph Drawing Challenge (Dogrusoz, Duncan, Gutwenger, and Sander [DDGS08]) was upward crossing minimization. The competition was won by Team Dortmund using the OGDF implementation of the `LFUP`. From the six challenge DAGs, this implementation achieved the smallest number of crossings for each DAG.

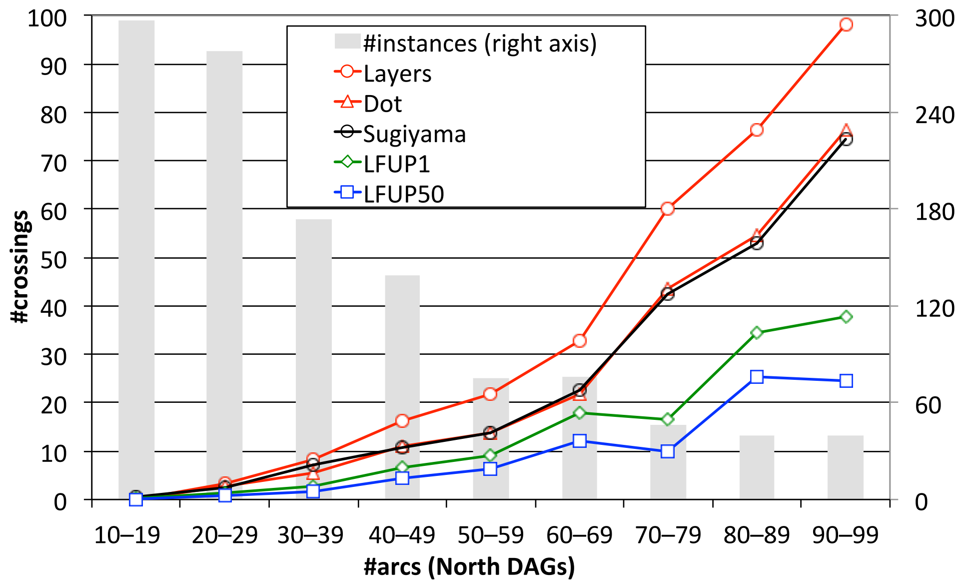
Table 4.1 shows the results for the 6 DAGs, both for `LFUP20` and OGDF's `Sugiyama50` algorithm; we observe that `LFUP20` achieves clearly smaller number of crossings, improving the `Sugiyama50` results by 25–82.9%.

4.3.4 Deeper Analysis

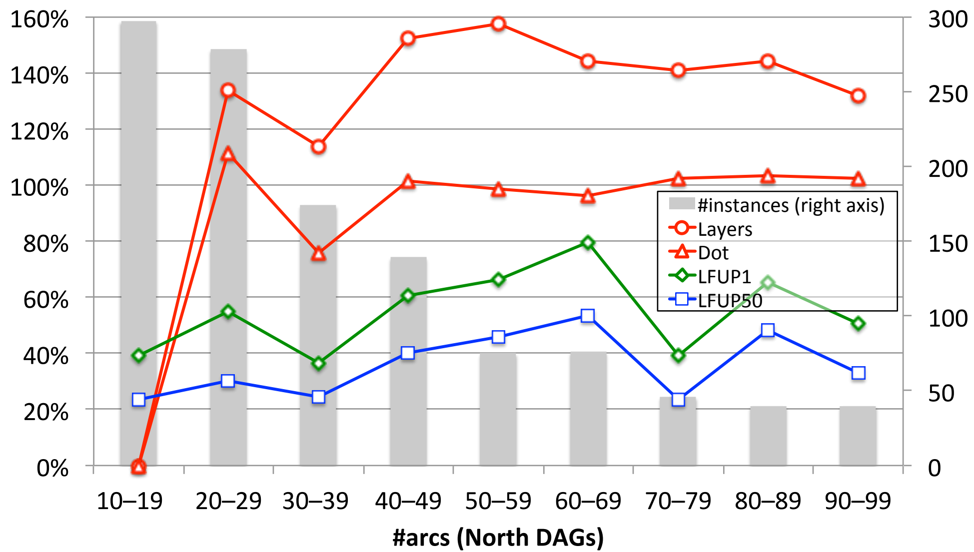
We give here a deeper evaluation of the layer-free upward crossing minimization approach.

Dependency on the Number of Crossings

We examine the benefit of using multiple runs with respect to the number of crossings obtained by applying `LFUP50`; see Figures 4.16(a), 4.16(b) and 4.17

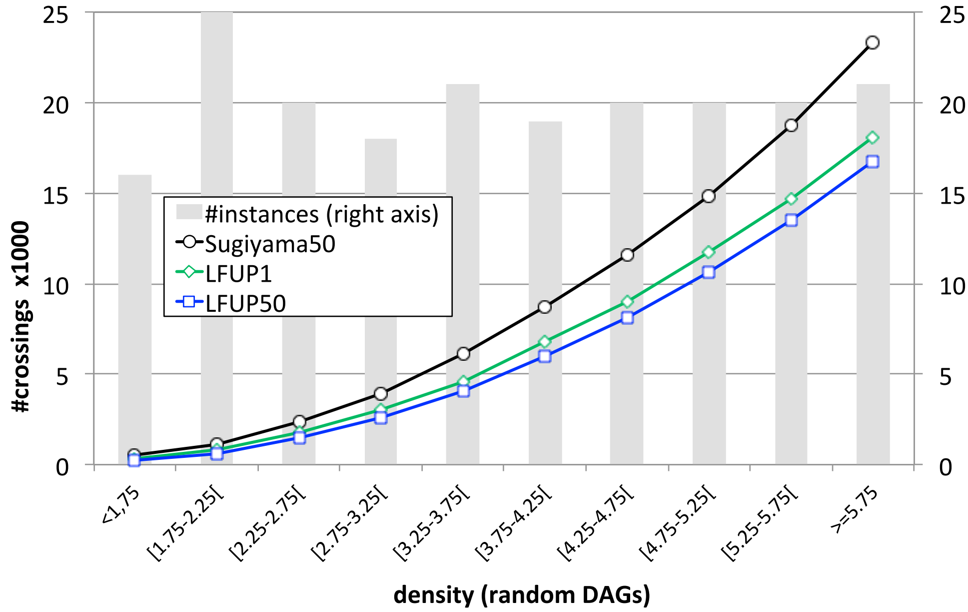


(a) Average number of crossings vs. number of arcs.

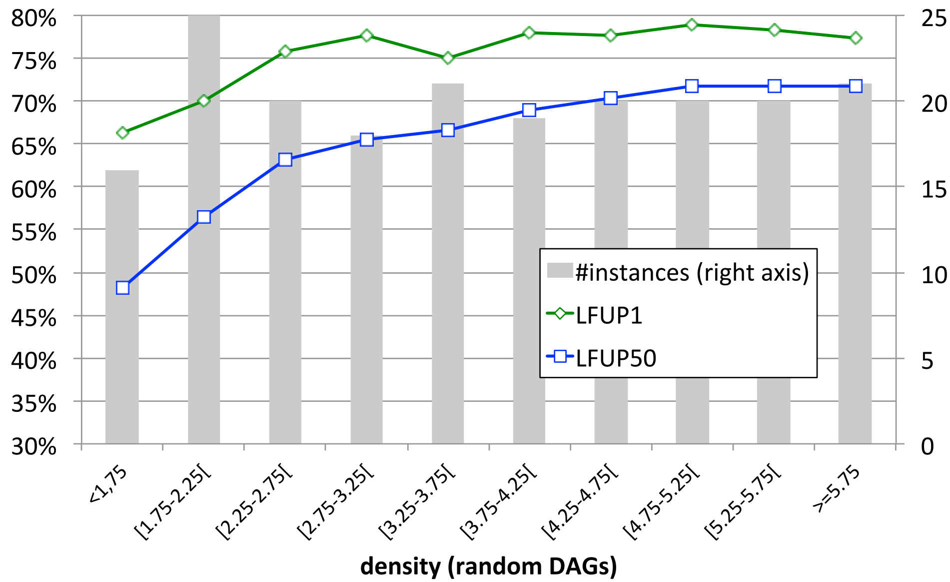


(b) Average number of crossings in relation to `Sugiyama50` (=100%). Due to the measuring inaccuracy, the data points of `Dot` and `Layers` in the first group (#arcs 10–19) is set to zero.

Figure 4.14: North DAGs.



(a) Average number of crossings vs. density of the graphs.



(b) Average number of crossings in relation to `Sugiyama50` (=100%).

Figure 4.15: Random DAGs.

graph	nodes	arcs	number of crossings LFUP20	Sugiyama50	*Impr*
g1	24	46	4	12	66.7%
g2	99	157	18	24	25.0%
g3	159	241	30	175	82.9%
g4	248	356	108	236	54.2%
g5	729	912	58	113	48.7%
g6	993	1383	295	802	63.2%

Table 4.1: GD 2008 Challenge graphs. *Impr* denotes the relative improvement of LFUP20 compared to Sugiyama.

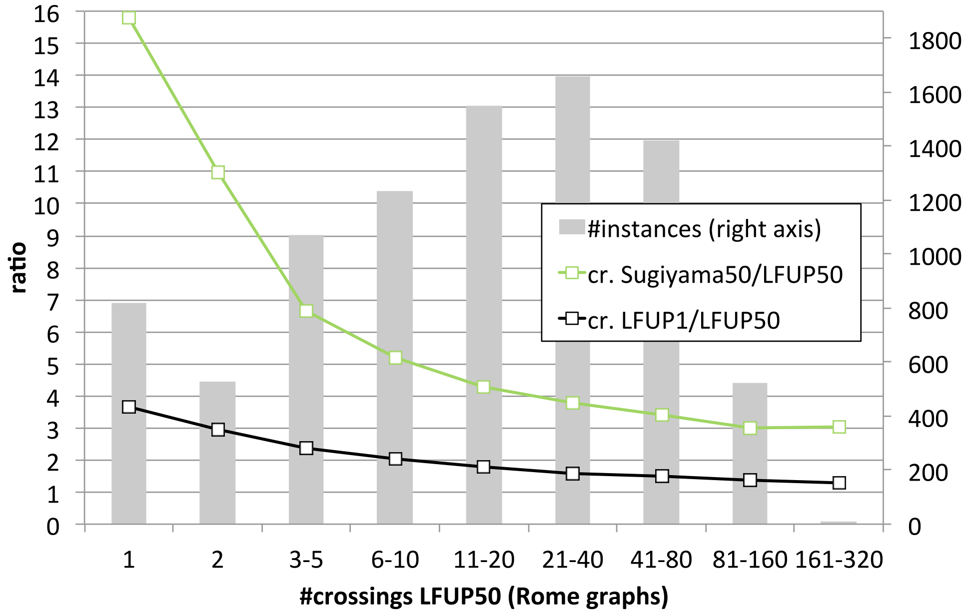
for the Rome graphs, the North DAGs and the random DAGs, respectively. The diagrams show the ratio between the number of crossings achieved by Sugiyama and LFUP50, and between LFUP1 and LFUP50. We observe that the gap is particularly large for small number of crossings. While the gap between Sugiyama50 and LFUP50 is still relevant for large crossing numbers, the benefit of multiple runs within the LFUP algorithm diminishes nearly. So we observe that the effectiveness of multiple runs and LFUP decrease with increasing number of crossings.

Dependency on the Number of Random Runs

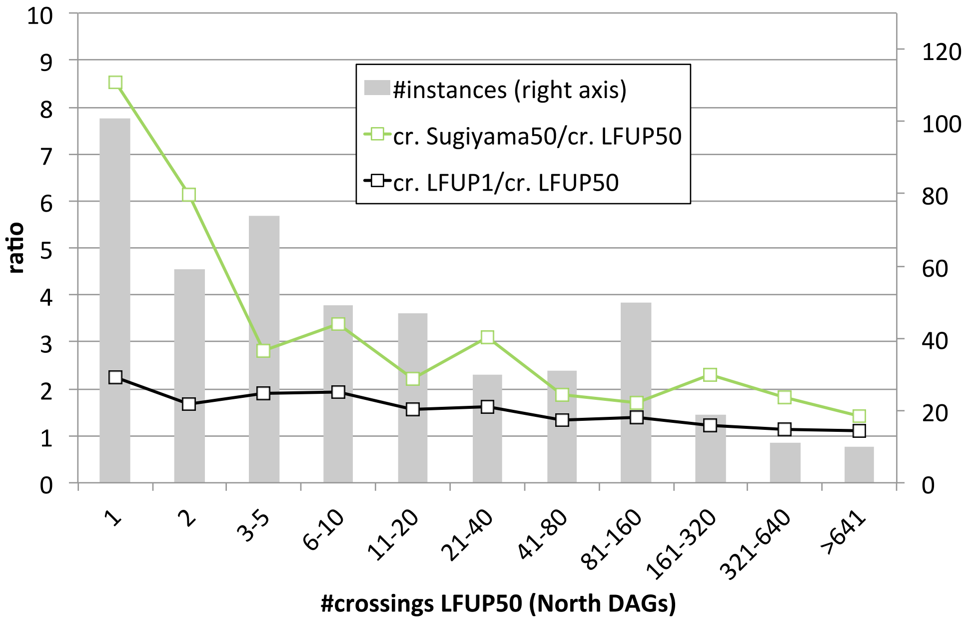
The plots in Figure 4.18 illustrate the experimental results based on the Rome graphs regarding random runs and the number of crossings. We can observe that the quality of the results can be largely improved by using multiple random runs: That is, using FLUP50 instead of LFUP1 can reduce the average number of crossings by more than 50% for graphs with $|V| \leq 40$ and more than 30% for graphs with $|V| = 100$. However, with increasing number of runs, the achieved improvement is getting smaller. We also can see that for graphs with $|V| \leq 35$, increasing the number of runs does not lead to a significant improvement while for graphs with $|V| > 35$ better results can still be achieved by increasing the number of runs. In context with the upcoming results regarding the instances with known upward crossing number, we can conclude that the optimum of many small instances is reached and no more improvement can be achieved. We also can observe that for the large graphs, significant improvement of the results of LFUP50 can only be achieved by vastly increasing the number of runs.

LFUP in the Context of k-LCM

Chimani, Hungerländer, Jünger and Mutzel [CHJM11] published an SDP-based approach for optimally solving k-LCM instances. The benchmark set used for the experimental evaluations includes the Rome graphs and the North



(a)



(b)

Figure 4.16: Average ratio of the number of crossings: Sugiyama50/LFUP50 and LFUP1/LFUP50 in dependency to the number of crossings achieved by LFUP50.

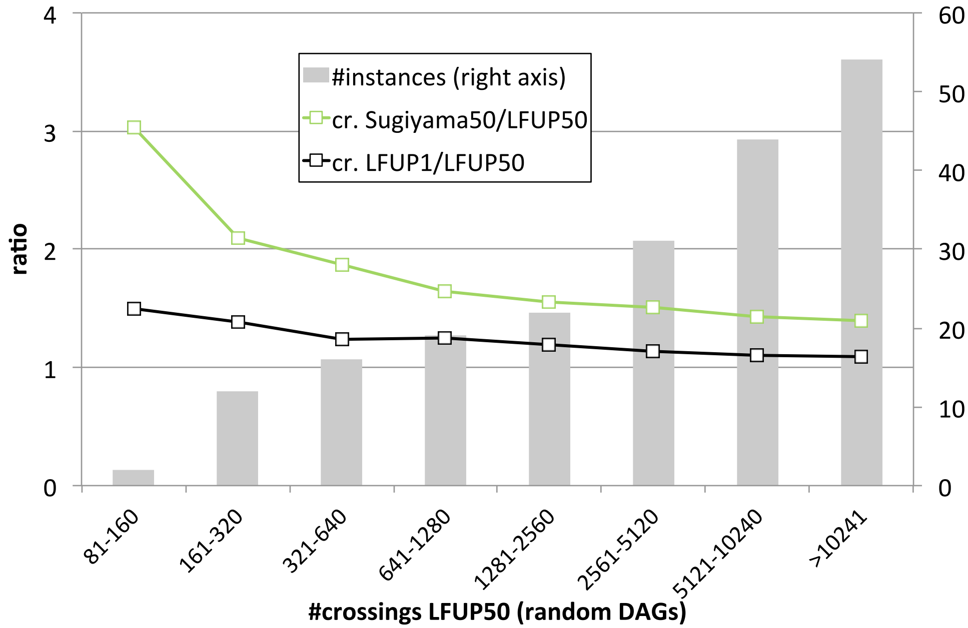


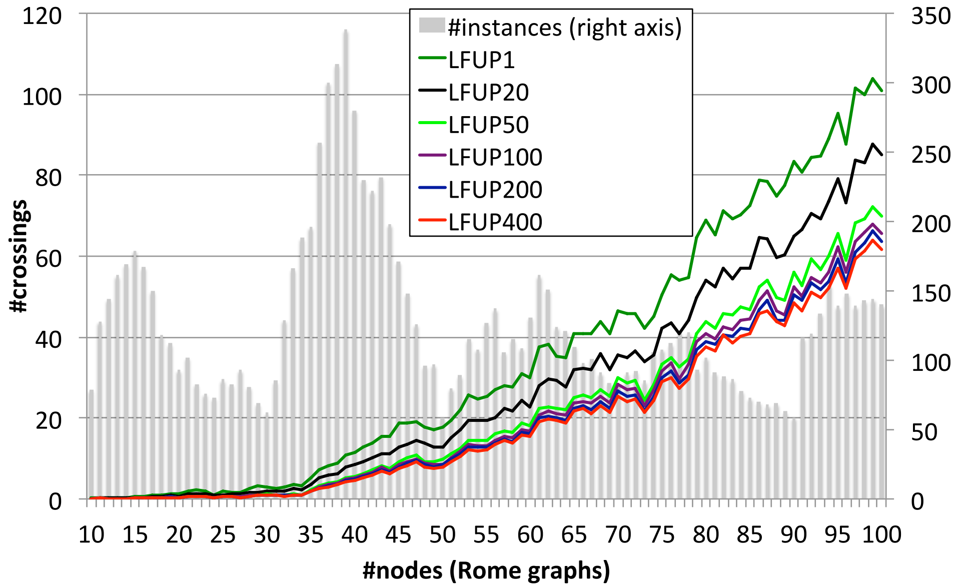
Figure 4.17: Average ratio of the number of crossings: `Sugiyama50/LFUP50` and `LFUP1/LFUP50` in dependency to the number of crossings achieved by `LFUP50`.

DAGs. The authors generated two groups of instances: The first group consists of instances (*GKNV-instances*) that are layered by `GKNV-Layering`, and the second group consists of instances (*UPL-instances*) that are derived from the layer free-upward planarization approach. That is, the Rome and the North instances are first upward planarized and then are layered with respect to the obtained UPRs by applying the new layering approach `UPL-Layering` introduced in the upcoming chapter. A node order for each layer of a UPL-instance $\mathcal{L}$ can also be derived from its UPR, hence we obtained a solution for $\mathcal{L}$ which can be considered as a heuristic solution produced by `LFUP`.

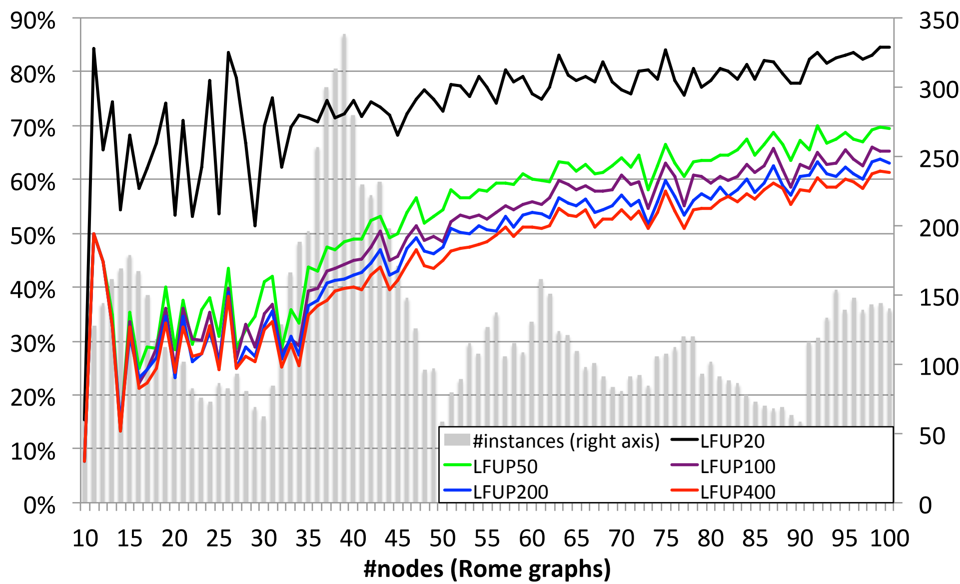
The runtime of the SDP approach depends on the size of the so-called *matrix dimension* ζ which corresponds to the number of variables of the SDP formulation. It is defined as $\zeta = 1 + \sum_{i=1}^{k} \binom{|L_i|}{2}$ where $|L_i|$ gives the number of regular and dummy nodes on the i-th layer. Since optimally solving large instances can take hours, a limitation is set such that only instances with $\zeta < 900$ (Rome graphs) and $\zeta < 1500$ (North DAGs) are considered, respectively.

Figure 4.19 shows some results published in [CHJM11]. "GKNV Barycenter" denotes the results obtained by applying `Barycenter5`[3] to the GKNV-instances and "GKNV optimum" the optimal solution of the GKNV-instances. Analogously, "UPL5" denotes the results of `LFUP5` and "UPL5 optimum" the

[3]Whose results are identical to `Sugiyama5`



(a) Number of crossings in dependency on the number of random runs.



(b) Number of random runs in relation to the results achieved by one run (LFUP1=100%).

Figure 4.18: The influence of the number of random runs to the number of crossings.

optimal solution of the UPL-instances derived from the UPRs obtained by applying `LFUP5`.

Evaluation. There are 7862 (Rome+North) GKNV-instances, all but two North DAGs were solved optimally by the SDP approach. The barycenter heuristic solved 24.44% (1922) of the considered instances optimally.

The UPL-instances consist of 3321 digraphs and all of them were solved optimally by the SDP approach. `LFUP` optimally solves *52.51%* thus 1744 instances. In considering the hardness of k-LCM, this is remarkable.

We observe a large gap between "GKNV optimum" and `LFUP5`; see Figure 4.19. When considering the plots of the Rome graph (Figure 4.12(a)) and the North DAGs (Figure 4.14(a)) in the previous evaluations and setting them in relation to the plots of Figure 4.19, we can conclude that `LFUP` outmatched the classical layered upward crossing minimization approach for many instances, even when the k-LCM instances is solved optimally.

We end this subsection by citing the analyses of the authors regarding the crossing minimization heuristics [CHJM11]:

> *Analyzing the distinct benchmark sets, we observe that the traditional leveling and crossing minimization heuristics leave plenty of room for improvement when considering the minimum number of crossings. In contrast to this, the graphs leveled by the UPL approach only allow much smaller improvements. In fact, it shows that the upward planarization approach [CGMW10a] gives near-optimal solutions for its respective leveling. We also observe that the fact that UPL produces more but smaller levels and requires less crossings is beneficial for both exact approaches[4]: they solve all UPL instances, while the GKNV instances are harder.*

DAGs with Known Upward Crossing Number

The best way to test the quality of our heuristic is to compare its results with the optimum. For this, we have applied `LFUP` with 100, 500, and 1000 random runs on the instances with known crossing numbers. The results are illustrated in Figure 4.20.

The x-axes gives the numbered instances from 1 to 20. The instances 1–4 correspond with the set of DAGs with $|V| = 20$, the instances 5–8 with the set with $|V| = 40$, etc. First of all, we can see that `LFUP` can find the optimal solutions for small instances, that is, $|V| \leq 40$. This fact coincides with the results of Figure 4.18 where the improvement for small instances not significant increases even when we start more random runs. For mid-size instances, that is, $40 < |V| \leq 60$, the computed solution is very close to the

[4]Besides the SDP approach, the authors had also considered an approach based on an ILP.

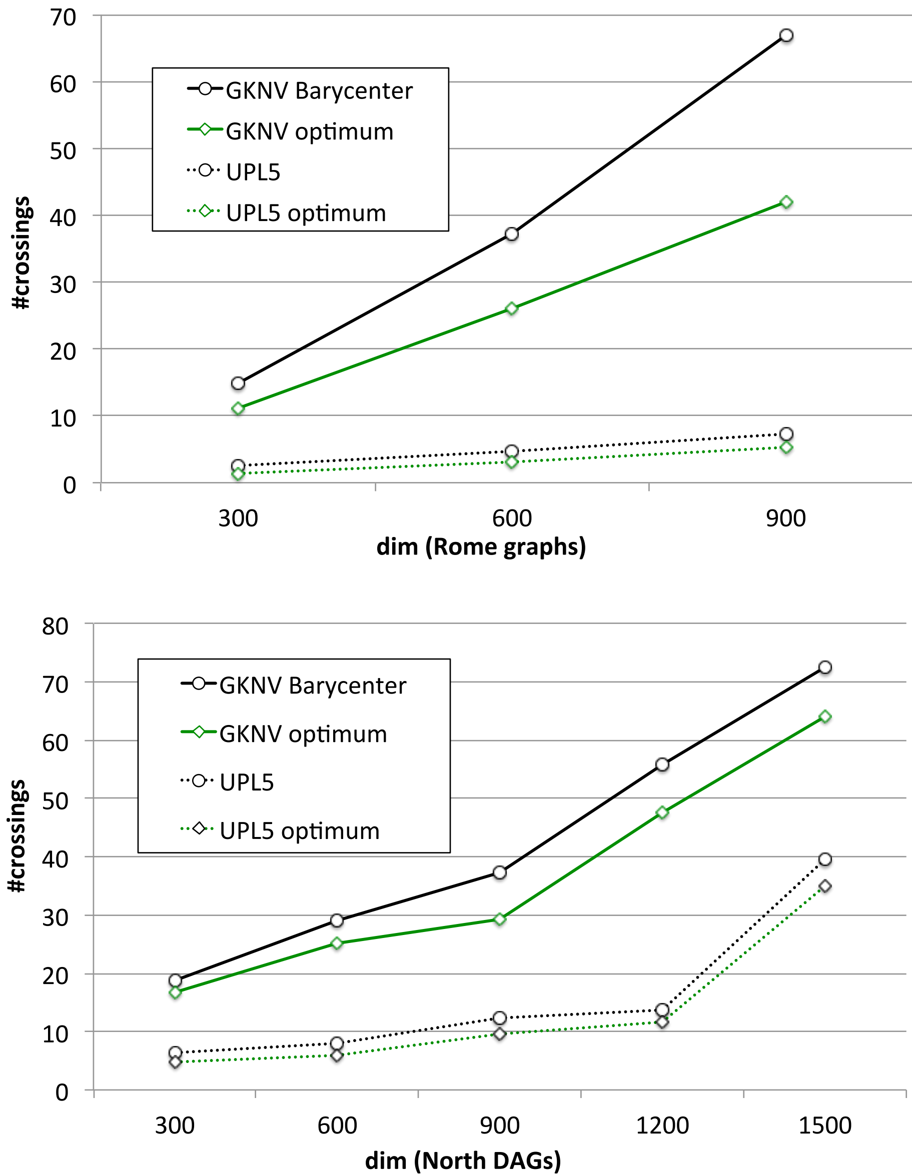


Figure 4.19: Optimal and heuristic solutions of the GKNV- and the UPL-instances. (For two GKNV-instances, no solution could be found by the SDP approach.) Observe that the underlying DAG of the GKNV- and UPL-instances are not necessary identical. (Data taken from[CHJM11].)

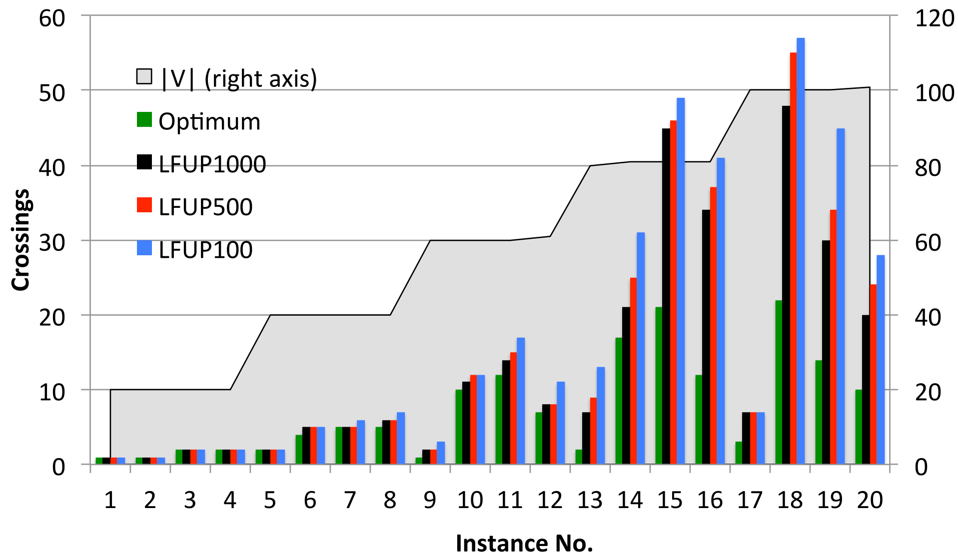


Figure 4.20: DAGs with known upward crossing number.

optimum, but for large instances, that is, $|V| > 60$, even with more random runs and even if the upward crossing number of the instances is small, `LFUP` is not able to find the optimal solution. The computed number of crossings are on average twice the optimum. So there is potential for improvement.

Upward Planar Graphs

There is another statistic of interest: While `LFUP`50 was able to find crossing-free UPRs for 25.5% and 62.3% of the Rome and North instances, respectively, `Sugiyama`50 was only able to find crossing-free layouts for 4.0% and 42.8% of the respective instances. In absolute numbers, this means that our approach found upward planar embeddings for 2249 and 250 (multi-source-multi-target) graphs, respectively, where `Sugiyama`50 required crossings. It also means that most of the North instances are upward planar. The random DAGs did not contain any graphs that could be drawn without crossings.

Constraint Feasibility

A very interesting outcome of our studies is that the feasible minimal path obtained by `FeasibleMinUIP` is already constraint feasible in most of the cases and therefore allows us to insert the arc provably optimally.

From the 2,708,474 arc insertion calls performed by `LFUP`20 in total over all Rome graphs, only 114 (0.004%) require to call the `ConstraintFeasibleUIP` heuristic; this corresponds to 0.87% of the instances requiring this heuristic at

all. Furthermore, the heuristic was *never* used for any North DAG or random DAG.

4.3.5 Runtime

The experiments were conducted on an Intel i5 2.67GHz with 8GB RAM per process under Windows 7[5]. The maximum computation time of `LFUP1` over all Rome and North instances was 0.312 seconds. The large Rome graphs ($|V| \geq 90$) take under 0.065 seconds on average, the large North DAGs ($|A| \geq 90$) require 0.078 seconds on average. For comparison, the runtimes of OGDF's `Sugiyama` implementation were at most 0.187 second.

Figure 4.21 shows the average computation time for the Rome graphs and the random DAGs. By construction, the computation time of `LFUP`i is nearly i times larger than `LFUP1`. Clearly, the running time increases for denser graphs, since we have to insert more arcs, and the respective insertion paths can be expected to be longer. Not surprisingly, we can also observe that the denser the graphs, the better becomes `Sugiyama`'s relative speed-up. Furthermore, `LFUP20` is a good compromise between effort (runtime) and quality, also `LFUP50` is not too slow.

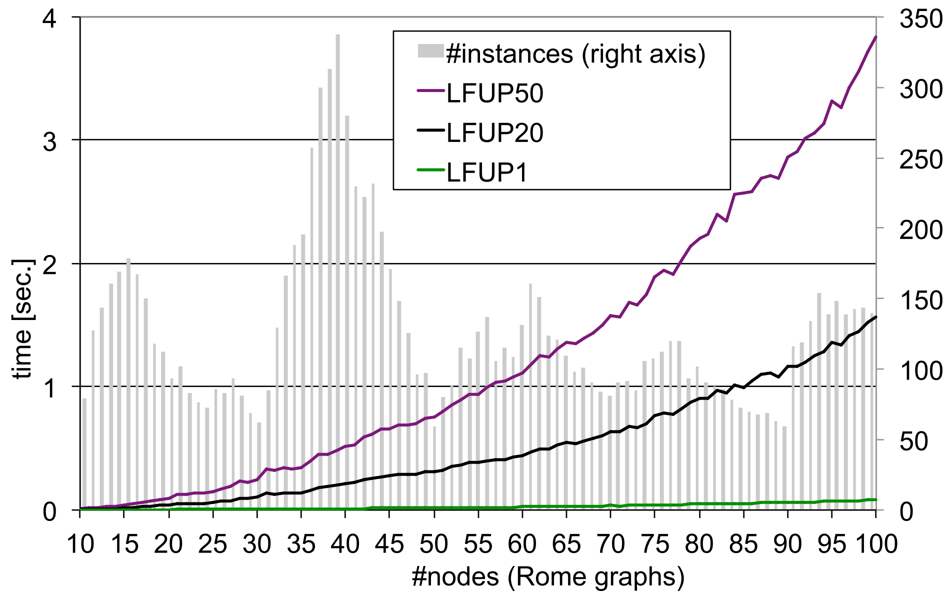
Figure 4.22 gives the runtime of the considered sifting algorithms. Notice that the sifting algorithms are implemented in Java within Gravisto [FBB$^+$05] and the data are obtained by running the algorithms on an Intel Xeon with 2.0 GHz, hence it cannot be used for a direct comparison with the runtime of `LFUP`.

4.3.6 Summary

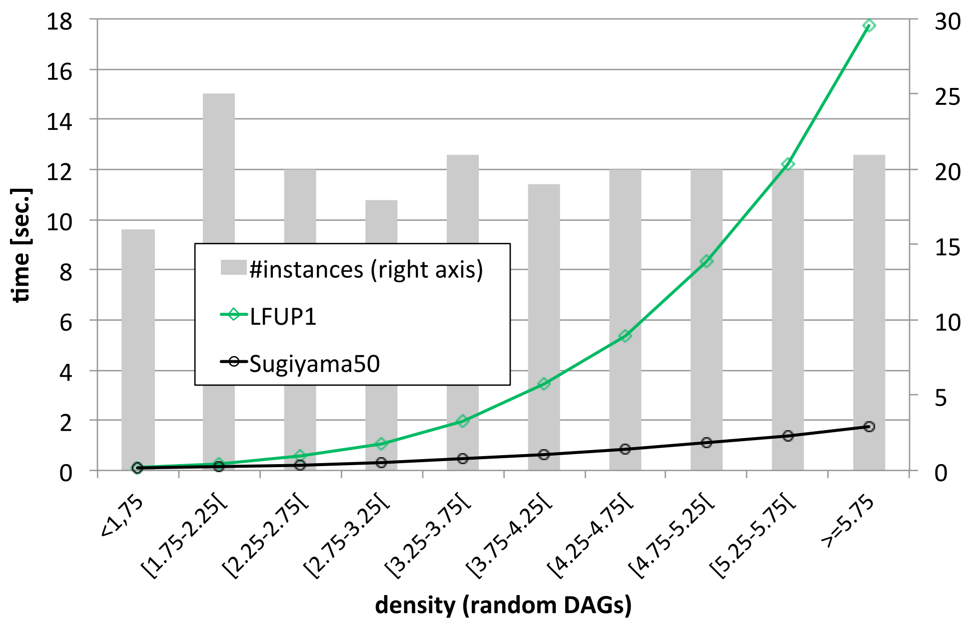
We have conducted the new upward planarization approach on several benchmark sets which include real-world-based and random instances. We summarize the results as follows:

- `LFUP` can be considered as the new state-of-the-art upward crossing minimization heuristic. It outperforms all the considered upward crossing minimization approaches.
- Multiple random runs of `LFUP` improve the results significantly.
- `LFUP` is able to find the optimum, that is, the upward crossing number, for small and sparse instances. For mid-size instances, the solution is near to the optimum.
- The maximal overall runtime of `LFUP1` (Rome graphs and North DAGs) was 0.312 seconds, hence the approach is practical.

[5] Hence the runtime data differ from the published results of [CGMW08, CGMW10a] due to the different hardware.



(a) Rome graphs: average runtime (in seconds) vs. number of nodes.



(b) Random DAGs: average runtime (in seconds) relative to the graph's density.

Figure 4.21: Runtime of LFUP.

bt

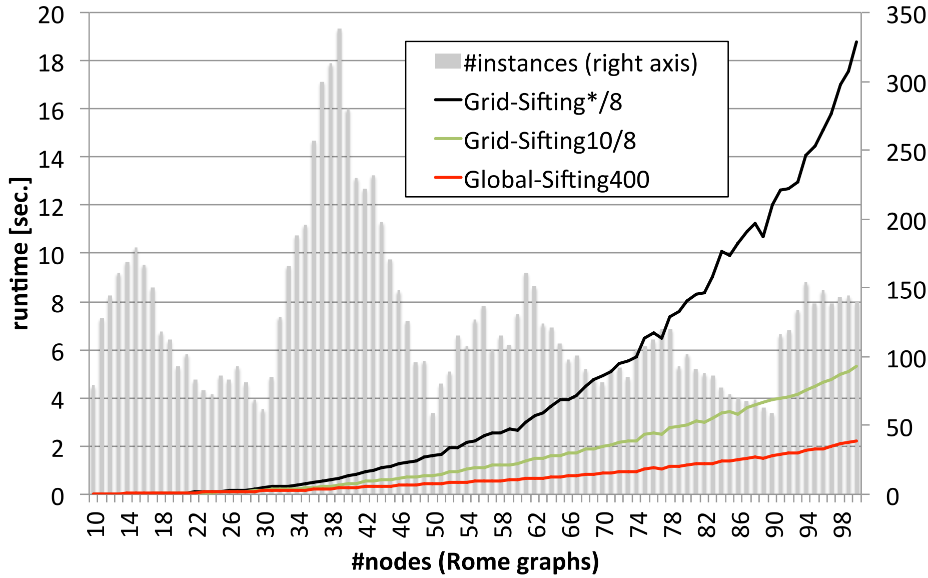


Figure 4.22: The runtime of the considered sifting algorithms (contribution of the authors of [BBG11]).

4.4 Extension to Port Constraints

Many practical applications of graph drawing come with domain specific constraints. In this subsection, we consider the given port positions that prescribe where a drawing of an arc can touch the images of its end nodes. Prescribed port positions arise in many technical applications where they are used for modeling the pins of electric components or the in- and output channels of data devices.

In this section, we introduce an extension of the upward planarization algorithm `LFUP`, allowing it to upward planarize digraphs with prescribed ports. We consider the fixed-port scenario, that is, a node v has no constraints or the given prescribed ports define a fixed circular arc order of the arcs incident to v. This scenario arises in many applications. Analogously to the upward planarization approach presented in the previous section, we focus on simple sT-graphs.

Let π be the circular arc order of a node v induced by the prescribed port positions. A circular arc order of v is *port constraint valid* (pc-valid) if it does not violate π. An upward planar embedding Γ is pc-valid if the circular arc order of each node induced by Γ is pc-valid. A UIP p is pc-valid if the embedding obtained after realizing p is pc-valid.

4.4.1 Chain Substitution

Given the prescribed positions of the ports of G, the port constraints $\mathcal{C}$ may require arcs $a = (u, v)$, with u being properly drawn below v,

(a) to leave u downwards, or

(b) to enter v from above, or

(c) to leave u downwards and enter v from above.

This clearly invalidates the pure upward drawing style (see Figure 4.23). Nevertheless, we want to allow such constructs in order to fulfill the given port constraints. Any such arc a requiring one of the three cases is substituted by a *chain* of *subarcs* (see Figure 4.23(c)):

(a) a is substituted by the chain $\langle(D, u), (D, v)\rangle$ and the subarc (D, u) is locked, that is, cannot be crossed.

(b) a is substituted by $\langle(u, D), (v, D)\rangle$ and the subarc (v, D) is locked.

(c) a is substituted by $\langle(D', u), (D', D), (v, D)\rangle$ and the subarcs (D', u) and (v, D) are locked.

The nodes D and D' are new introduced dummy nodes of the corresponding chain. Locking the accordant subarcs ensure that the corresponding line segments in the final drawing is directly neighbored to node u and v, respectively. For notational simplicity, we will continue to store the original unmodified arcs in the graph, denoted by the set *Chn*. Whenever we consider an arc $a \in \mathit{Chn}$, we in fact use the complete chain corresponding to a.

4.4.2 Feasible Subgraph

Besides ensuring the feasibility of the initial upward planar subgraph U, U also has to additionally be pc-valid with respect to the port constraints $\mathcal{C}$. Therefore, we extend the fixed-embedded FUPS test by adding an additional arc-wise test. This arc-wise test checks whether the arc orders of the nodes u and v fulfill the given port order when inserting the arc (u, v) in the current subgraph (also see Algorithm 1, line 7). In the end, we have a FUPS with a pc-valid upward planar embedding Γ. We refer to the adapted FUPS computation algorithm as `FUPS-PC`.

According to Lemma 12 a FUPS can be computed in runtime $\mathcal{O}(|A|^2)$. Since the arcs-wise test requires additional time $\mathcal{O}(|V|)$, we obtain the following corollary for the fixed port scenario:

Corollary 4. *Let $G = (V, A)$ be an sT-graph and $\mathcal{C}$ a set of given port constraints. Algorithm `FUPS-PC` computes a FUPS $U = (V, A')$ of G with a pc-valid feasible embedding Γ with respect to the arcs $A \setminus A'$ and the set $\mathcal{C}$ in time $\mathcal{O}(|A|^2)$.*

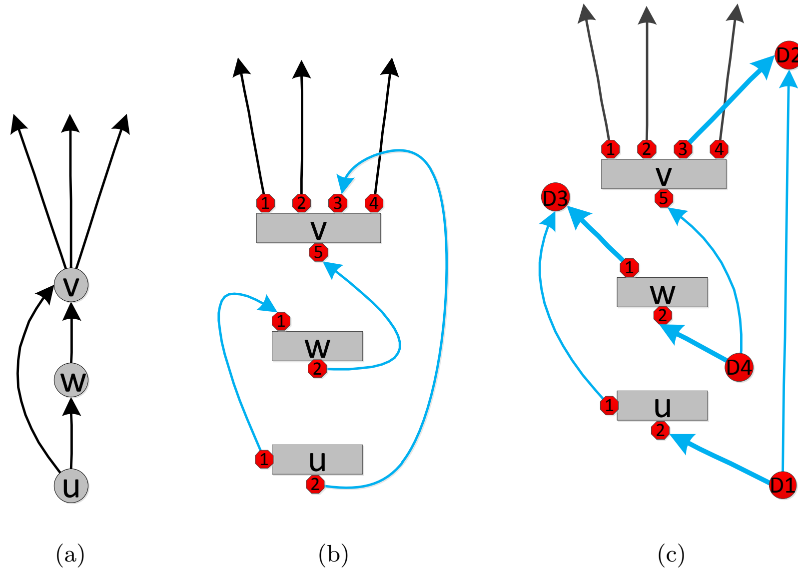


Figure 4.23: Port constraints: (a) No prescribed ports. (b) If $u \prec w \prec v$, then the prescribed port positions violate the upward property. (c) Chain substitutions: The bold subarcs of the chains are locked, that is, cannot be crossed in the upward planarization process.

4.4.3 Arc Reinsertion

Let $U = (V, A')$ be a FUPS of G with pc-valid feasible upward planar embedding Γ with respect to $\mathcal{C}$.

Routing network. As the routing network requires simple faces, we first augment each non-simple face of Γ to a simple face and thereafter substitute each arc $a \in \mathit{Chn}$ by its corresponding chain. The substitution may introduce new sinks and sources. We get rid of the new sinks by connecting them via sink-arcs to an appropriate top sink-switch of the faces incident to a. Analogously, we get rid of the new sources by connecting them via *source-arcs* to an appropriate source-switch. Again, the corresponding crossing-arcs of the sink- and source-arcs have lengths zero. As result, we have a new graph $\hat{U}$ with pc-valid upward planar embedding $\hat{\Gamma}$.

Merge graph. Assume $a \in \mathit{Chn}$ is substituted in $\hat{U}$ by the chain $g = \langle (D_1, u),\ (D_1, D_2), (v, D_2) \rangle$. When the merge graph $\mathcal{M}(\hat{\Gamma})$ is constructed, then the hierarchy between u and v may not mapped correctly to $\mathcal{M}(\hat{\Gamma})$, since due to the substitution, no path may exist in $\mathcal{M}(\hat{\Gamma})$ from u to v. Therefore, whenever we consider a merge graph of a FUPS or an intermediate UPR, we implicitly mean the merge graph that is constructed by ignoring the port

constraints. In particular, no arcs of *Chn* are substituted in $\mathcal{M}(\hat{\Gamma})$. Also recall that the statically locked arcs are computed based on an accordantly merge graph. So if $a \in \mathit{Chn}$ is locked, we in fact locked the whole chain corresponding to a in $\hat{U}$.

Insert an arc. Let $B = A \setminus A'$ and $\tilde{a} = (x, y) \in B$ be the current arc to be reinserted. Further, let R be the routing network for inserting (x, y) in $\hat{U}$ (with respect to $\hat{\Gamma}$) and let p be an insertion path for (x, y). We call the first face f_s and the last face f_t in the corresponding sequence of traversed faces of p the *start-* and *target-face* of p, respectively.

If the port constraints require that $\tilde{a}$ leaves x or enters y at some special positions, we can easily restrict the routing network R to use only applicable start- or target-faces for the routing (see Figure 4.24). Yet, there are additional augmentations necessary when $\tilde{a} \in \mathit{Chn}$. If $\tilde{a}$ has to leave x downwards or enter y upwards, we have to extend the routing network. Assume that $\tilde{a}$ only requires the second property (the first one is independent of the second and can be solved analogously). Usually, all arcs dominated by y will be statically locked, that is, we may not cross through them. Now, we unlock the arcs that directly leave y and search for a path p entering y from there; depending on the exact port constraints, only a single face above y may be a valid target-face. We realize p by inserting the corresponding chain of a into $\hat{\Gamma}$. Such a realization is referred to as *pc-valid realization* of p. After pc-valid realizing p, we maintain the simple face property by adding corresponding sink- and source-arcs to the intermediate UPR if necessary.

We can now naturally extend Definition 5 (UIP) to the fixed port scenario:

Definition 13 (UIP (fixed-port scenario))**.** A UIP p for (x, y) is an insertion path such that the embedding of the intermediate UPR obtained from realizing p is upward planar. The realization can be an ordinary realization according to Definition 1 or, if $(x, y) \in \mathit{Chn}$, a pc-valid realization.

This definition also extends Definition 7 ((constraint) feasible UIP). Now let R^* denote the adapted routing network.

Lemma 14. *Let p be an insertion path for (x, y) computed by performing `FeasibleMinUIP` on R^*. Then, p is a feasible minimal UIP with respect to the arcs of B and p is pc-valid.*

Proof. By the construction of R^*, p cannot violate the port constraints occurring at the nodes x and y. The minimality and feasibility of p follows from Corollary 2 together with the chain substitution. □

Similar to the port-free case, if after the realization of p the corresponding merge graph is acyclic, then p is also constraint feasible minimal and pc-valid.

We can use the insertion strategy introduced in Section 4.2.3 to insert all arcs of B. Following similar argumentation of the proof of the previous lemma, we can show that `ConstraintFeasibleUIP` also works fine on R^*.

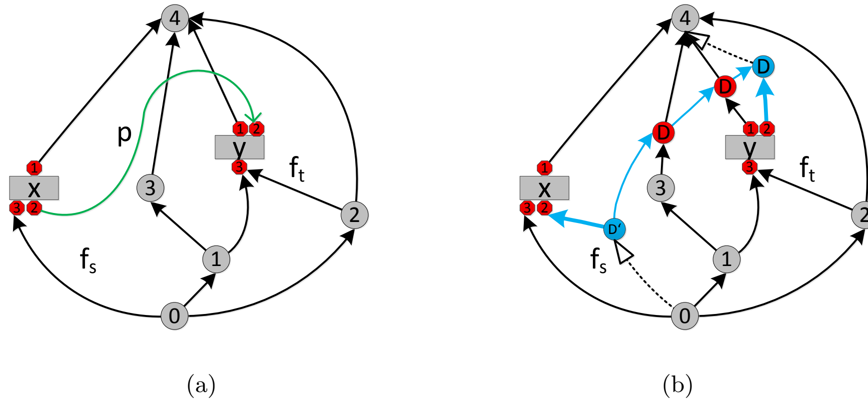


Figure 4.24: A simple example of a pc-valid realization: The nodes with given prescribed ports are drawn as rectangles, the ports are drawn in small red octagons. (a) An insertion path p for (x, y) with prescribed positions. Usually, the arc $(y, 4)$ is locked due to the static locking, but in order to fulfill the given port constraints, it is unlocked now for the insertion path computation of (x, y). (b) A pc-valid realization of p. Since $(x, y) \in Chn$, (x, y) is replaces by a chain (drawn in blue). The bold arcs of the chain are locked for the further path computation. The intermediate UPR is augmented to a single sink sT-graph by adding a source- and a sink-arc (dotted arcs). Thus, the new UPR is again an upward planar single sink sT-graph.

Lemma 15. *Let p be an insertion path for (x, y) computed by performing* `ConstraintFeasibleUIP` *on R^*. Then, p is a constraints feasible UIP with respect to the arcs of B and p is pc-valid.*

Runtime. The fixed-port scenario distinguishes from the port-free scenario in two aspects: the routing network construction and the chain substitutions. The network construction and the chain substitutions are bounded by $\mathcal{O}(|V|)$ (also see Lemma 2). Therefore, the runtime stated in Lemma 13 and Theorem 3 also holds for the fixed-port scenario.

4.5 Extension to Hypergraphs

Many technical applications such as data flow diagrams or electric schematics require the visualization of directed hypergraphs rather than digraphs. Layer-based methods that consider this task, for example, Spönemann, Fuhrmann, von Hanxleden, and Mutzel [SFvHM10], often suffer from too many arc crossings, and thus planarization-based methods—where minimizing the number of crossings is the main objective—may be preferable.

In this subsection, we describe a first approach for upward planarization of directed hypergraphs based on the layer-free upward planarization algo-

rithm. The ideas depicted in the following can also be combined with the port constraints extension.

Algorithm overview. Given a directed hypergraph $\mathcal{H}$, a suitable directed underlying graph H of $\mathcal{H}$ is first computed. Thereafter, H is transformed to an sT-graph G and a special FUPS U of G is constructed such that it does not contain any arc of F, where F is a minimal feedback arc set obtained by transforming H into G. Then, one hyperarc after another is inserted into U, that is, each hyperarc α is inserted by incrementally inserting the arcs of the star-based underlying graph, reusing the already established tree-based substructure of α as far as possible and using a special operation called *hypernode splitting* to abbreviate the insertion path. By specially considering original arc directions, we thereby guarantee that all hyperarcs can be drawn as confluent trees. The final result of the upward planarization is a UPR $\mathcal{R}$ of G, and hence of $\mathcal{H}$. Within $\mathcal{R}$, hyperarcs of $\mathcal{H}$ are represented as confluent trees, and crossings are represented as dummy nodes.

4.5.1 Pre-processing

In this very first step, we transform a directed hypergraph $\mathcal{H} = (V, \mathcal{A})$ into an ordinary digraph. We can assume that $\mathcal{H}$ contains no hyperarcs with self-loops, as they could be easily reinserted as a post-processing step without requiring any further crossings.

The choice of an appropriate underlying graph of $\mathcal{H}$ —star-based or tree-based— can influence the final number of crossings. Since we are interested in a UPR of $\mathcal{H}$ with as few crossing dummies as possible, but we do not know which underlying graph of $\mathcal{H}$ will lead us to this goal, we start with the star-based underlying graph of $\mathcal{H}$. We will see that a star-based underlying tree of a hyperarc can become any confluent tree-based underlying tree during the upward planarization process, hence does not narrow down the possible insertion path. So let now H be the star-based directed underlying graph of $\mathcal{H}$. We may assume that H is already transformed into a single source graph, but H may not be acyclic. It remains to make H acyclic by reversing the direction of the arcs in a minimal feedback arc set F. Unlike the case of ordinary digraphs, where the upward planarization process makes no difference between the reversed arcs of F and the not reversed arcs, we have here to take care of the set F due to the confluence property of the hyperarcs. So as the final result of the pre-processing, we have a simple sT-graph $G = (V, A)$ and a set F with $F \subset A$.

4.5.2 Feasible Subgraph and Arc Reinsertion

Based on the techniques for hypergraph planarization published by Chimani and Gutwenger [CG07], we explain in the following how to upward planarize

directed hypergraphs.

Feasible Subgraph

The FUPS computation differs slightly from Algorithm 1 or `FUPS-PC` (see Section 4.4.2) as we need here a FUPS that does not contain any arcs of F. Fulfilling this demand allows us to deal with the arcs of F only in the arc reinsertion phase. To achieve this goal we forbid the arcs of F for the FUPS computation. That is, we first compute a spanning tree $\mathcal{T} = (V, A')$ of G, with $A' \cap F = \emptyset$, which clearly exists due to the minimality of F. Then we insert the arcs $a \in A \setminus (A' \cup F)$ one by one into $\mathcal{T}$. As result, we have a FUPS U with an upward planar embedding Γ and an arc set $B = (A \setminus A')$.

Arc Reinsertion

Starting from the FUPS U, we will now iteratively insert full hyperarcs, until we obtain a UPR of G and thus of the input hypergraph $\mathcal{H}$. Note that hyperedge insertion in the tree-based paradigm is already NP-hard in the undirected, non-upward setting (Chimani and Gutwenger [CG07]). We therefore introduce a novel piecewise insertion strategy which realizes a low-crossing number hyperarc reinsertion ensuring the confluence of the hyperarcs.

For any original hyperarc $\alpha \in \mathcal{A}$ in $\mathcal{H}$, G contains a set of arcs $A_\alpha \subset A$ and a set of hypernodes V_α. Initially, $|V_\alpha| = 1$ since we started with the star-based underlying graph of $\mathcal{H}$. We will see that both sets will grow during the subsequent insertion steps. Also note that, in general, some arcs of A_α may be in A' and some in B.

Let $U' = (V, A')$ be an intermediate UPR obtained after reinserting some arcs during the insertion process, B' the not yet inserted arcs, and $\alpha \in \mathcal{A}$ the hyperarc to insert. The arc set $A'_\alpha = A_\alpha \cap A'$ forms a tree corresponding to the partially tree-based underlying tree of α, which is already in U'. In the initial FUPS U, this tree is at most a star (and at least a single edge) and is confluent. The arcs $B_\alpha = A_\alpha \cap B'$ with $B_\alpha \subset B'$ are the arcs corresponding to α that have yet to be inserted in the current iteration step.

Inserting α into U' with respect to its upward embedding Γ' means that we insert each arc $\tilde{a} = (x, y) \in B_\alpha$ one by one into U in a confluent way such that $\tilde{a}$ is connected to the partial underlying tree induced by A'_α. For now we assume that $\tilde{a} \notin F$. We first compute the minimal directed subtrees T_s, T_t of A'_α that contain all source and target nodes, respectively, that are already connected by A'_α (Figure 4.27). Let h_s and h_t be the sink and source of T_s and T_t, respectively.

Since the input graph G is a star-based underlying graph of $\mathcal{H}$, we have: If x is a hypernode, then y is a target node of α, and we search for a shortest feasible UIP from h_s to y. Otherwise, y is a hypernode, x is a source node of α, and we search for a shortest feasible UIP from x to h_t. For this, we modify the routing network in two ways:

Hypernode Splitting. The knowledge of the properties of hypernodes allow a further improvement, derived from [CG07] (Bokal, Fijavz and Mohar), where it was used in the context of undirected edge insertion for the so-called *minor-monotone* crossing number [BFM06]. We can cross "through" a hypernode h of another hyperedge, that is, we split h into two hypernodes h and h' and add the arc (h, h') which then will be crossed by the insertion path. The idea is illustrated in Figure 4.25: Instead of crossing two edges (in the fixed-embedding scenario), we can reduce the number of crossings by one when the insertion path for (x, y) is allowed to cross the hypernode h. However, since the hyperarcs must drawn in a confluent fashion, the idea of hypernode splitting cannot be adapted one by one without any modifications. For example, Figure 4.25(c) illustrates a directed case of the example in Figure 4.25(a), where splitting node h leads to a non-confluent underlying tree. We can observe: As long as we do not separate both, the source and the target node of the hyperarc, from each other, splitting of h still allows us to draw the hyperarc in a confluent fashion.

Let A_{in} and A_{out} be the arcs formerly entering and leaving h, respectively. To ensure confluence, we only allow a split where

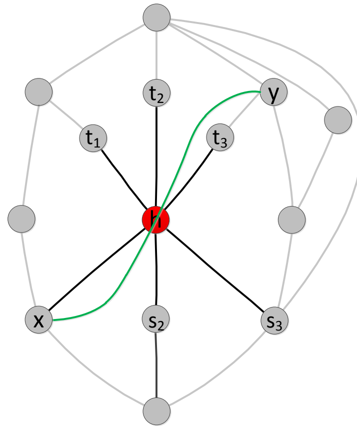
(a) all A_{in} remain incident to h;

(b) all A_{out} become incident to h'; or

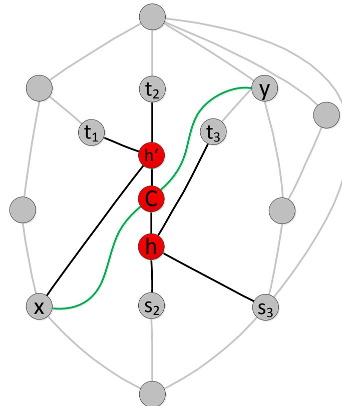
(c) neither of both, but either no arcs of A_{in} is incident to h' or no arcs of A_{out} is incident to h.

After this split, the routing path can cross over the arc (h, h'). Notice that since we consider a fixed-embedding, all valid hypernode-crossings can be modeled via arcs in the routing network directly connecting two faces. A schematic buildup of the modified network for faces incident to a hypernode is given in Figure 4.26. We observe that a star-based underlying tree becomes a tree-based underlying tree due to the split. Hence, the cardinalities of V_α and A_α increase.

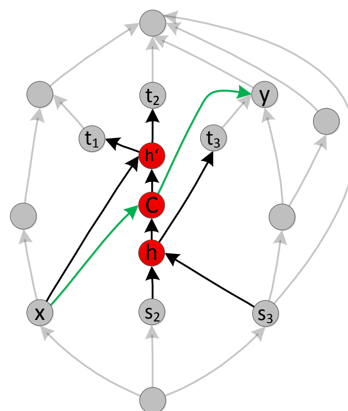
Arc Reuse. We want to reuse the already existing partial underlying tree induced by A'_α in U' in order to achieve a small total number of crossings. Therefore, the routing allows zero length of the crossing-arcs corresponding to the arcs of A'_α and to the arcs incident to crossing dummies in A'_α. In other words, the new path may reuse already established paths of α in U'. For example, Figure 4.27 illustrates the already existing partial tree induced by A'_α in U'. Inserting the arc $\tilde{a}$ into U' with respect to Γ' is equivalent to extend the tree A'_α in a confluent way by finding the shortest connection from node h_s to node y. The insertion path along the arcs of A'_α has length zero, and thus does not cause any crossings.



(a) The insertion path p for (x, y) crosses the hypernode h.



(b) A realization of p with one crossing.



(c) When considering the example as a directed graph as illustrated here, p would lead to an infeasible split since node x does not dominate node t_3.

Figure 4.25: Hypernode splitting: the undirected (a)-(b) and the directed (c) case.

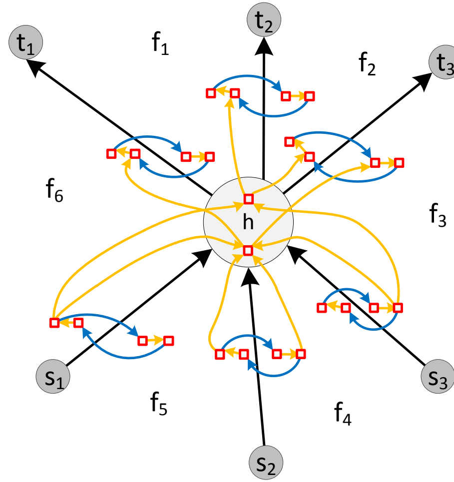


Figure 4.26: Schematic illustration of the partial routing network for a confluent hypernode splitting (also see Figure 4.7): Since a confluent split increases the number of crossing by one (see Figure 4.25), we set the length of all auxiliary routing arcs that "route in" the hypernode h to one. Notice: Some auxiliary routing arcs are omitted for the sake of readability.

Inserting reversed arcs. Let $\tilde{a} \in F$. In order to ensure that the hyperarcs can be drawn in a confluent fashion, we have to take special care of the arcs that were reversed in the pre-processing step. To avoid notational complexity, we will add such arcs only after all arcs of B_α (that is, not arcs of F) are already inserted.

Assume the arc $\tilde{a} = (x, y)$ originally connected a source node to the hypernode of the star-based underlying graph, but got reversed and hence connects the hypernode to a target node. Nonetheless we have to ensure that it connects to the aforementioned subtree T_s instead of T_t. Let S be the set of original sources in T_s before inserting $\tilde{a}$. A feasible UIP for $\tilde{a}$ can be obtained by setting the length of the crossing-arcs corresponding to the arcs of T_s and to the arcs incident to crossing dummies of T_s to zero and then computing the minimal insertion path from any node in S to y. The analogous holds, if $\tilde{a}$ originally connects a target node to the star-based hypernode.

Feasibility. We have sketched our ideas for upward planarizing directed hypergraphs. From now on we can assume that these ideas are incorporate into the algorithms `FeasibleMinUIP` and `ConstraintFeasibleUIP`. The remaining task is to prove the feasibility of hypernode splitting and arc reuse. Let R^* denote the modified routing network.

Lemma 16. *Let p be an insertion path for (x, y) computed by performing `FeasibleMinUIP` on R^*. Then p is a feasible minimal UIP with respect to*

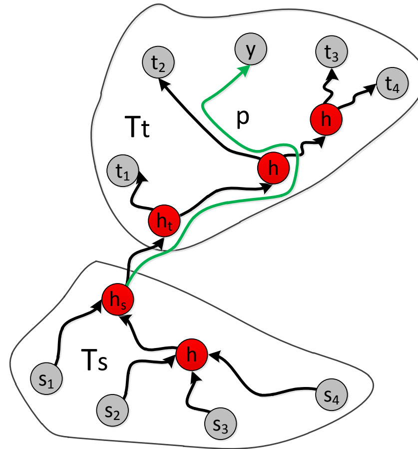


Figure 4.27: Arc reuse: The figure illustrates an already existing tree structure induced by A'_{α} in an intermediate UPR U'. The tree can be divided into two subtrees T_s and T_t. h_s is the sink of tree T_s an h_t the source of T_t. The insertion path p for $\tilde{a} = (x, y)$ starts at node h_s in order to ensure the confluence property of the extended tree structure.

the arcs of B, and the intermediate UPR with embedding Γ' —obtained after realizing p—has a confluent drawing which induces Γ'.

Proof.
Hypernode splitting: If a hypernode h is an end node of a statically or dynamically locked arc, then hypernode splitting must not be applied. The assignments of the length of the auxiliary routing arcs for hypernode splitting ensure that the length of a path p' in R^* corresponds to the number of crossings caused by its corresponding insertion path p. Hence Corrolary 2 also holds for `FeasibleMinUIP` where the technique of hypernode splitting is incorporate into. Thus p is feasible minimal. The underlying trees are confluent due to the construction of the routing network R^* for hypernode splitting.
Arc reuse: As we do not change the routing network but the length of its crossing arcs, we ensure that a path p computed by using the idea of arc reuse is feasible and the number of crossings caused by p again corresponds to the length of p' in R^*. Furthermore, as we insert (x, y) by using the subtrees T_s and T_t, the confluence of the extended underlying trees is ensured. Hence we can also conclude that p, computed by `FeasibleMinUIP` using the technique of arc reuse, is feasible. □

Again, if after the realization of p the corresponding merge graph is acyclic, then p is also a constraint feasible minimal UIP and if p is not constraint feasible, then we have to resort to the algorithm `ConstraintFeasibleUIP`.

Lemma 17. *Let p be an insertion path for (x, y) computed by performing `ConstraintFeasibleUIP` on R^*. Then p is constraint feasible with respect to the arcs of B, and the intermediate UPR with embedding Γ' —obtained after realizing p—has a confluent drawing which induces Γ'.*

Proof. Since `ConstraintFeasibleUIP` uses an intermediate merge graph to test whenever the currently considered path is constraint feasible or not, hypernode splitting is only applied if the path obtained thereafter is constraint feasible. Regarding arc reuse, an arc of the already existing tree structure is only added to the currently considered path if the intermediate merge graph is acyclic. Hence the computed UIP p is constraint feasible and does not violate the confluence property. □

4.6 Example

We illustrate LFUP step by step on a directed hypergraph with given prescribed port positions for some nodes.

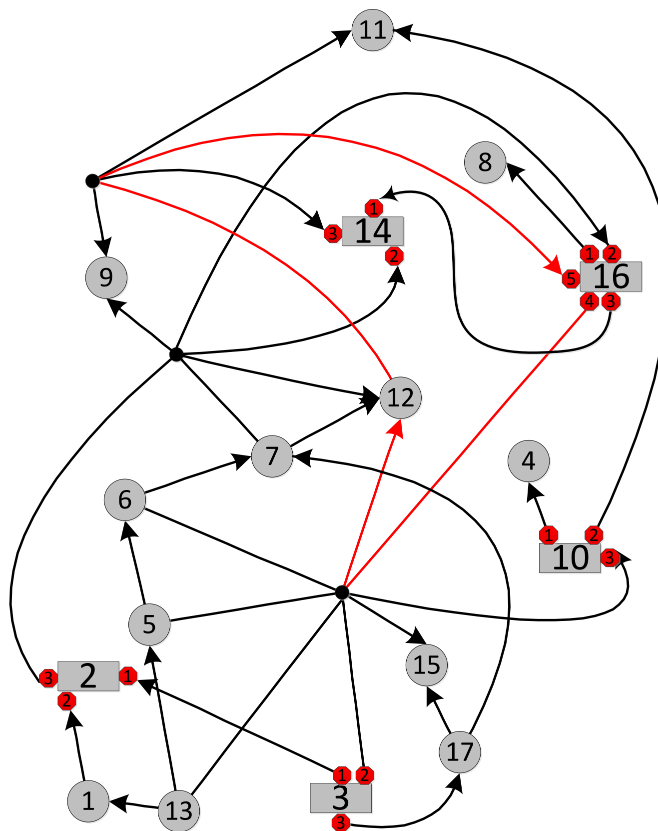
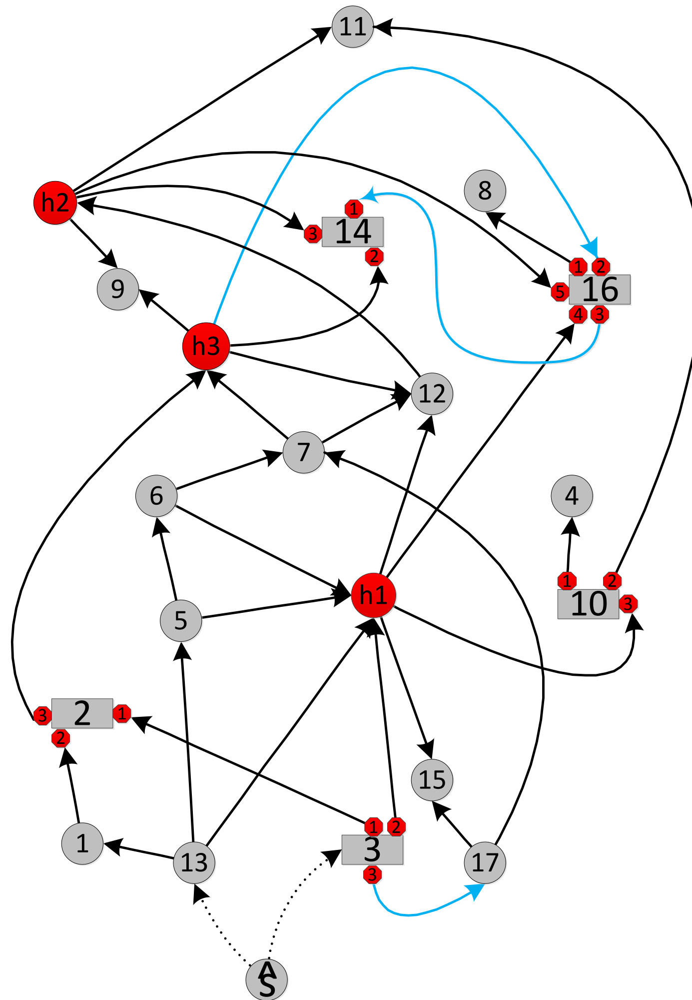


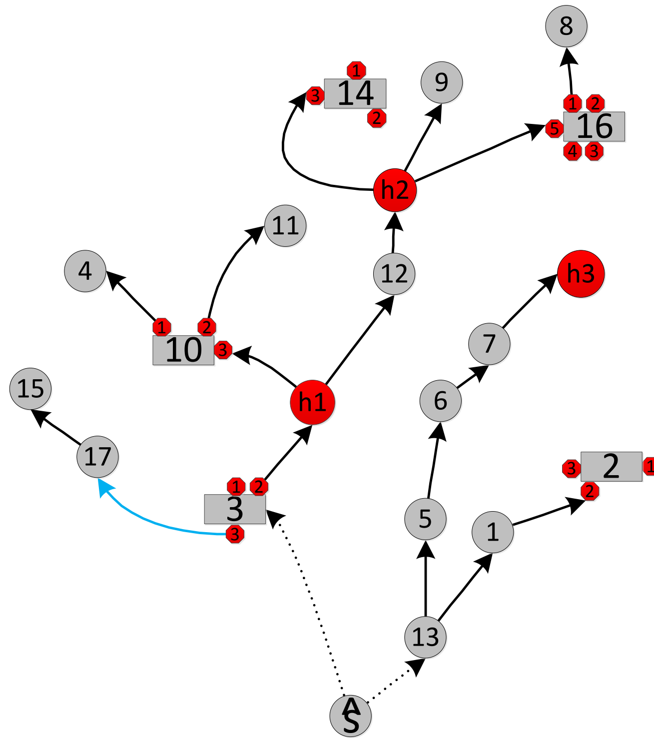
Figure 4.28: The input hypergraph $\mathcal{H}$ with given prescribed port positions on the node 2, 3, 10, 14, and 16. The ports are drawn as red octagons. The hypergraph contains a cycle (drawn in red).



(a) The augmented star-based underlying graph H of $\mathcal{H}$. The red nodes labeled with h_i are hypernodes and the blue arcs are arcs of *Chn*.

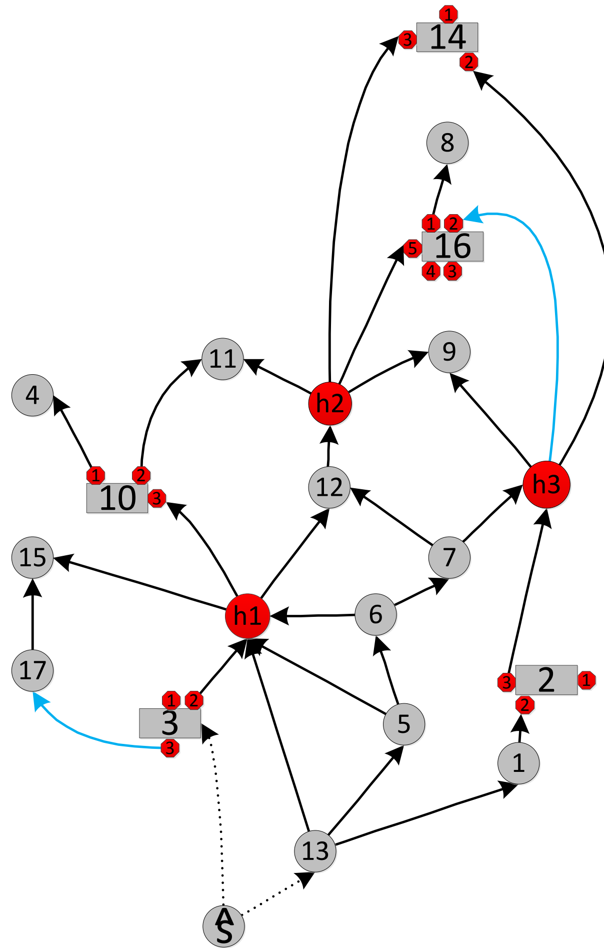
Figure 4.29: Pre-processing: The input hypergraph $\mathcal{H}$ is transformed into a directed star-based underlying graph H and then augmented to an sT-graph by adding the super source $\hat{s}$ and connecting it with the sources 13 and 3 of H. By reversing the direction of arc $(16, h_1)$, H becomes a DAG, thus we have the feedback arc set $F = \{(16, h_1)\}$. The following arcs violates the upward property and hence must later be substituted by chains: $Chn = \{(3, 17), (h_3, 16), (16, 14)\}$

.



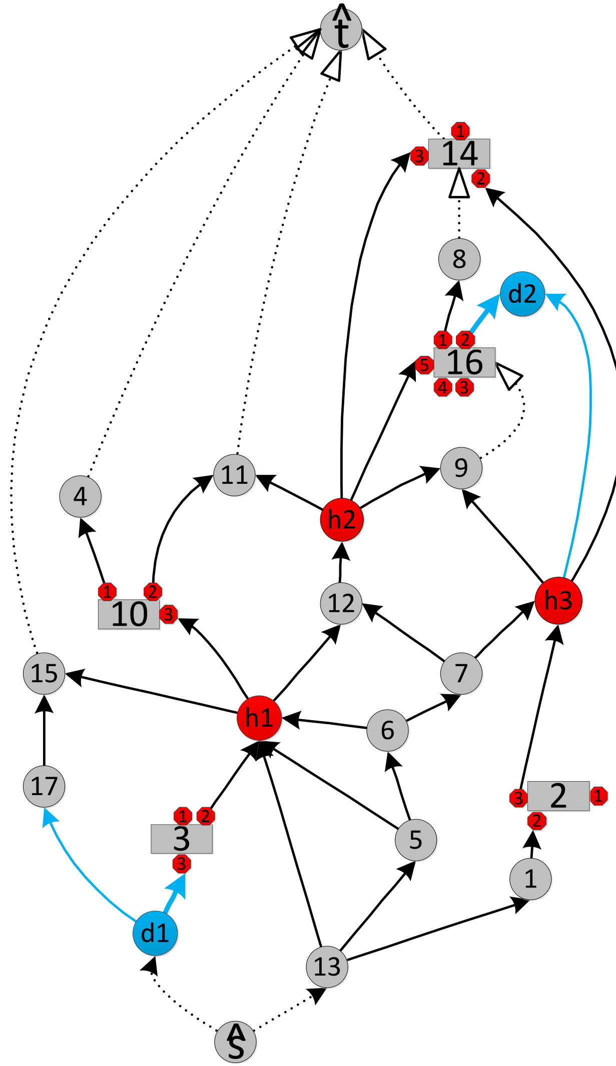
(a) The spanning tree $\mathcal{T}$ of the underlying graph H.

Figure 4.30: Feasible subgraph computation: In the first step, a spanning tree $\mathcal{T}$ of H is computed such that $\mathcal{T}$ does not contain any arcs of F.



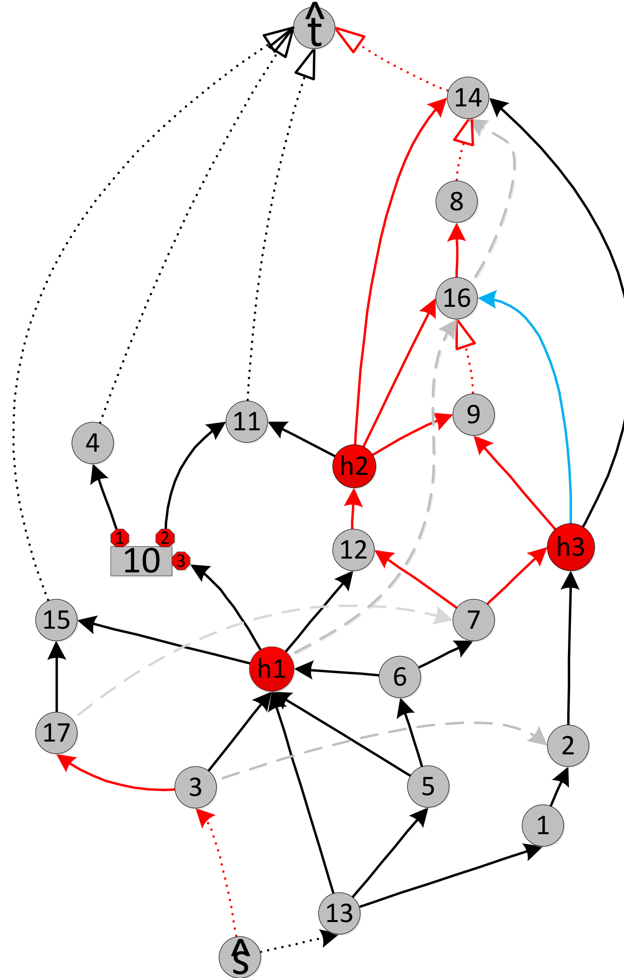
(a) The pc-valid embedded FUPS U of H.

Figure 4.31: Feasible subgraph computation: The following arcs of H are deleted: $\{e_1 = (17, 7), e_2 = (3, 2), e_3 = (16, 14), e_4 = (h_1, 16)\} = B$



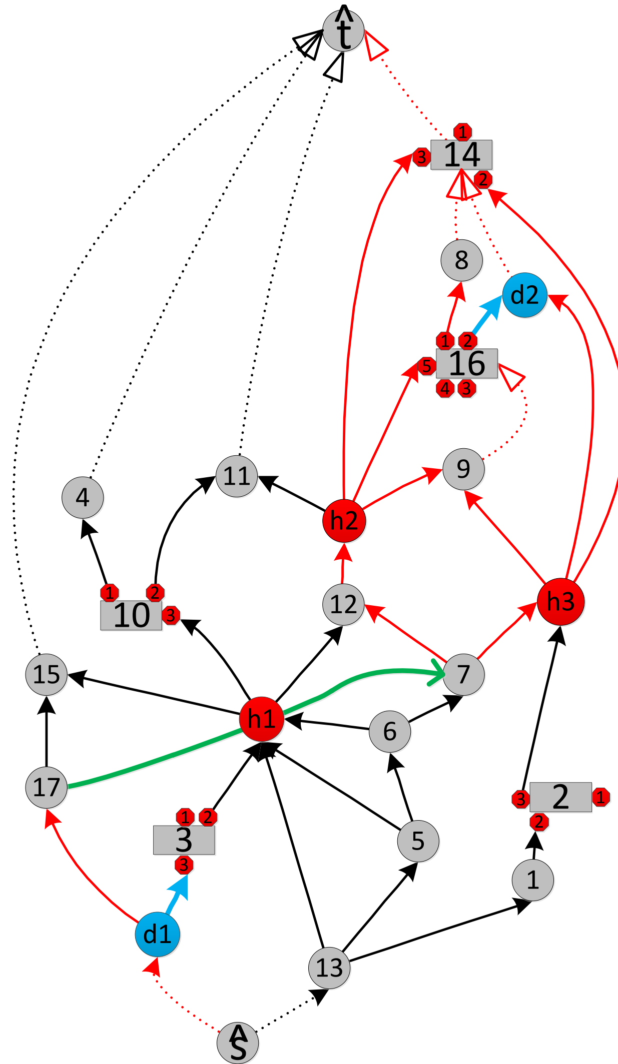
(a) The augmented embedded subgraph $\hat{U}$ of U.

Figure 4.32: For inserting the first arc e_1, U is augmented by adding appropriate sink-arcs (drawn as dotted arcs) to a single sink sT-graph. Then, the arcs of *Chn* are substituted by corresponding chains (drawn as blue arcs) and the resulting graph is transformed to a single sink sT-graph again. Recall that the bold drawn subarcs of the chains must not be crossed during the insertion path computation. We denote the upward planar embedding of $\hat{U}$ with $\hat{\Gamma}$. The faces of $\hat{\Gamma}$ are simple. This fact allows us to construct well structured sub-network for each single face.



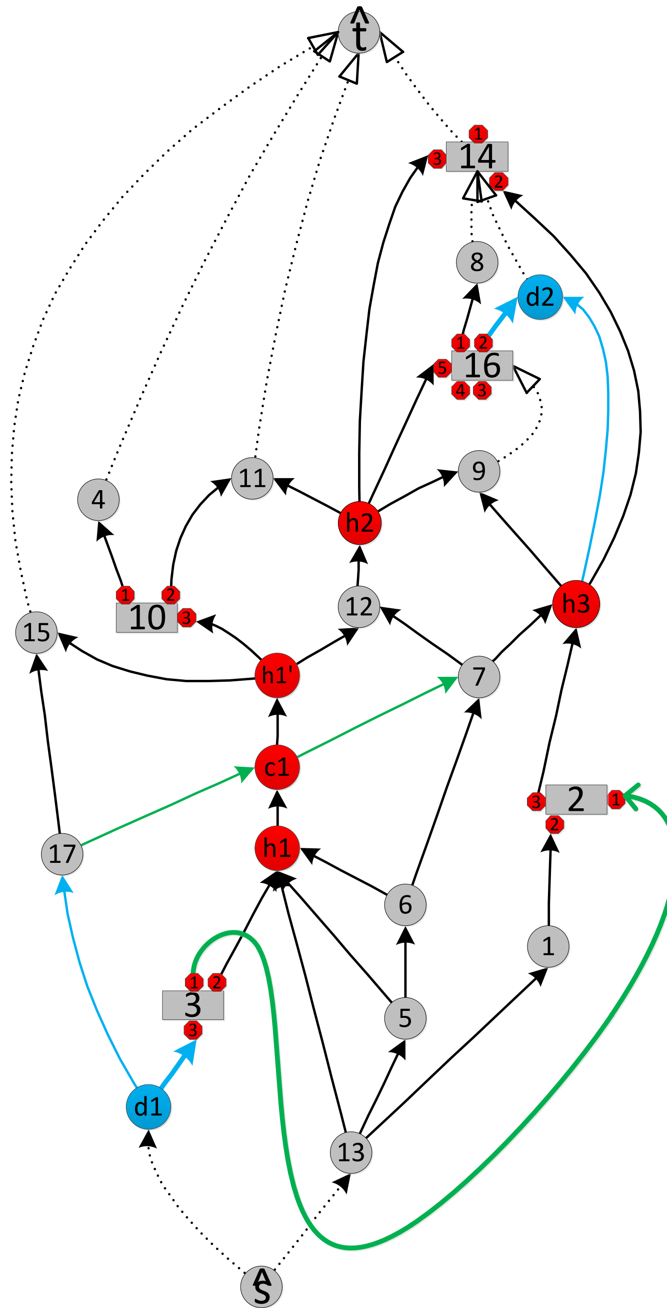
(a) The merge graph $\mathcal{M}(\hat{\Gamma})$ of $\hat{U}$. The grey dashed arcs are the deleted arcs, that is, arcs of B, and the red arcs are arcs of the static lock.

Figure 4.33: Computing the insertion path for the first arc $e_1 = (17, 7)$: In order to compute the static lock, we first construct the merge graph $\mathcal{M}(\hat{\Gamma})$ of $\hat{U}$ and then compute the dominating subgraph X of node 17 and the dominated subgraph Y of node 7. The arcs of both subgraphs are the arcs of the static lock and must not be crossed by the insertion path for $(17, 7)$. Recall that the merge graph is constructed regardless of the given port constraints and no chain substitutions are applied.



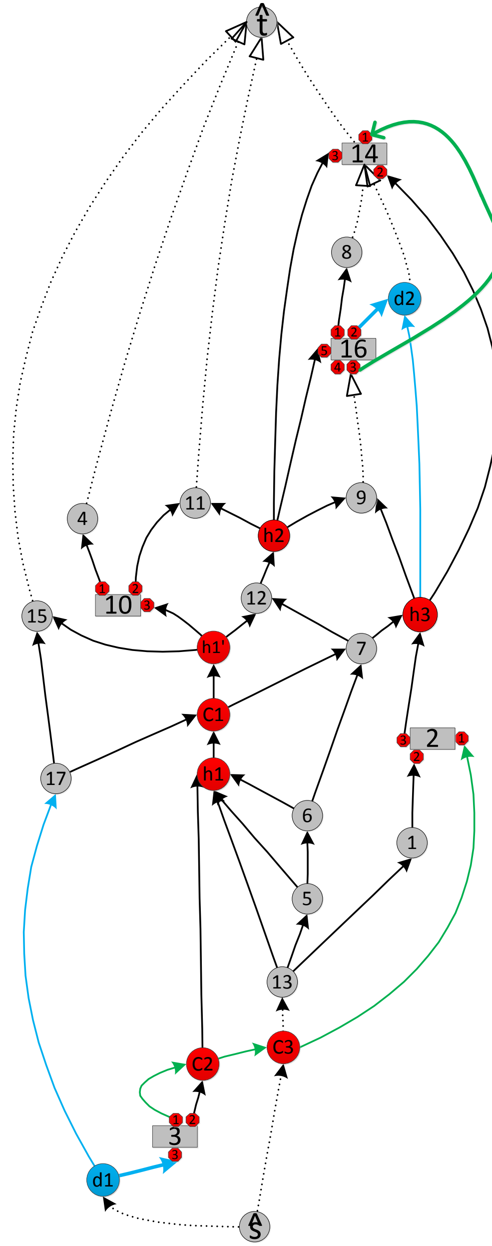
(a) The insertion path p_1 (drawn in green) for the first arc $e_1 = (17, 7)$. The statically locked arcs are drawn in red.

Figure 4.34: The technique of hypernode splitting is used for the insertion path computation of e_1: The path p_1 crosses the hypernode h_1.



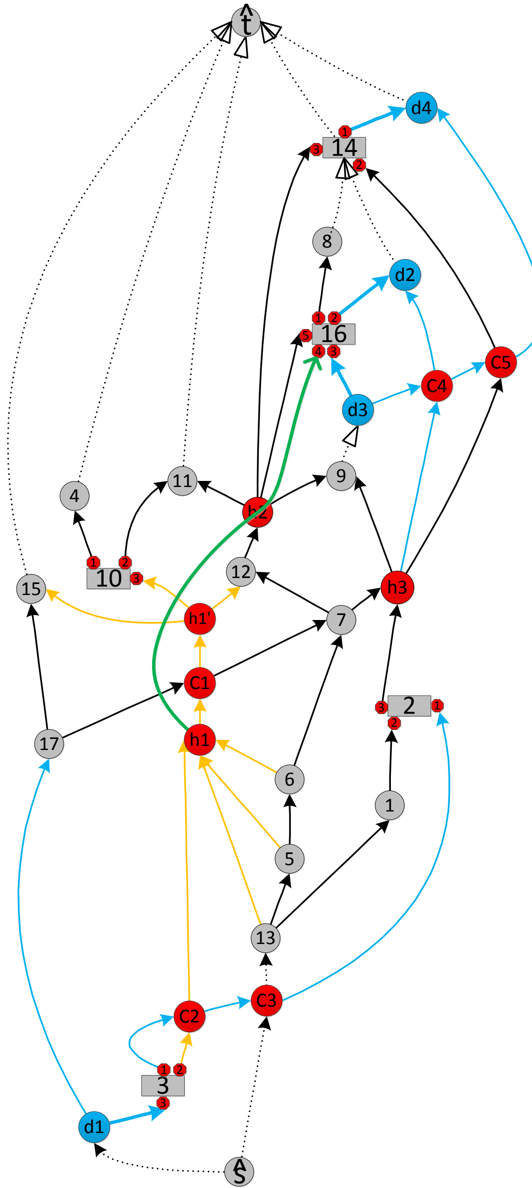
(a) The intermediate UPR U_1 after realizing p_1. The subarcs $(17, c_1)$ and $(c_1, 7)$ corresponds to p_1 an represent the arc $(17, 7)$ in the UPR.

Figure 4.35: The realization of p_1: The hypernode h_1 is split into two nodes, which allows the insertion for p_1 only causing one arc crossing. The corresponding crossing dummy is drawn in red and labeled with c_1. Also illustrated in the figure: the insertion path for the arc $e_2 = (3, 2)$ (bold green path). Observe that from now on, the static locks are not explicitly visualized (for the sake of readability).



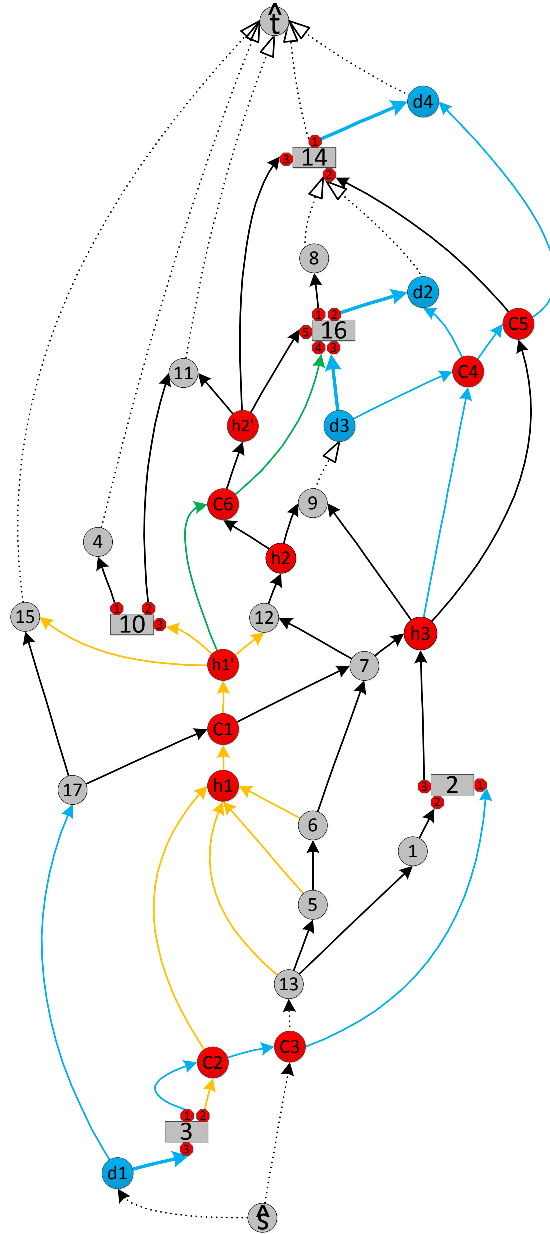
(a) The intermediate UPR U_2 obtained after realizing p_2. Arc $(3, 2)$ is represented by a chain of green subarcs. Recall that the bold blue arcs of the chains—$(d_1, 3)$, $(16, d_2)$—must not be crossed by any insertion paths.

Figure 4.36: The realization of p_2: The insertion of p_2 causes one arc crossing (crossing dummy c_2). The crossing occurring on arc $(\hat{s}, 13)$ will not appear in the final drawing because the arc has no counterpart in the original input graph $\mathcal{H}$. Also illustrated here: The insertion path p_3 (green bold path) for the next arc $(16, 14)$ to be inserted.



(a) The intermediate UPR U_3 after realizing p_3. Inserting the arc e_4 is equivalent to connecting node 16 to the existing tree (drawn in yellow).

Figure 4.37: pc-valid realization of p_3: Since $e_3 \in Chn$, we have to apply a chain substitution. Then add the source arc $(9, d_3)$ and the sink arc $(d_4, \hat{t})$ into the resulting intermediate UPR in order to make the faces simple. Insertion of p_4: The insertion path p_4 for the last arc $e_4 \in F$ requires the technique of arc reused and hypernode splitting. We have to compute an insertion path from h_1 to node 16, since e_4 is a reversed arc. Observe that the path along the yellow drawn arcs—the arcs of the already existing tree of the corresponding hyperarc—has zero length due to the construction of the routing network for inserting e_4.



(a) The final UPR U_4 after realizing the path p_4.

Figure 4.38: Due to the technique of hypernode splitting and arc reuse, realizing p_4 causes only one arc crossing which is modeled by dummy node c_6. We will continue this example in Chapter 5 Section 5.4, and illustrate how to turn this UPR into an upward drawing.

Chapter 5

Upward Planarization Layout

Organization of this chapter. In Section 5.1, we give an introduction to the problem of drawing DAGs based on UPRs, and in Section 5.2, we depict briefly how we can modify the Sugiyama framework for computing upward drawings based UPRs. Section 5.3 describes an approach for deriving a layering from a given UPR $\mathcal{R}$. In the same section, we also depict how the port constraints can be integrated into the obtained layering.

The second part of this chapter is dedicated to two layout approaches: the polyline hierarchical upward layout (Section 5.3.3) and the orthogonal layout approach (Section 5.3.5). In Section 5.3.4, we experimentally evaluate the first layout approach. We illustrate the new layout approach by an example in Section 5.4 and in the final section, we give some upward drawings produced by `Sugiyama` and `UPL`.

5.1 Introduction

In the previous chapter we have presented a novel algorithm for upward planarization of DAGs and directed hypergraphs. The final outcome of the algorithm is a UPR of the input graph.

In this chapter, we introduce a new drawing framework referred to as *upward planarization layout* (`UPL`) that combines the upward planarization approach `LFUP` with the novel ideas for computing layouts based on a given UPR.

In principle, each approach that can draw upward planar digraphs, for example, `Dominance`, `Visibility`, or even a modified `Sugiyama` can be used for constructing upward drawings based on UPRs. But the new approach `UPL` constitutes the first approach which takes the crossing dummies into account, hence is also the first algorithm specialized for layout digraphs and directed hypergraphs using UPRs. As our experiments show, it generates drawings that are preferable over the considered algorithms for upward drawings.

Figure 5.1 gives an overview of the considered algorithms for drawing

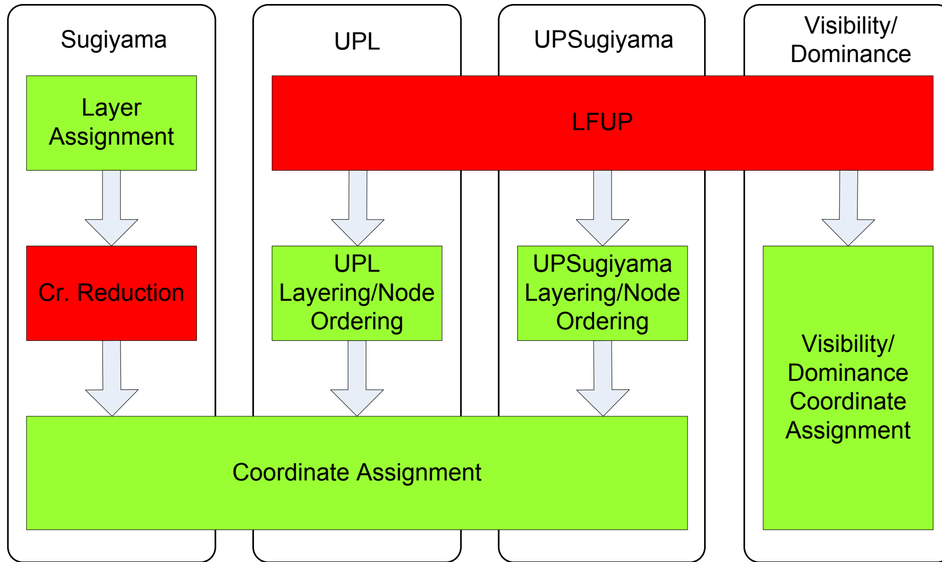


Figure 5.1: Overview of the frameworks for polyline hierarchical drawings: the classical drawing framework `Sugiyama`; the new upward planarization layout approach `UPL`; the straight-forward application of the Sugiyama framework on the UPRs `UPSugiyama`; the `Dominance` and `Visibility` approaches. For the latter three drawing algorithms, we use the layer-free upward crossing minimization approach `LFUP` to compute a UPR.

digraphs: the classical framework by Sugiyama et al., and the drawing algorithms based on upward planarization, which can be properly divided into two main steps: the upward crossing minimization and the layout step. The layout step of `UPL` and the straight-forward approach `UPSugiyama` based on the Sugiyama framework can further be divided into two sub-steps: the layering/node ordering step and final coordinate assignment. A first impression on the drawing characteristics of the considered drawing algorithms is given in Figure 5.2.

Let $\mathcal{R}$ be a UPR of a digraph or directed hypergraph G and let Γ be the upward planar embedding of $\mathcal{R}$. We assume that $\mathcal{R}$ contains auxiliary nodes and arcs. In particular, $\mathcal{R}$ is an upward planar embedded single source sT-graph.

Definition 14 (Realization of a UPR)**.** Let $\mathcal{D}'$ be an upward planar drawing of $\mathcal{R}$ inducing Γ. A *realization of* $\mathcal{R}$ is a drawing $\mathcal{D}$ obtained from $\mathcal{D}'$ by:

- replacing the images of the crossing dummies with crossings,
- replacing the hypernodes with branching points,
- replacing the drawings of the chains with drawings of their corresponding arcs, and by

- deleting the auxiliary elements of $\mathcal{R}$, that is, nodes and arcs without any corresponding counterpart in the original graph G.

Furthermore, all arcs are drawn according to their original direction.

So the main task here is to develop an algorithm for realizing UPRs.

5.2 Straight-forward Approach

The straight-forward approach—in the following referred to as `UPSugiyama`—is a simple method for realizing UPRs of digraphs that combines the upward planarization approach `LFUP` with existing layering and coordinate assignment algorithms. A layering $\mathcal{L}$ is straight-forwardly obtained by layering the UPR $\mathcal{R}$ using existing algorithms like `LongestPath` or `GKNV-Layering`. Thereafter, the node order on each layer is determined by using the "left path" method which is explained in the next subsection. Then a hierarchical drawing is constructed with respect to $\mathcal{L}$ by applying existing coordinate assignment algorithms, for example, `GKNV-Layout`. The drawing of the nodes are then replaced by arc crossings and artificial elements like sink-arcs, the super source, or super sink are deleted. The final drawing is a realization of $\mathcal{R}$.

Our experimental evaluations reveal that `UPSugiyama` produces quite unsatisfactory drawings with too many layers and much too long arcs. In addition, $\mathcal{R}$ often contains many crossing dummies which slow down the overall computation time. Nevertheless, we consider this approach in order to show the impact of the crossing dummies to the height of a layering.

5.3 Upward Planarization Layout

The upward planarization layout approach (`UPL`) addresses the weaknesses of the straight-forward approach `UPSugiyama`. It considerably improves the straight-forward approach by enhancing the computation of layers and node orderings and taking the special roles of the crossing dummies into account. This enhancement allows us to reduce the height of the drawings and lengths of the arcs substantially, resulting in much more pleasant drawings.

Upward planarization layout overview. In the layer assignment and node ordering step, we compute a *proper ordered layering* $\mathcal{L}$ of G which reflects the hierarchical order of the nodes of $\mathcal{R}$. Unlike the approach `UPSugiyama`, we do not compute a layering of $\mathcal{R}$, instead, we use an auxiliary graph H that allows us to ignore the crossing dummies, hence leads to a layering with fewer layers. Furthermore, we determine the node order on each layer of $\mathcal{L}$ by exploiting the information of Γ. However, due to the existence of some auxiliary arcs in $\mathcal{R}$, the layering $\mathcal{L}$ may contain unnecessary long arcs

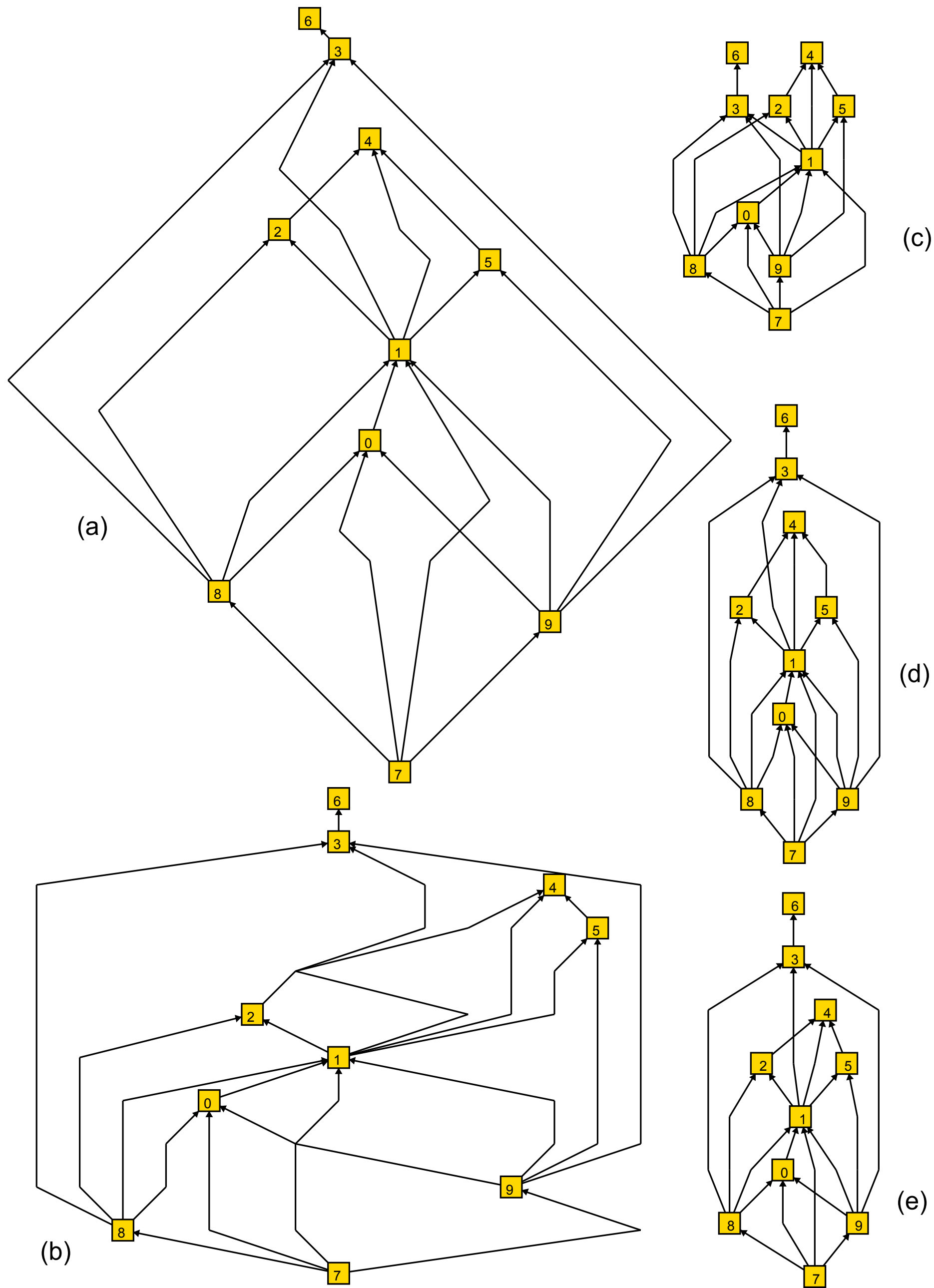


Figure 5.2: Characteristic layouts of instance *g.10.19* (North DAGs; see Section 4.3): (a) `Dominance`; (b) `Visibility`; (c) `Sugiyama`; (d) `UPSugiyama` (e) `UPL`. The drawing of (a),(b),(d), and (e) are based on the same upward planar representation computed by `LFUP`.

dummies. Therefore we introduce a post-processing step for long arc dummy reduction.

The coordinate assignment step is divided into two main parts. In the first part, we address the problem of computing appropriate coordinates for a polyline hierarchical layout. For this, we adopt existing techniques, for example, `GKNV-Layout` [GKNV93]. The obtained layout may have arc-node overlapping. We do not tackle this problem by enlarging the distance of the layers, instead we introduce an arc bending approach to handle this problem. The result is a more compact drawing.

In the second part we consider the problem of computing an orthogonal layout. Again, we first apply existing coordinate assignment algorithms to compute a layout and, by applying simple orthogonal arc routing, we obtain an initial orthogonal drawing. We refine this drawing by applying orthogonal compaction. For this, we extract orthogonal representations from the initial drawing. By assigning appropriate lengths to the line segments, we prevent the compaction algorithms from violating the goals achieved in the previous steps, that is, upward property and port validity.

5.3.1 Layer Assignment and Node Ordering

Let $\mathcal{L}$ be a proper layering of a digraph G. The layering $\mathcal{L}$ is an *ordered layering* if the node order on each layer of $\mathcal{L}$ is fixed.

Definition 15 (Realization of a Proper Ordered Layering $\mathcal{L}$)**.** Let $\mathcal{L}$ be a proper ordered layering. A *realization of* $\mathcal{L}$ is a layered drawing $\mathcal{D}$ with respect to $\mathcal{L}$ such that the order of the node images on each horizontal line Λ_i is identical to the corresponding node order on layer L_i of $\mathcal{L}$;

With this definition, the layer assignment task in the context of realizing UPRs can be decribed as follows:

> *Given a UPR $\mathcal{R}$ of G, find a layering $\mathcal{L}$ of G such that the realization of $\mathcal{L}$ is also a realization of $\mathcal{R}$.*

We first depict how to extract a layering from a given UPR of a DAG. Then we explain how to extend the introduced ideas to directed hypergraphs and how to incorporate the port constraints into a proper ordered layering.

Layer Assignment

Let $G = (V, A)$ be a DAG. Let H be a copy of G that we will use to obtain a valid layering for G. For any two nodes $u, v \in V$, we add an auxiliary arc (u, v) to H, that is, $H := H \cup (u, v)$ if:

(a) there exists no directed path from u to v in G and in H, but

(b) there exists a directed path from the corresponding nodes $u_{\mathcal{R}}$ to $v_{\mathcal{R}}$ in $\mathcal{R}$.

Part (a) prohibits the unnecessary generation of transitive arcs; part (b), in conjunction with the sink-arcs and the single-source, single-sink property of $\mathcal{R}$, ensures that the hierarchical order of $\mathcal{R}$ is mapped to H. Since G and $\mathcal{R}$ are DAGs, H is also acyclic and we can use any existing layering algorithm on H to obtain a layering $\mathcal{L} = \langle L_1, L_2, ..., L_k \rangle$ for H, and therefore also for G. We assume that $\mathcal{L}$ is made proper after applying some layering algorithm and refer to this layering approach as `UPL-Layering`.

Observe that even if G is only of moderate size, the upward planar representation $\mathcal{R}$ can become much larger due to the number of inserted dummies. This causes weak runtime performance, many layers, and overall unsatisfying drawings. By using the auxiliary graph H (which does not contain any dummies) for computing a layering of G, we get rid of these problems.

Node Ordering

Considering the proper layering $\mathcal{L}$, we now have to arrange the nodes on each layer according to the order induced by $\mathcal{R}$ such that the number of arc crossings occurring in the later realization of $\mathcal{L}$ is identical to the number of crossing dummy nodes of $\mathcal{R}$[1]. For this purpose, we consider the circular arc order of each node, as given by the upward planar embedding Γ of $\mathcal{R}$. In particular, we can recognize the *left incoming* arc for any node v, which is the embedding-wise left-most arc with target v. Note that this arc is defined for each node except for the super source $\hat{s}$.

Now consider any two distinct nodes u and v on the same layer L. We can decide their correct node order on L using the following strategy: we construct a *left path* p_u from $\hat{s}$ to u_R from back to front, that is, starting at $u_{\mathcal{R}}$, we select its left incoming arc a as the end of p_u and proceed from the source node of a, choosing its left incoming arc as the second to last arc in p_u, and so on. The construction of p_u ends when we reach the super source, which will always happen as $\mathcal{R}$ is a single source and single sink graph. Analogously, we construct the left path p_v from $\hat{s}$ to $v_{\mathcal{R}}$.

The paths p_u and p_v may share a common subpath starting at $\hat{s}$; let $c_{\mathcal{R}}$ be the last common node of p_u and p_v, and let a_u and a_v be the first different arcs, respectively. We determine the ordering of u and v directly by the order of a_u and a_v at c_R.

Algorithmically, we can consider each layer independently. Introducing an auxiliary digraph C, the above relationship between two nodes on the same layer can be modeled as an arc between these two nodes in C. We can construct a correct ordering for the layer by computing the topological order

[1] Observe: The number of crossing dummies refers to dummy nodes which occur in non auxiliary arcs.

in C. Note that therefore we do not have to compute the arc direction for all node pairs, but only for the ones that are not already “solved” by other arcs through transitivity. We obtain a final proper ordered layering such that the node order of each layer is determined by Γ and the hierarchy of the nodes is determined by $\mathcal{R}$. In the following, we assume that `UPL-layering` computes a proper ordered layering.

We observe that `UPL-Layering` utilizes exiting layering algorithms which can have influence to the height of the final layering. We have:

Lemma 18. *Let $\mathcal{R}$ be a proper ordered layering of a UPR $\mathcal{R}$ of G computed by `UPL-Layering`. Then a realization of $\mathcal{L}$ is also a realization of $\mathcal{R}$.*

For the class of sT-graphs, the `UPL-Layering` has the following nice property:

Lemma 19. *Let $\mathcal{R}$ be a UPR of an sT-graph G and $\mathcal{L}$ a proper ordered layering of $\mathcal{R}$ obtained by applying `UPL-Layering` based on `LongestPath`. Let h be the height of $\mathcal{L}$. Let $\mathcal{L}'$ be a layering of $\mathcal{R}$ with height h' such that any realization of $\mathcal{L}'$ is also a realization of $\mathcal{R}$. Then $h \leq h'$ holds.*

Proof. Let $p = \langle a_1, \ldots, a_k \rangle$ be a directed path from $u_\mathcal{R}$ to $v_\mathcal{R}$ in $\mathcal{R}$ such that there is no corresponding directed path from u to v in G and H (condition (a)). By the construction of H, the auxiliary arc $a' = (u, v)$ is added to H. We prove by induction over the number of arcs k of p that a' is necessary in order to map the hierarchy of $\mathcal{R}$ to H properly. The minimal height of $\mathcal{L}$ follows by the property of `LongestPath`; it computes layerings with minimal height.

Induction basic: Let $k = 1$. The path p consists of only one arc $a_1 = (u_\mathcal{R}, v_\mathcal{R})$. Since $u_\mathcal{R}$ and $v_\mathcal{R}$ are not crossing dummies, a_1 is not a subarc of any arcs. (Recall, a subarc arises by splitting an arc when introducing a dummy node.) Also a_1 does not correspond to an original arc of G due to condition (a), hence a_1 is a sink-arc. $u_\mathcal{R}$ is sink-switch and $v_\mathcal{R}$ is the top-sink-switch of an inner face f of the embedding Γ of $\mathcal{R}$. Due to $u_\mathcal{R} \rightsquigarrow v_\mathcal{R}$ in $\mathcal{R}$ in any realization of $\mathcal{R}$, $u \prec v$ must hold. Hence adding arc a' to H ensures this fact.

Induction hypothesis: Let $p' = \langle a_1, \ldots, a_{k-1} \rangle$. We assume that the hierarchy order given by p' is mapped to H and p' ends at a node which has a corresponding counterpart in the original graph G.

Induction conclusion: Let $p = \langle a_1, \ldots, a_k \rangle$ with $a_k = (x_\mathcal{R}, v_\mathcal{R})$. The arcs of p are not incident to the source and to the sink of $\mathcal{R}$ since both nodes have no corresponding original nodes in G. So we have three possible cases:

(i) a_k is a sink-arc: Since a_k is a sink-arc, in any realization of $\mathcal{R}$, $x \prec v$ must hold. By induction hypothesis, the hierarchy of the subpath $u_\mathcal{R} \rightsquigarrow x_\mathcal{R}$ is mapped to H, hence $u \prec x$ and by transitivity $u \prec v$. Adding arc a' to H ensures this fact.

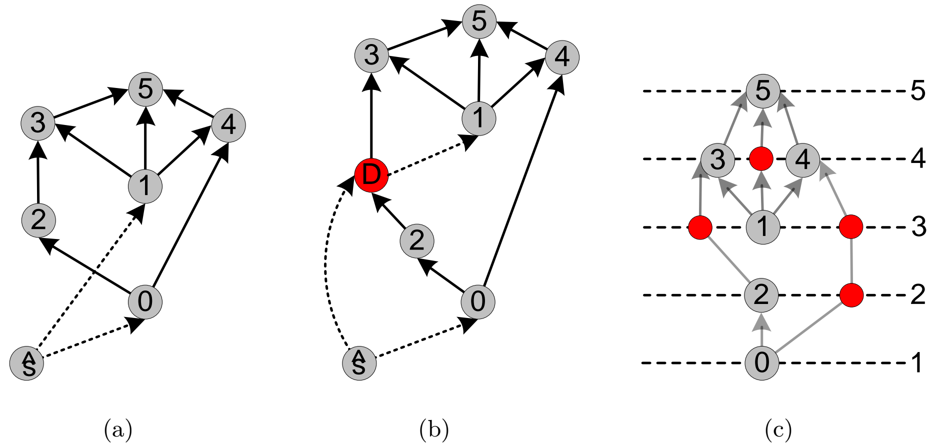


Figure 5.3: An example of a layering with unnecessary layers: (a) input DAG G which is augmented to an sT-graph; (b) a UPR $\mathcal{R}$ of G; (c) a proper ordered layering of G with respect to $\mathcal{R}$ obtained by performing `UPL-Layering` based on `LongestPath`. The number of layers can be reduced by one if we assign node 1 to layer two (as the right neighbor of node 2), node 3 and 4 to layer three, and node 5 to layer four.

(ii) a_k corresponds to an original arc $(x, v) \in G$: By the induction hypothesis, the order $u \prec x$ is already mapped to H, hence there is the path $u \rightsquigarrow x$ in H. Since H is a copy of G, the path can be extended to $u \rightsquigarrow v$ via arc (x, y). Due to condition (a) no auxiliary arcs are added to H.

(iii) a_k is a subarc: $x_\mathcal{R}$ is a crossing dummy. Let $w_\mathcal{R}$ be the node in p with the shortest distance to $x_\mathcal{R}$ (in p) with $w \in G$. By induction hypothesis, the hierarchy given by the subpath $u_\mathcal{R} \rightsquigarrow w_\mathcal{R}$ of p is mapped to H, hence $x \prec w$. The subpath path $w_\mathcal{R} \rightsquigarrow x_\mathcal{R}$ consists of the sequence $\langle (w_\mathcal{R}, C_1), \ldots, (C_{l-1}, C_l), (x_\mathcal{R}, v_\mathcal{R}) = a_k \rangle$, where C_i are crossing dummies. Notice that the case $C_1 \equiv x_\mathcal{R}$ may occur. In order to preserve the upward property, each crossing ξ_i that is modeled by the dummy C_i must be drawn such $\xi_1 \prec, \ldots, \prec \xi_l \prec \xi_{x_\mathcal{R}}$, hence by transitivity we have $u \prec \xi_{x_\mathcal{R}}$. Due to the arc $a_k = (x_\mathcal{R}, v_\mathcal{R})$ in $\mathcal{R}$ it is $\xi_{x_\mathcal{R}} \prec v$ and by transitivity $u \prec v$. Hence adding a' to H is necessary.

□

Unfortunately, `UPL-Layering` does not compute a layering with minimum height for general DAGs, for example, see Figure 5.3. Due to the artificial augmentation of the input graph to an sT-graph (a), there exists a path from node 2 to node 1 in the UPR (b). Therefore, node 1 is layered higher than actually necessary.

Extension to Directed Hypergraphs

We extend `UPL-Layering` to UPRs of directed hypergraphs. Let $\mathcal{R}$ be a UPR of a directed hypergraph $\mathcal{H}$. The layering algorithm consists of three stages: In the first stage, a *coarse layering* $\mathcal{L}' = \langle L'_1, \ldots, L'_k \rangle$ of the nodes of $\mathcal{H}$ is computed by performing `UPL-Layering` on $\mathcal{R}$. Thus, the nodes of the subgraph between two consecutive layers of $\mathcal{L}'$ are either crossing dummies or hypernodes. In the second stage, we compute a *fine layering* $\mathcal{L}''_i$ for the hypernodes and the crossing dummies between each two consecutive layers L'_i and L'_{i+1} of $\mathcal{L}'$, again using `UPL-Layering` (If the final drawing is not drawn in orthogonal style, then it is sufficient to layer the hypernodes only.). In the third stage, we merge the layering $\mathcal{L}'$ of the real nodes and the layering $\mathcal{L}''_i$ of the hypernodes and crossing dummies to a whole layering $\mathcal{L} = \langle L_1, \mathcal{L}''_1, \ldots, \mathcal{L}''_{k-1}, L_k \rangle$. Since the nodes on a layer of $\mathcal{L}'$ are regular nodes, there may be hypernodes and crossing dummies which are not uniquely located in the subgraph of two consecutive layers L'_i and L'_{i+1}, for example, the adjacent regular nodes of a dummy are assigned to layers L'_i and L'_j with $|j - i| > 1$. These nodes are now assigned to the layers of $\mathcal{L}$ which were formally layers of the fine layering. Thereafter, $\mathcal{L}$ is made proper by splitting long arcs and the node order of each layer is determined.

Definition 16 (Realization of a Layering $\mathcal{L}$)**.** Let $\mathcal{L}$ be a proper ordered layering of a directed hypergraph obtained as described above. Let $\mathcal{D}'$ be a realization of $\mathcal{L}$ where the hypernodes and the crossing dummies are considered as real nodes. A realization of $\mathcal{L}$ is a drawing obtained from $\mathcal{D}'$ by replacing the hypernodes with branching points and the images of the crossing dummies with arc crossings.

The algorithm `UPL-Layering` computes a layering $\mathcal{L}$ such that the node hierarchy induced by $\mathcal{R}$ is mapped to $\mathcal{L}$ and the node order (real or hypernodes) on each layer is induced by the embedding Γ of $\mathcal{R}$, hence we obtain the next Corollary which also summarizes the finding of this subsection:

Corollary 5. *Let $\mathcal{R}$ be a UPR of a digraph or a directed hypergraph G and let $\mathcal{L}$ be a proper ordered layering computed by `UPL-Layering`. Then a realization of $\mathcal{L}$ is also a realization of $\mathcal{R}$.*

Observe that the dummy nodes of the chains (if any exist) are not assigned to any layers yet. We now describe how to deal with them.

Chains and port constraints. If port constraints occur, then we have to take care of the prescribed port positions and the chains which represent some original arcs. Assume we have two end nodes u and v of a chain corresponding to the arc $a = (u, v)$. Let $L(u)$ denote the assigned layer of the node u in $\mathcal{L}$. The dummy nodes of the chains are assigned to layers which contain no regular nodes. If such a layer does not exist—this is in particular the case

when $\mathcal{R}$ is a UPR of a DAG—then we introduce a new layer directly above or underneath the node u (see Figure 5.4 and also Figure 4.23 in Section 4.4). In the final drawing these dummies are bend-points of the arc a. The dummies of a chain a are assigned as follows:

(a) a leaves u downwards: The dummy node is assigned to layer $L(u) - 1$.

(b) a enters v from above: The dummy node is assigned to layer $L(v) + 1$.

(c) a leaves u downwards and enters v from above: We have the chain $C = \langle (D_1, u), (D_1, D_2), (v, D_2) \rangle$. D_1 is assigned to layer $L(u) - 1$ and D_2 is assigned to layer $L(v) + 1$.

Due to these assignment rules, the corresponding bend-point is drawn not too far from the nodes u and v, respectively. Also recall that we does not allow some subarcs of the chain to be crossed (see Section 4.4). If there are more than one dummies on a layer L_i, then the node order of L_i is again determined by the embedding Γ of $\mathcal{R}$.

5.3.2 Post-processing

We introduce two post-processing algorithms to improve the quality of a layering $\mathcal{L} = \langle L_1, \ldots, L_k \rangle$. Although both algorithms have been developed for drawing digraphs [CGMW11, CGMW10b], they can also be used for layerings of directed hypergraphs.

Long-Arc Dummy Reduction

Most layering algorithms—in particular also the optimal LP-based approach `GKNV-Layering`—will put the nodes on the lowest possible layer. While this is in general a good idea, this approach can be counter-productive in the context of the super source node that will be removed from the final drawing: Since every source node s in G is attached to the super source node $\hat{s}$ (which is on the lowest layer), s may end up very low in the drawing, even though most of its dominated subgraph requires higher layers, hence introducing long arcs.

We tackle this problem using an approach similar to the *promotion layering approach* by Nikolov and Tarassov [NT06] by re-layering parts of the dominated subgraphs after the removal of $\hat{s}$, without modifying the hierarchical order induced by $\mathcal{R}$ (see Algorithm 7). Layers that become empty by these operations can be removed afterwards.

Repositioning the Sources

Since the upward planarization algorithm considers G as a regular sT-graph although it has been augmented with an artificial super source $\hat{s}$ and additional auxiliary arcs, the final upward drawing may contain artifacts in the

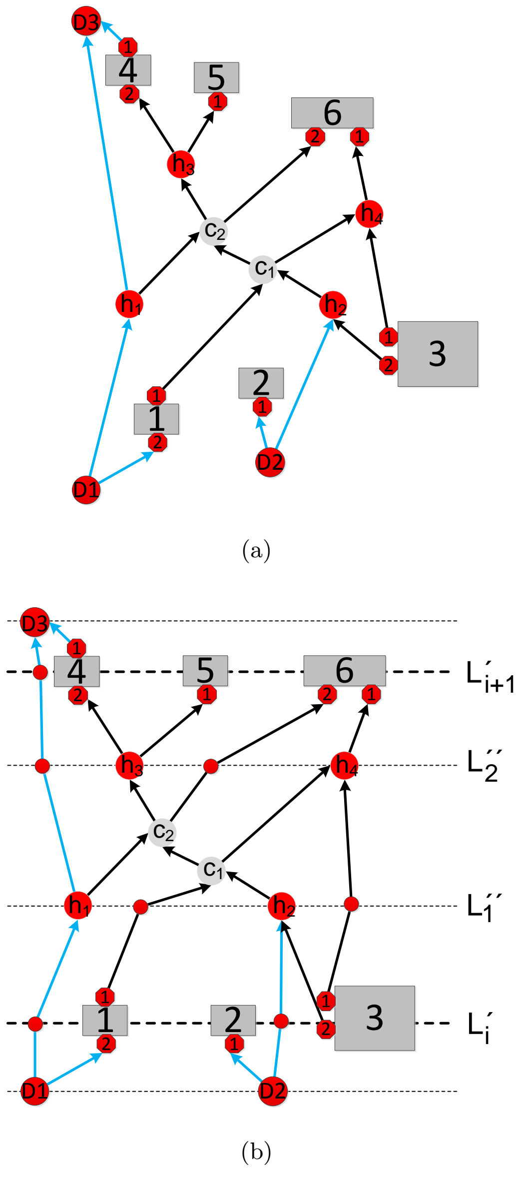


Figure 5.4: A layering of a UPR of a directed hypergraph: The red nodes labeled with h are hypernodes, nodes labeled with C are crossing dummies, and the red cycles are long arc dummies. The blue arcs are subarcs of a chain. The ports of on the nodes are numbered and drawn as red octagons. (a) A subgraph of a UPR of a hypergraph. (b) Coarse and fine layering of the subgraph.

Algorithm 7 Reduce the number of long arc dummies.

Require: Layering $\mathcal{L} = \langle L_1, \dots, L_k \rangle$ of G
1: **for** each source $s \in G$ in decreasing order of their layering index j **do**
2: mark the subgraph dominated by s
3: ▷ Let M_i be the set of marked nodes on layer L_i $(1 \leq i \leq k)$.
4: **for** $i = j + 1$ **to** k **do**
5: **if** all nodes of M_i are long-arc dummies **then**
6: (a) remove the nodes M_i
7: (b) lift the marked subgraph on the layers below L_i by one layer
8: $\rightarrow$ new layering $\mathcal{L}'$
9: **if** $\mathcal{L}'$ causes more crossings **or** more dummies **then**
10: **undo** step (a) and (b)
11: **break** ▷ continue with next source
12: **end if**
13: **end if**
14: **end for**
15: **end for**

form of seemingly unnecessary crossings when these additional objects are deleted; see, for example, the white node in Figure 5.5. To overcome this, we sift each source s through all possible positions on its layer and choose the position where it causes the fewest crossings.

5.3.3 Polyline Hierarchical Layout

We now discuss here how to compute a *polyline hierarchical layout*, that is, a hierarchical upward drawing in non-orthogonal style, which is a realization of $\mathcal{R}$ and $\mathcal{L}$. During the layout we treat the hypernodes as ordinary nodes. In the final drawing, the hypernodes are substituted by branching points and the dummies by bend-points.

Coordinate Assignment

x-coordinate. Conceptually, we can use any coordinate assignment strategy (for example, `BJL-Layout` by Buchheim et al. [BJL99]; `GKNV-Layout` by Gansner et al. [GKNV93]), which is known for Sugiyama's layout algorithm to compute the x-coordinates. All these methods assign horizontal coordinates to the nodes while preserving the given node ordering on each layer, hence also preserve the number of crossings. The aim is to compute the x-coordinates for the nodes and bend-points such that the subdivided long arcs are drawn as vertical straight-lines.

y-coordinate. Usually, the vertical coordinates for the nodes on layer L_i are simply given by $\delta \cdot i$, where δ is the minimal layer distance. Yet, often we

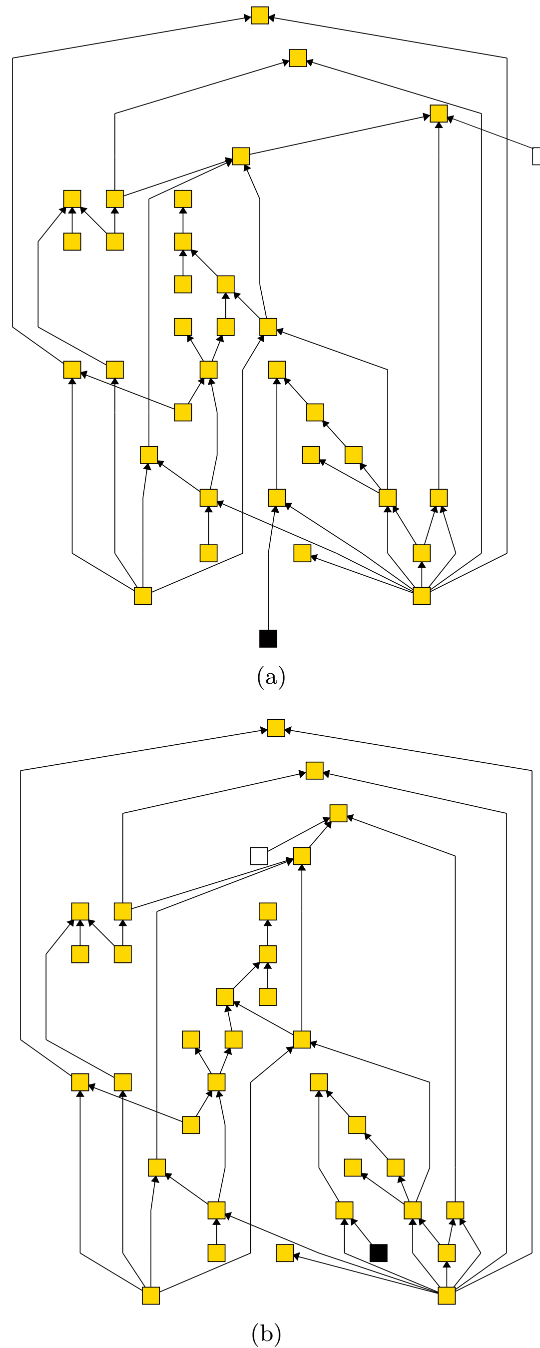


Figure 5.5: A drawing of graph *grafo2379.35* (Rome graphs): (a) without post-processing, (b) after applying source repositioning (white node) and long-arc dummy reduction (black node).

may prefer larger distances between layers in order to counter the following problems:

- *node-arc crossings*: A line segment connecting nodes or bend-points between layer L_i and L_{i+1} may cross through some nodes of these two layers. This can easily happen when node sizes are relatively large compared to the layer distance.
- *long-line segments*: The general direction of upward drawings should naturally be along the vertical direction. Yet, there can be arc segments between two consecutive layers L_i and L_{i+1} which are very long since they span a large horizontal distance. Such long segments can make "Sugiyama-style" drawings hard to read.

Buchheim et al. [BJL99] propose a solution in which the distances of the layers are variable computed with respect to the gradient of the line segments. However, our experimental evaluations have shown that drawing DAGs using upward planarization tends to produce drawings with large height. Therefore we use a different approach which limits the maximal layer distance to 3δ.

Let σ_i be the number of arcs and subarcs between L_i and L_{i+1} whose lengths are at least 3δ. We set the vertical distance between these two layers to $(1 + \min\{\sigma_i/4, 2\})\delta$ (empirically evaluated).

Due to the limit of 3δ, the problems of node-arc crossings and long-line segments may not always resolved. In the upcoming subsection, we will explain how to tackle this problem by introducing additional bends.

Port constraints. Regarding the port constraints, we need to consider them only when we route (draw) the arcs, since they determine where the arcs shall touch the connected nodes. But in case of ports that are located on the left or right side of a node, we can artificially broaden the node prior to the horizontal coordinate calculation in order to allocate enough space for the arc routing (see Figure 5.6) and then add one bend-point per arc such that incoming and outgoing arcs are redirected downwards and upwards, respectively.

Drawing

In order to obtain a drawing of the original graph which realizes $\mathcal{L}$, we have to perform the following post-processing steps:

a) Connect the end nodes of each arc with span of one directly by a line segment.

b) Substitute each sequence of subarcs corresponding to a long arc or to a chain by a polyline. (Recall that the long arc dummies and dummies of the chains are now considered as bend-points of the polyline.)

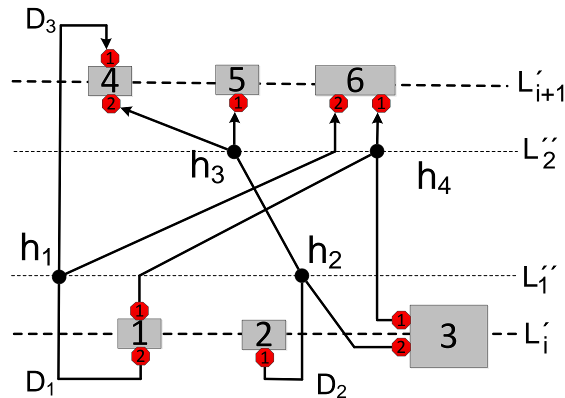


Figure 5.6: A drawing of the subgraph of Figure 5.4. Observe that due to the ports located on the left hand side of the node 3, we have to artificial broaden this node before the final coordinate assignment in order to allocate enough space between node 2 and 3. The bend-points D_1–D_3 are formally dummy nodes of the corresponding chain.

c) Eliminate all hypernodes by directly connecting the line segments of the incoming and outgoing arcs. Thus, the hypernodes are replaced by branching points.

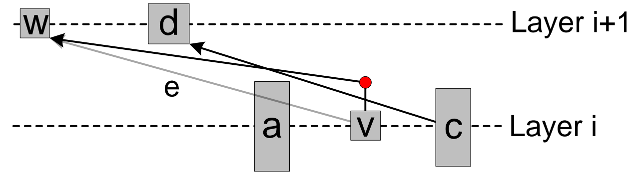
d) Reverse all arcs of F, which were previously reversed to break cycles.

Figure 5.6 shows a realization of the layering of Figure 5.4, where the bend-points D_1–D_3 corresponding to the dummy nodes of the chains are connected to their corresponding nodes via orthogonal polyline.

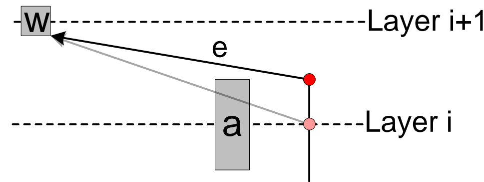
We can beautify the drawing by applying post-processing which bends some arcs and then recompute the coordinate of the bend-points:

Bending arcs. While enlarging the layer distance also helps to prevent node-line crossings, the required increase in height is usually not worth it from the readability perspective. We therefore propose a strategy that allows trading additional bend-points for layer distance. The strategy can be parameterized to find one's favorite trade-off between these two measures, namely, increase the layer distances to reduce the number of bend-points or keep the layer distances small, and instead introduce new bend-points to avoid node-line crossings. Note that these strategies are not only applicable to our layout algorithm, but to any Sugiyama-style layout.

A line segment $\epsilon = (v, w)$ is pointing *upward from left to right* (*right to left*) if $X(v) < X(w)$ ($X(v) > X(w)$, respectively). Since purely vertical line segments cannot cross the nodes which are assigned to layer $L(v)$ or $L(w)$, we distinguish four cases:



(a) Bending the arc/line segment e causes additional arc crossings.



(b) Avoiding node-line crossings by shifting up the bend-point on layer i.

Figure 5.7: Avoiding node-line crossings.

(a) ϵ is pointing upward from right to left and v is on layer i

(b) ϵ is pointing upward from right to left and w is on layer i

(c) ϵ is pointing upward from left to right and v is on layer i

(d) ϵ is pointing upward from left to right and w is on layer i

In all these cases, ϵ has to bend if it overlaps some nodes of L_i. However, bending ϵ might cause additional arc crossings; see Figure 5.9(a). To avoid this, we also have to bend the line segments that cross the just bended line. Without loss of generality, we only discuss the case (c). The other cases can be solved analogously.

Let $Width(v)$ and $Height(v)$ denote the width and height of the bounding box of a node v. Let a be the node on layer L_i with the highest bounding box, and let $\kappa = Height(a)/2$. If v is a bend-point and not shifted downwards before, then we do not need to introduce an additional bend. Instead we move v upwards by κ; see Figure 5.9(b) If v was already shifted downwards before due to one of the other cases, then we bend ϵ by introducing a new bend-point b and set $X(b) = X(v)$ and $Y(b) = Y(v) + 2\kappa$. We observe: By setting $Y(v)$ to κ or in the latter case, setting the new bend-point b to $Y(v) + 2\kappa$, we ensure that ϵ cannot overlap any nodes on layer L_i.

Assume v is not a bend-point. Then we have to introduce a bend-point along ϵ, and we have to consider that other arcs might also get rerouted and so we must accommodate enough space for them as well, such that no two bend-points may coincide. In particular, it might be that the arcs leaving v's left neighbor to the right might also require additional bend-points (see

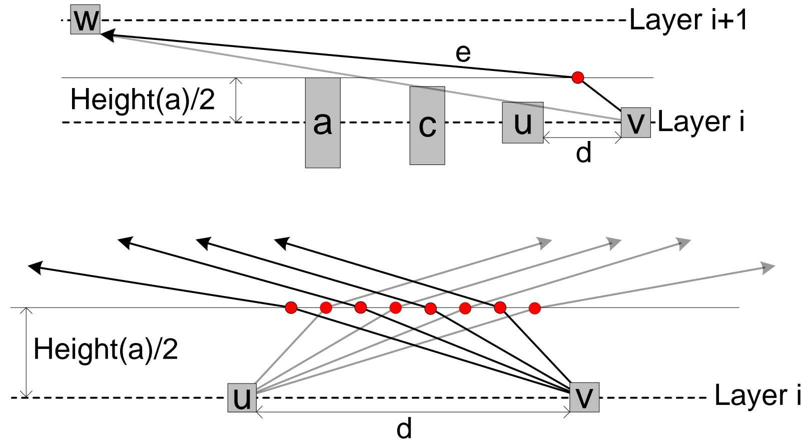


Figure 5.8: Avoiding node-line crossings by introducing new bend-points into an arc e (top); all horizontal coordinates of the bend-points must be distinct, especially when all involved arcs require a bend (bottom).

Figure 5.8). Let u be the left neighbor of v on L_i and

$$d = \mathrm{X}(v) - \mathrm{X}(u) - \mathrm{Width}(v)/2 - \mathrm{Width}(u)/2$$

their inner distance. Let r be the number of line segments adjacent to v and pointing from right to left; among these, assume that ϵ is the j-th segment when counting from left to right. Let q be the number of line segments adjacent to u and pointing from left to right. Then, $\Delta = \frac{d}{q+r+1}$ gives the distances between the potential bend-points, and the coordinates of the new bend-point b are:

$$\begin{aligned}
\mathrm{X}(b) &= \mathrm{X}(u) + \frac{\mathrm{Width}(u)}{2} + \Delta \cdot (j + \min\{q, j-1\}) \\
\mathrm{Y}(b) &= \mathrm{Y}(v) + \kappa
\end{aligned}$$

In the worse case we have to introduce a new bend-point for each line segment in order to prevent overlapping of the bend-points and to prevent newly arising arc crossings (see Figure 5.8 (bottom)). Therefore the number of newly introduced bend-points for a layer L_i is bounded by the number of line segments connecting the nodes or bend-points of L_i and L_{i+1}. Figure 5.9 gives an example where arc bending is used.

5.3.4 Experimental Evaluation

Considered algorithms. We compare the new drawing framework `UPL` to the following algorithms: `Dominance`, `Visibility`, `UPSugiyama` and `Sugiyama`.

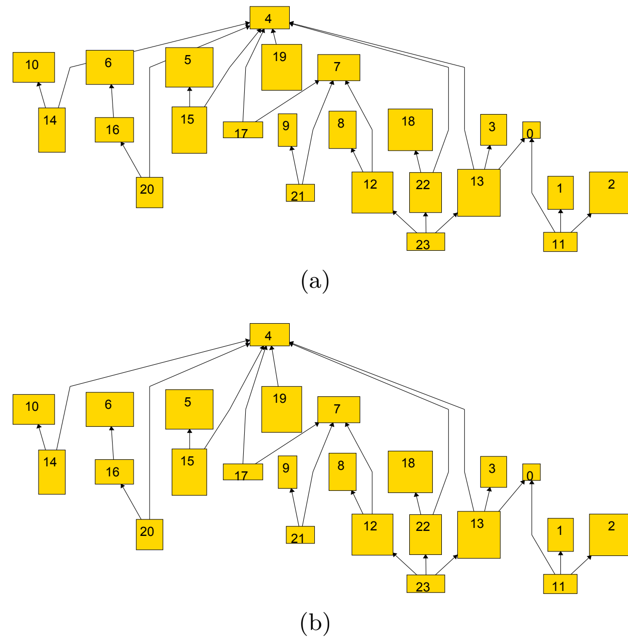


Figure 5.9: A drawing of graph *grafo159.24* (Rome graphs) with random node sizes: without (a) and with (b) the bending arc method and individual layer distance assignment. The drawing in (a) contains two arc-node crossings (of nodes 5 and 6).

We call the first three drawing algorithms *alternative upward drawing approaches*.

As shown by our previous experiments, `GKNV-Layering` computes layerings with similar height as `LongestPath` but it ensures that the total number of long arc dummies is at the minimum, hence it offers very satisfying results. Furthermore, it is quite fast. Thus, `GKNV-Layering` is used by `UPSugiyama`, `Sugiyama` and also used by `UPL`. For coordinate assignment of `Sugiyama` and `UPL`, the approach `GKNV-Layout` by Gansner et al. [GKNV93] is applied. Although this approach offers the best results, it is not used for `UPSugiyama` due to the fact that the LP-based approach requires too much time. Recall that `UPSugiyama` layouts the whole UPR that possibly contains a huge number of crossing dummies. Hence, instead of `GKNV-Layout`, the fast layout algorithm `BJL-Layout` by Buchheim et al. [BJL99] is applied. So we can assume that the most competitive algorithms for the individual steps of `Sugiyama` and `UPL` are used.

All algorithms are implemented in the free and open-source Open Graph Drawing Framework (OGDF) [ogd].

Benchmark sets. We use the following benchmark sets without any given port constraints: Rome graphs, the North DAGs, and the random DAGs (see Section 4.3.1). The North DAGs used in Section 4.3 were limited to

instances with $|A| < 100$ and are grouped according to the number of arcs in order to allow us to compare to the drawing algorithms `Dot` and `Layers`. The North instances used here are not limited, hence all 1277 digraphs were used. Further, the instances are grouped into nine sets, where the first set contains digraphs with 10 to 20 nodes and the i-th set contains $10i + 1$ to $10(i + 1)$ nodes for $i = 2, \dots, 9$.

All data points of the diagrams represent average values for the corresponding node or density group. This allows us to refer to the data of the experimental evaluations of Section 4.3 for the Rome graphs and the random DAGs.

Considered criteria. The algorithms `Dominance` and `Visibility` are simple drawing algorithms originally developed for drawing planar *st*-graphs. `Sugiyama` follows a very different paradigm than `UPL` and the alternative drawing approaches make it difficult to compare directly. Hence we use the following conventions for the general evaluation:

height: The height of a drawing is simply the number of required layers (in case of `UPL` and `UPSugiyama`) or the number of vertical grid coordinates (in case of `Dominance` and `Visibility`), respectively.

width: The width of a drawing is defined as the maximum number of *elements* per layer or horizontal grid line, respectively, where the elements on a layer or grid line Λ are the nodes on Λ as well as the edge lines crossing Λ.

area: The required drawing area is defined as *height* $\times$ *width*.

aspect ratio: The aspect ratio is defined as *width*/*height*.

impression: The aforemetioned criteria can only give us a hint about the quality of the drawings, but the best way to evaluate is still by manually inspecting the drawings. Therefore we introduce the criterion *impression* which reflects the properties of a drawing that cannot be straight-forwardly formalized, that is, in particular the reflected structures, clarity, and tidiness of the upward drawings. For evaluating this criterion the author has inspected hundreds of drawings.

Observe that even when the coordinate assignment of `Dominance` and `Visibiliy` is not elaborated, the definition of the criteria–except *impression*– is given regardless of any differences which are only due to spacing parameters, hence allows a fair comparison. One important criterion, the number of crossings, is not listed above. For this, we refer to the experimental evaluation of Section 4.3.

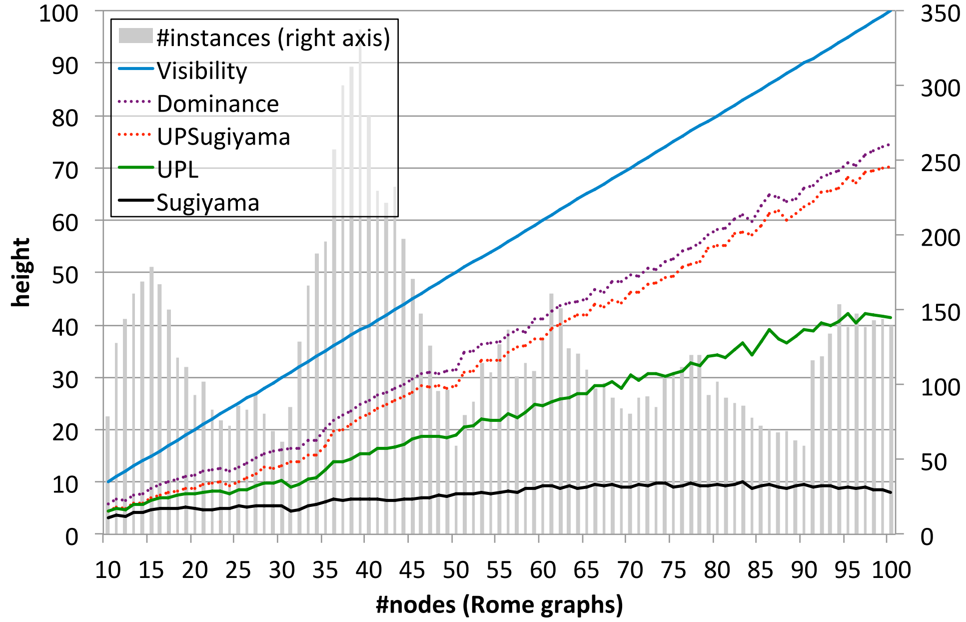
General Evaluation

As outlined in the introduction of this chapter, there are various alternatives to realize a UPR of a digraph. Therefore, the UPR $\mathcal{R}$ used for `UPL` is also used as input for the alternative drawing algorithms `Dominance`, `Visibility`, and `UPSugiyama`.

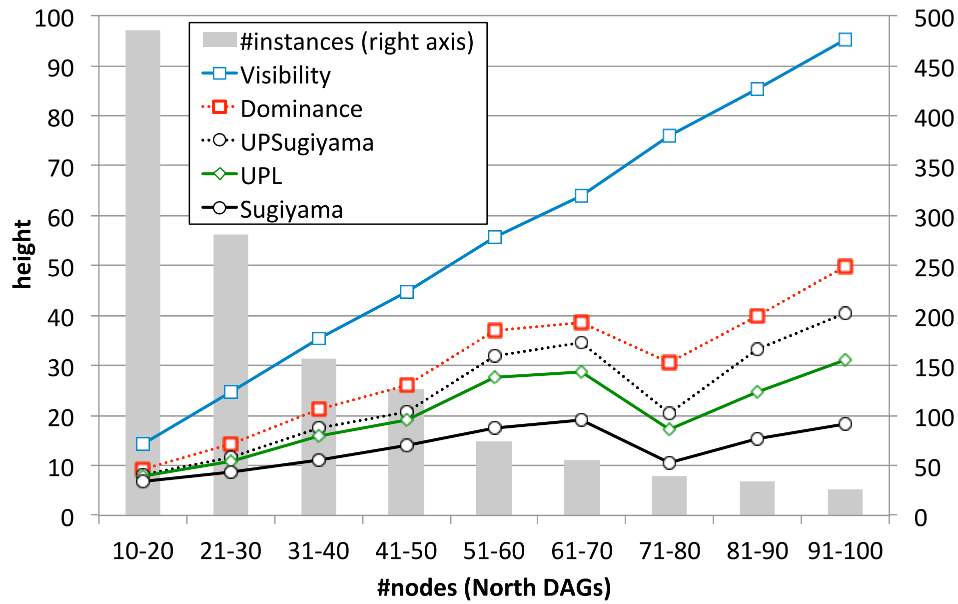
Height. Figures 5.10–5.11 show the average height of the upward drawings for the Rome graphs, the North DAGs, and the random DAGs, respectively. We observe that except for the North DAGs, the height of the drawings tends to increase with increasing number of nodes. This is not surprising, since with increasing number of nodes, the length of the longest path typically also increases. The diagram for the North DAGs (Figure 5.10) contains a break at group 71–80. The reason is that this group has the shortest average longest path and the lowest average density. Inspecting the plot of Figure 5.11, we can see that the average height of the drawings increases when the graphs become denser. Also the gap between `UPL` and the alternative upward drawing algorithms decreases, and in addition, the height converges to the maximal possible height. The plots of the Figures 5.10–5.11 illustrate the impact of the layering approach `UPL-Layering` to the height of the upward drawings: We can see a large gap between `UPSugiyama` and `UPL`.

Regarding `Sugiyama`, the plots reflect the strength and also the weakness of this framework: On the one hand, it only takes into account the topological order of the nodes and ignores all other graph theoretical properties, hence its drawings have minimal height when using `LongestPath` or nearly minimal height when using `GKNV-Layering` (see Section 3.1.2). But on the other hand, such compact drawings may not always be an advantage, in particular in the context of visualizing graph structures, for example, recall Figure 1.3. Furthermore, we can see that the average height of the `Sugiyama`'s drawings only increases slightly with increasing number of nodes and density of the graphs. In particular, this holds for the Rome graph where the ascending slope of the corresponding plot becomes flat with increasing number of nodes. Hence straight-forwardly layering the Rome instances lead to very unsatisfying results. In contrast, the average height of the drawings of `UPL` increases with increasing number of nodes which is more reasonable. We will discuss this fact in detail in section 5.3.4.

Regarding height, we conclude that the new approach `UPL` computes layerings with notably fewer layers than the alternative upward drawing approaches. Since these algorithms—as well as `UPL`—realize UPRs, we can conclude that, in contrast to `Sugiyama`, the low height is achieved without the lost of the structure informations that are mapped from the input graph to its UPR. Hence, the visualized structures in the upward drawings produced by UPL and the alternative approaches are identical.

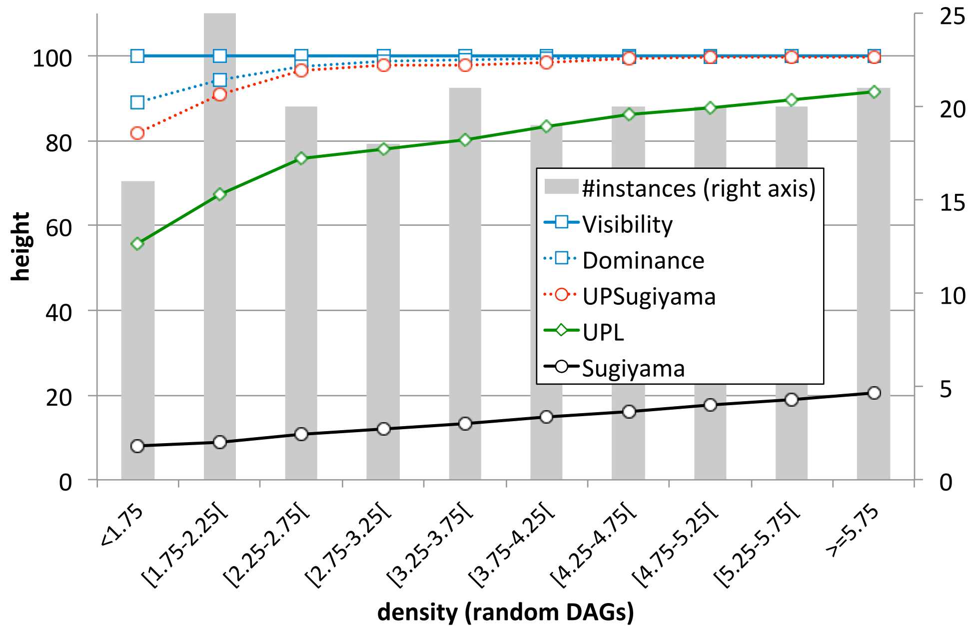


(a) Rome graphs: average height vs. number of nodes.



(b) North DAGs: average height vs. number of nodes.

Figure 5.10: Comparison of the height of the upward drawings.



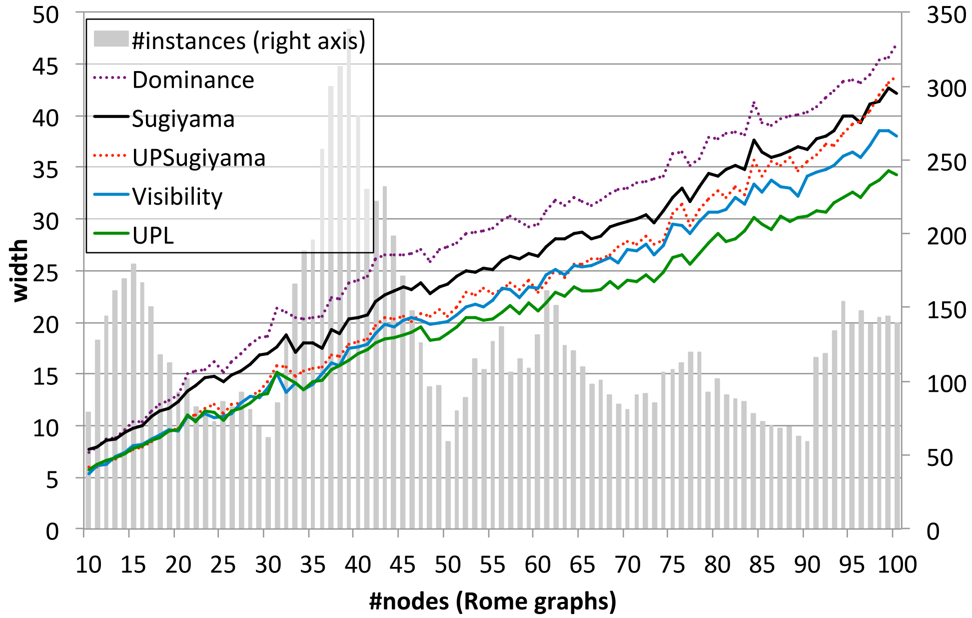
(a) Rome graphs: average height vs. number of nodes.

Figure 5.11: Comparison of the height of the upward drawings.

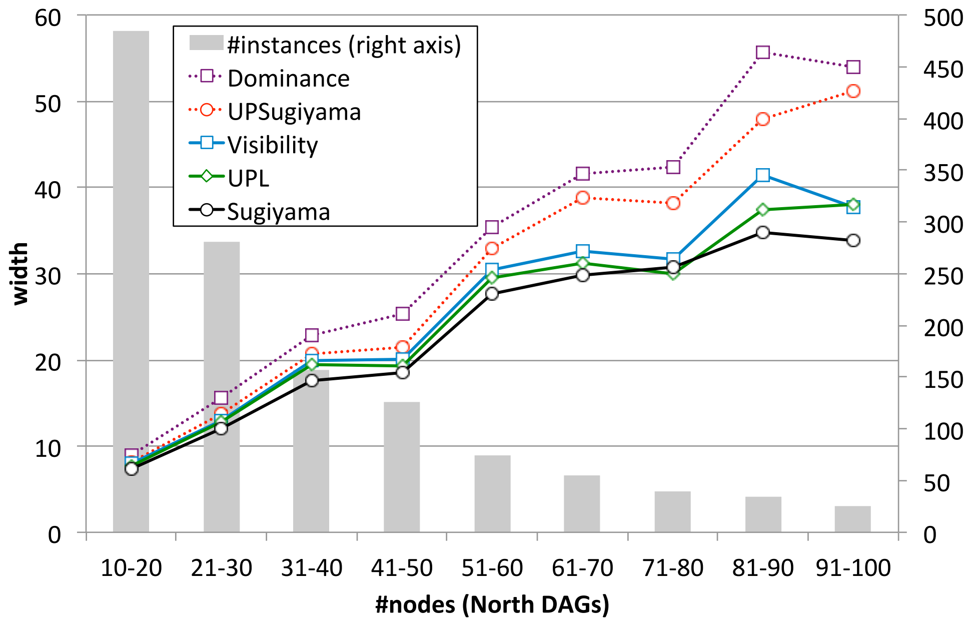
Width. The results regarding the average width of the drawings are illustrated in Figures 5.12–5.13. As expected, the average width increases with increasing density and the number of nodes of the instances. The reason is that the digraphs with high numbers of nodes "need" more layers, and thus there are more long arcs, hence, the number of long arc dummies on each layer increases.

In comparison to the alternative upward drawing approaches, the new approach `UPL` produces layerings with smaller width on the instances of the Rome graphs and the North DAGs, while all upward planarization based algorithms perform nearly the same on the random DAGs. In comparison with `Sugiyama`, `UPL` often produces drawings with smaller width for the Rome and North instances, while `Sugiyama` achieves better results for the random DAGs. The latter observation is due to the lower average height of the `Sugiyama` drawings which results in fewer long arc dummies. With increasing density, the impact to the width by the increasing number of dummies becomes more significant.

Aspect ratio and drawing area. The average aspect ratio and the average area requirement of the drawings are given in Figures 5.14–5.15 and in Figure 5.16–5.17, respectively. Regarding the aspect ratio, we expected that Sugiyama produces upward drawings that are broader than the drawings of the upward planarization based algorithms, since it ignores the upward

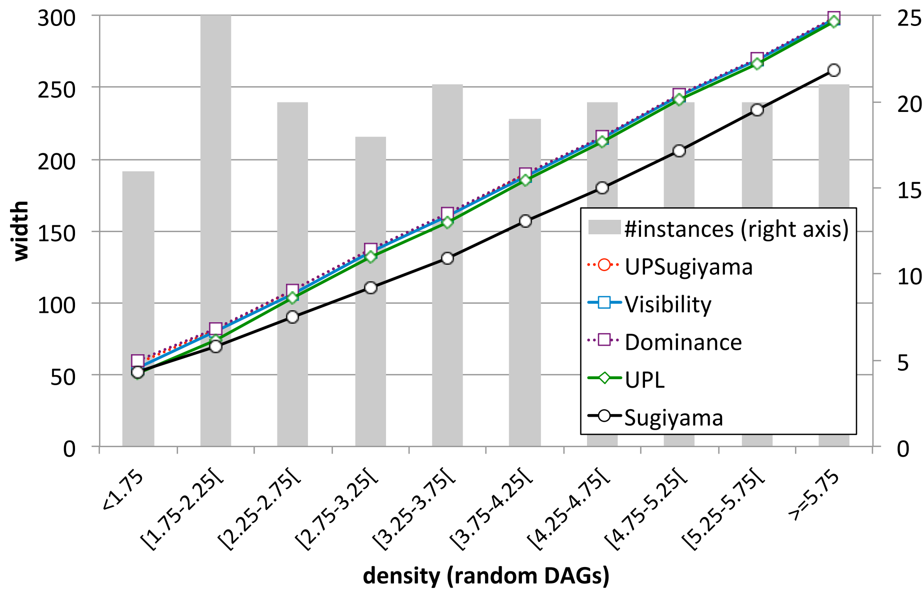


(a) Rome graphs: average width vs. number of nodes.



(b) North DAGs: average width vs. number of nodes.

Figure 5.12: Comparison of the width of the upward drawings.



(a) Rome graphs: average width vs. number of nodes.

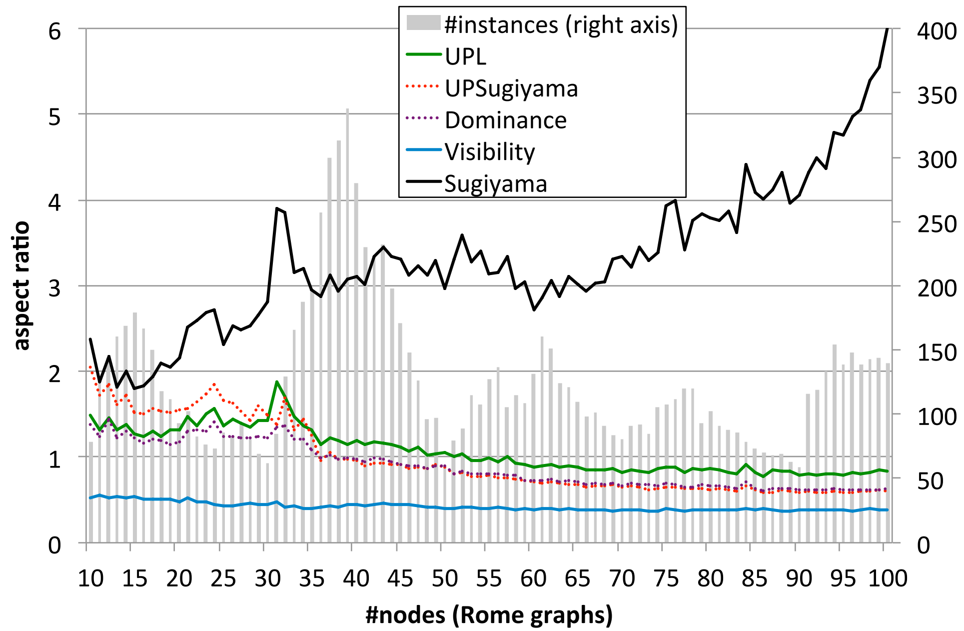
Figure 5.13: Comparison of the width of the upward drawings.

property of the input graphs. Indeed, we can observe this in the plots of Figures 5.14–5.15. Due to the characteristic of `Visibility`—it always requires a height which is identical to the number of nodes—its aspect ratio is the smallest among all considered algorithms. Overall, we can observe that in comparison to `Sugiyama`, `UPL` and the alternative upward drawing approaches obtain a more balanced aspect ratio, therefore they are more suitable for visualizing flows.

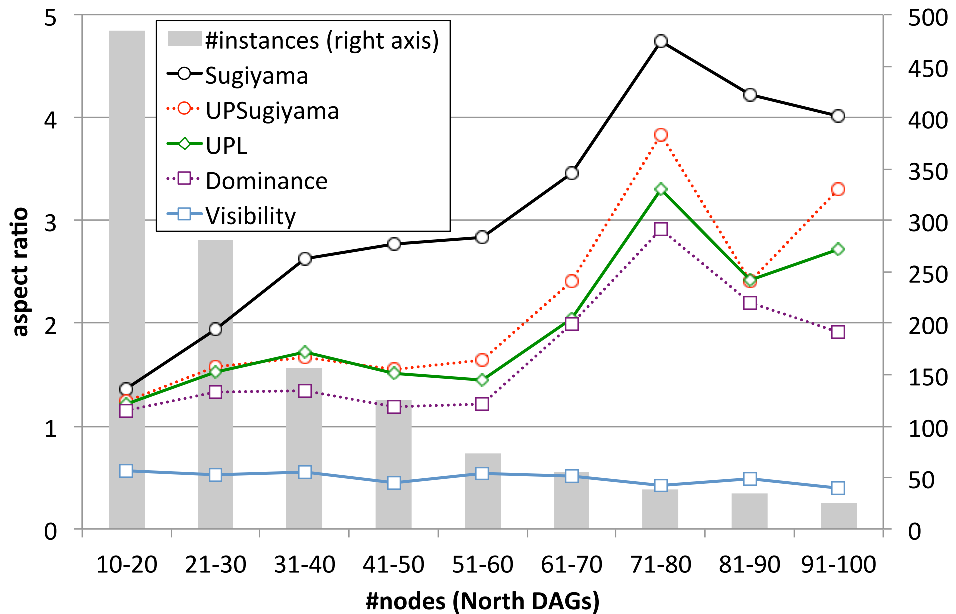
In accordance with the plots regarding the average height and width of the upward drawings, we see that the `UPL` drawings require less drawing area than drawings of the alternative upward drawing approaches. We also observe one characteristic of `Sugiyama` when used for visualizing the Rome instances: The average area requirement of the drawings only increases slightly when the graphs become larger. This results in too compact and unpleasant upward drawings.

Deeper Analysis

The best way to evaluate the quality of the drawings is by manually inspecting them and then setting the impression in relation to the obtained experimental data. For this intension, the author has grouped the instances by the number of nodes, density, and the number of crossings (computed by `LFUP`50), then randomly selected drawings from each group and evaluated them manually with respect to the criterion *impression*.

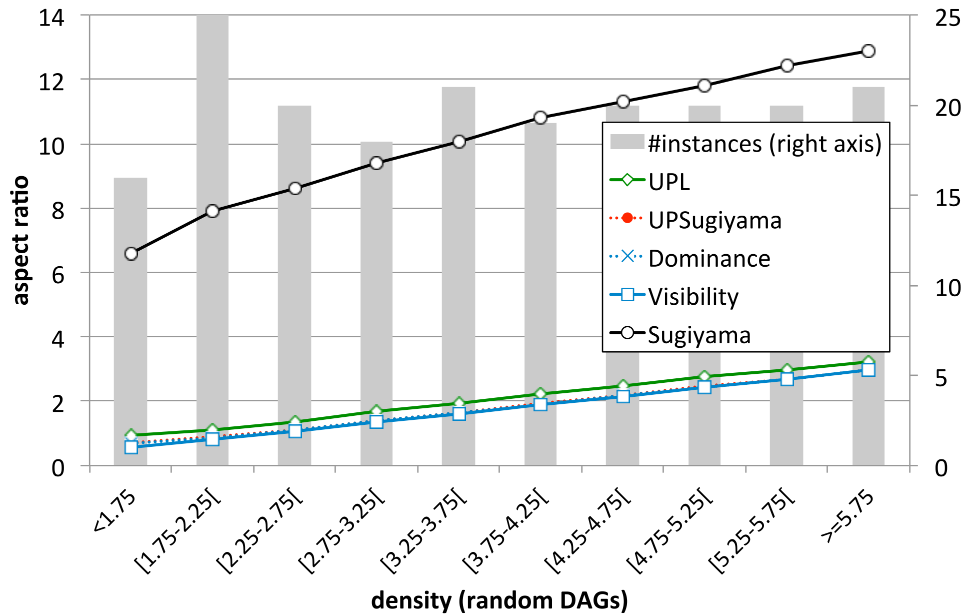


(a) Rome graphs: average aspect ratio vs. number of nodes.



(b) North DAGs: average aspect ratio vs. number of nodes.

Figure 5.14: Comparison of the aspect ratio of the upward drawings.



(a) Rome graphs: average aspect ratio vs. number of nodes.

Figure 5.15: Comparison of the aspect ratio of the upward drawings.

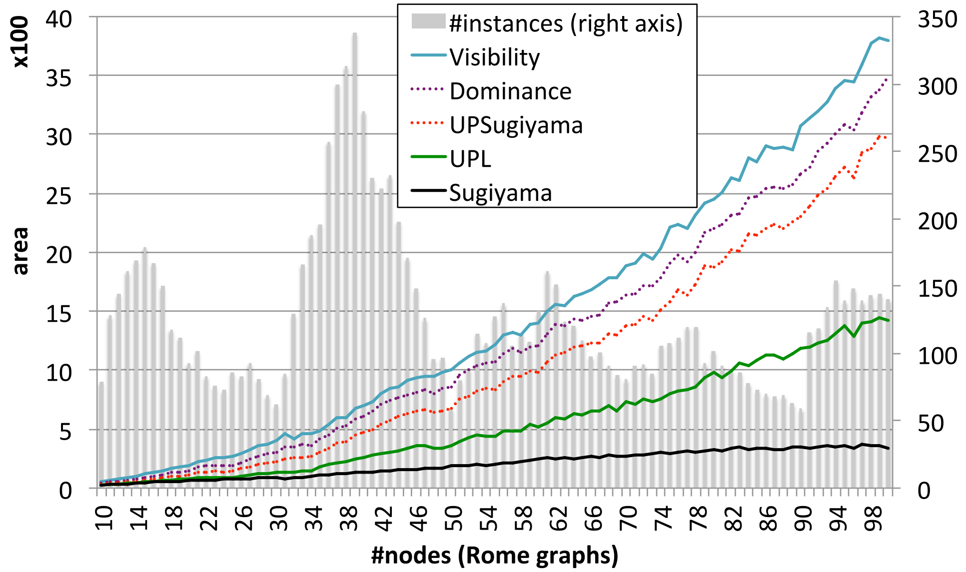
Alternative Upward Drawing Algorithms

Regarding the algorithms `Dominance`, `Visibility`, and `UPSugiyama`, we conclude that drawing the UPR including dummy nodes and then replacing the crossing dummies by arc crossings produces unsatisfying results, even for small and sparse digraphs, which can be considered as to be drawn easily. The crossing points are often also bend-points of the corresponding arcs (`Visibility` and `UPSugiyama`) and limiting the number of bends to at most one for each arc (`Dominance`) is counter productive and increases the drawing area significantly (see Figures 1.1 and 5.2). Considering the evaluation of the criteria like *height* or *width* which are defined regardless of the final coordinate assignment, we conclude that these alternative algorithms are not competitive to `UPL`.

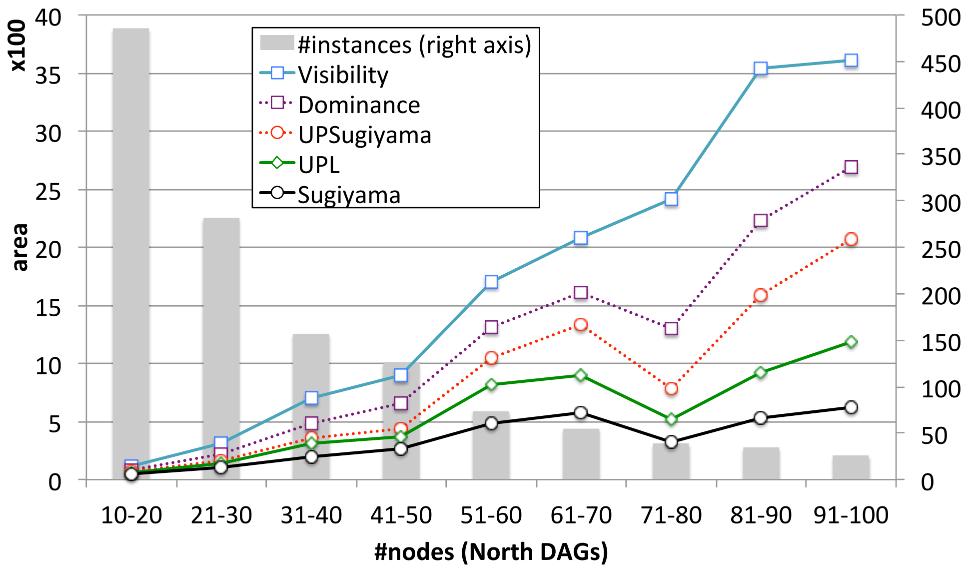
Framework by Sugiyama et al.

Due to the low height of the drawings, the visualized graph is compressed to fit into the given limit. This has the side-effect that many nodes are assigned to few layers which results in broader drawings and in a typical drawing characteristic of `Sugiyama`, i.e, many arcs are routed criss-crossing between the layers; for example, see Figure 5.36 in Section 5.5 (Drawing Gallery).

Structures of the input graph which require a height beyond the given limit are not correctly or cannot be visualized. Even when a structure fits into the

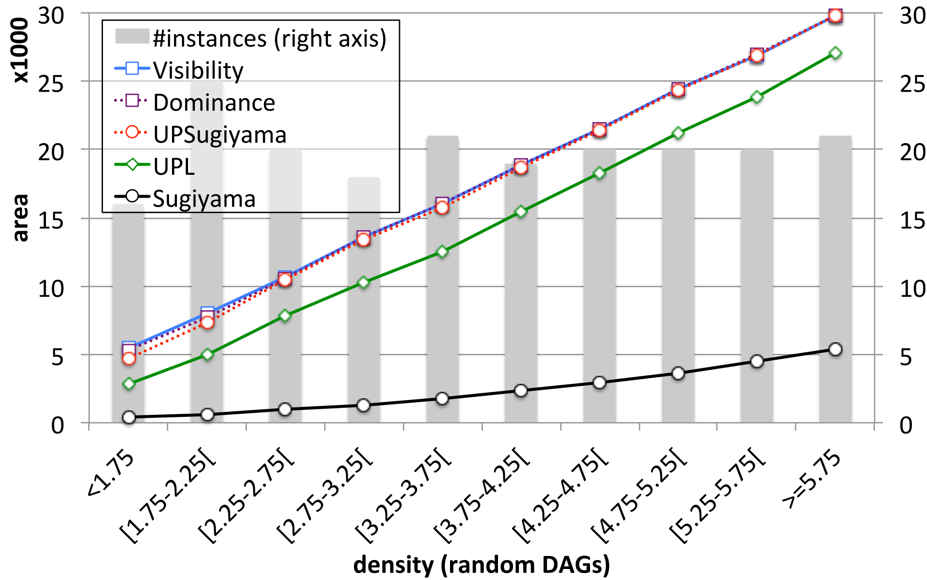


(a) Rome graphs: average area requirement vs. number of nodes.



(b) North DAGs: average area requirement vs. number of nodes.

Figure 5.16: Comparison of the area requirement of the upward drawings.



(a) Rome graphs: average area requirement vs. number of nodes.

Figure 5.17: Comparison of the area requirement of the upward drawings.

given limit of the height, due to the strong compactness of the drawing, it is overlapped by the criss-crossing routed arcs, making it hardly to detect; for example, see Figure 1.3. As shown by the previous evaluations, the average length of a longest path is quite small with respect to the number of nodes, arcs, and the density of the digraphs. So the visualization of the large or dense digraphs often results in squeezed upward drawings. Hence, our impression of the upward drawings can be summarized as follows:

> *The drawings produced by* `Sugiyama` *are unnaturally flat with many arcs routed criss-crossing between the layers. Often, it seems that the nodes are arbitrarily or randomly assigned to the layers and structures of the visualized digraphs are hardly reflected.*

Random runs and manual post-processing. A simple method for improving the results is to start k random runs of the drawing algorithms and then inspect the k drawings and choose the best one. For further improvement, one can modify the chosen drawing by hand. However, since only the crossing minimization step is randomized, the improvement for `Sugiyama` obtained by random runs is marginal; for example, see Figure 5.25 in Section 5.5 (Drawing Gallery). Since the layering plays an important role regarding the quality of an upward drawing, it requires more effort than a few minutes to re-layer the upward drawing manually. Hence neither of both methods can considerably improve the final results, but by re-positioning certain nodes,

some arc crossings can be eliminated. Further, increasing the distance of the nodes and the layers can reduce the squeezing effect of `Sugiyama`.

Upward Planarization Layout

Unlike the layered approach for upward crossing minimization, `LFUP` does not require any layering of the input digraph, hence no squeezing effect arises in the upward drawings of `UPL`. Most oft the arcs are drawn as vertical straight line, hence, in contrast to `Sugiyama`, only very few arcs are routed criss-crossing between the layers. However, the number of such arcs increases with increasing size and density of the instances. As shown in the previous subsection, for large ($|V| > 70$) or dense digraphs, the area requirement increases considerably. This has the side-effect that the length of the arcs increases since they span more layers. Another side-effect is that long arcs connecting nodes between two consecutive layers may hardly to be distinguish because the minimal distances between two arcs cannot be kept due to the given maximal layer distance. Furthermore, the node order on each layer is determined by the UPR, hence optimized for crossing minimization and not for minimizing the arc length. Choosing another node order may decrease the length of some arcs without to increase the number of arc crossings. Fortunately, this negative effect occurs only for large or dense instances; for instances with $|V| \leq 70$ we conclude:

> *The upward drawings produced by `UPL` offer clarity, are tidy, and have relatively few arc crossings. In comparison to `Sugiyama`, the drawings better reflect the structures of the digraphs.*

Random runs and manual post-processing. Unlike `Sugiyama`, random runs have several effects to the final drawing since the layering and the order of the nodes on each layer can be changed; for example, see Figure 5.26. Hence, we can use random runs to obtain an UPR whose realization suffers less from the negative effects due to the large area requirement. In addition, by manual re-routing troublesome arcs and re-positioning certain nodes, the results can be improved significantly.

Conclusion

On the crucial criteria *crossings* and *impression* `UPL` outperforms `Sugiyama`. It delivers well structured and tidy drawings with considerably fewer arc crossings for the most real-world based instances. Furthermore, `UPL` draws upward planar sT-graphs without any arc crossings, while Sugiyama only considers the hierarchy of the nodes, which can lead to unpleasant drawings even when the digraphs are considered to be easy to draw; for example, see Figure 5.34 in Section 5.5 (Drawing Gallery). Also recall that over 60% of the North DAGs are upward planar which also includes non sT-graphs. Although

Algorithm	Rome graphs		North DAGs		random DAGs	
	avg	max	avg	max	avg	max
Visibility	0.00022	0.002	0.00013	0.006	0.035	0.125
Dominance	0.00025	0.002	0.00013	0.005	0.032	0.125

Table 5.1: Average and maximal runtime of `Visibility` and `Dominance` in seconds.

the upward drawings of large instances of `UPL` may contain miss-routed arcs, it is worth to re-route them manually instead of using `Sugiyama`.

`Sugiyama` is only preferable if compact visualization has a high priority or if the instances have a high upward crossing number. For these digraphs, reducing the number of arc crossings does not considerably improve the readability.

Runtime

We use here the same hardware as described in Section 4.3, hence the data distinguishes from the data of publications [CGMW11] and [CGMW10b].

The `UPL` version which uses the fast layout algorithms `BJL-Layout` is denoted as `UPL Fast`. The average runtimes (in seconds) of `UPL`, `UPL Fast`, `UPSugiyama` and `Sugiyama`50 are shown in Figures 5.18–5.19. For the latter one, the stated runtime is the runtime of the whole framework, in particular including the runtime of the crossing minimization step, while the runtime of the first three algorithms refer only to the layout.

The runtimes of the linear-time algorithms `Dominance` and `Visibility` are omitted, since they are usually below any measurable threshold. Instead, we give their average and maximum runtime values in Table 5.1.

On the one hand, `UPL` uses the same layering and layout algorithm as `Sugiyama`50, but the runtime of the latter includes the crossing minimization step, hence we expect that `UPL` is faster than `Sugiyama`50.

On the other hand, `UPL` produces layering with much more layers than `Sugiyama`50, thus the layerings have more long arc dummies and hence the runtimes increase. Another aspect which also has great influence to the runtime of `UPL` is the size of the UPRs. These facts coincide with the plots of the runtime data: Most instances ($|V| \leq 75$) of the Rome graphs have small UPRs and the layering computed by UPL is not too high, so we can observe that `UPL` is faster than `Sugiyama50`, but for larger instances the impact of the size of the UPRs to the runtime grows, hence the layout step requires more runtime and `Sugiyama` becomes faster.

The instances of the north DAGs have in general fewer crossings than the Rome graphs; see Section 4.3.1. Furthermore, over 62% of the North DAGs are upward planar. Due to the fact that the OGDF implementation of `Sugiyama` prematurely stops the k-level crossing minimization if no more crossing reduction can be achieved, the runtime of the crossing minimization

step is quite low, hence `Sugiyama50` is faster than `UPL` on the North DAGs. On the plot of the North DAGs we can see a runtime peak for the group of DAGs with $51 \leq |V| \leq 60$. This group contains few instances with very high density which increases the average runtime. (The UPR of some instances have more than 2000 nodes.)

Due to the `UPL-Layering` approach, `UPSugiyama` is generally slower than `UPL`. The runtime of `UPL` increases rapidly when the instances become denser since the size of the UPRs and the height of the layering also increase rapidly. However, `UPL` is not too slow for practice, requiring below 0.25 seconds on average for the large instances of the Rome graphs and the North DAG. Hence the runtime of `UPL` based on `LFUP` is dominated by the upward planarization step.

5.3.5 Orthogonal Layout

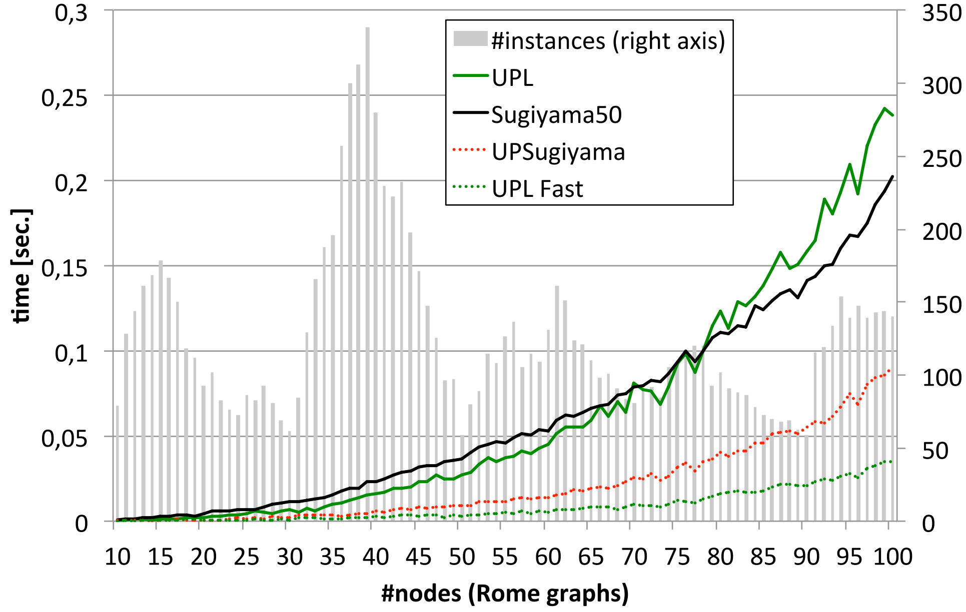
Orthogonal drawings arise in many technical applications like electric schematics or industrial process- and control models. In this section we depict a layout approach for orthogonal drawing of digraphs/directed hypergraphs based on a given proper ordered layering $\mathcal{L}$ which corresponds to a UPR $\mathcal{R}$. The approach consists of three steps:

Orthogonal Layout	
Step 1	Pre-processing
Step 2	Initial Orthogonal Layout
Step 3	Orthogonal Compaction

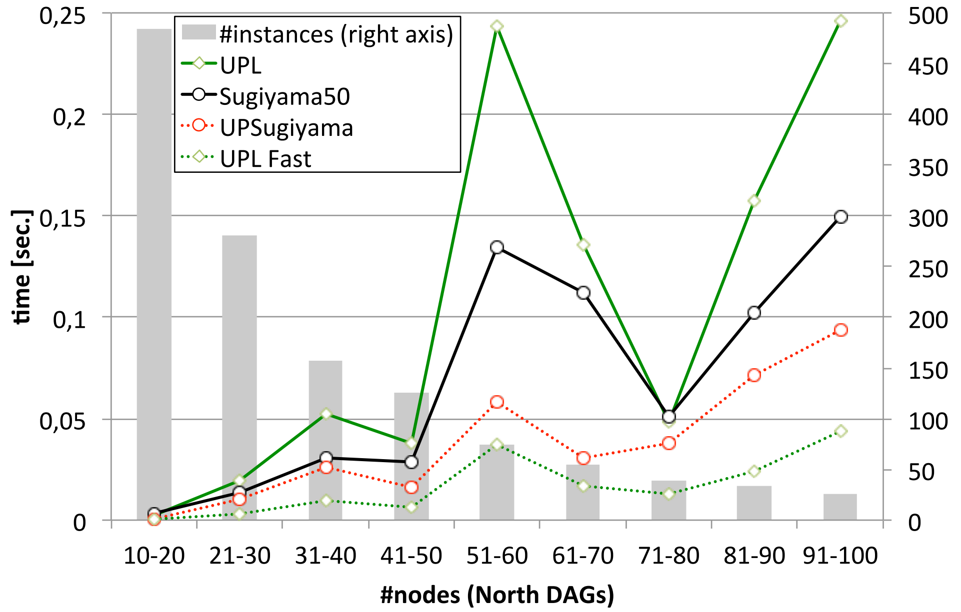
In the first step, we determine for each subarc incident to a crossing dummy whether it is drawn as vertical or horizontal line. Then in the second step, we assign the coordinates to the nodes and dummies and draw the arcs in orthogonal style. We obtaine an initial orthogonal drawing $\mathcal{D}'$ that realizes the input layering $\mathcal{L}$. In the final step, we extract orthogonal representations from $\mathcal{D}'$ which allow us to apply orthogonal compaction to $\mathcal{D}'$ without violating the previous achieved results, that is, the upward property and port validity of $\mathcal{D}'$.

Pre-processing

Let C by a crossing dummy that models a crossing of the arcs (a, b) and (u, v) in the UPR $\mathcal{R}$ of the input graph. We have the subarcs $(a, C), (C, b)$ and $(u, C), (C, v)$ in $\mathcal{R}$. Let (a, C) be the left incoming arc in a realization of $\mathcal{R}$, thus (C, b) is the right outgoing arc of C. In an orthogonal drawing, we have to decide whether (a, C) and (C, b) are drawn horizontally or vertically.

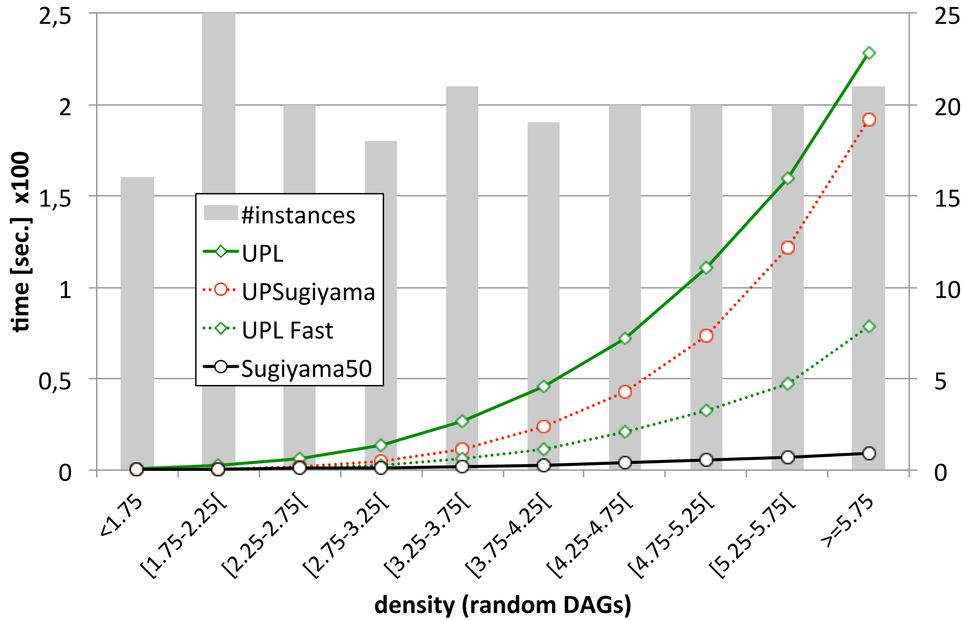


(a) Rome graphs: average runtime vs. number of nodes.



(b) North DAGs: average runtime vs. number of nodes.

Figure 5.18: Comparison of the runtime.



(a) Random graphs: average runtime vs. number of nodes.

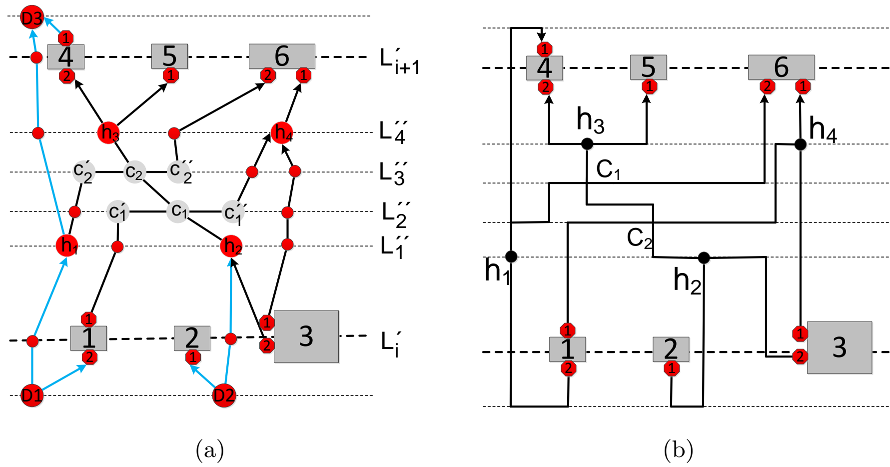
Figure 5.19: Comparison of the runtime.

Assume that we want to draw (a, C), (C, b) and (u, C),(C, v) horizontally and vertically, respectively. We can attain this by the following modification of the given layering $\mathcal{L}$ (see Figure 5.20(a)): We first split C by adding new dummy nodes C' and C'' such that C' is the immediate left and C'' the immediate right neighbor of C on the assigned layer of C. Thereby we incorporate this information into the layering $\mathcal{L}$. Then we redirect the left incoming and the right outgoing arc of C such that C' is the new target and C'' is the new source node of the incoming and outgoing arc, respectively. The arc (a, b) is now represented by the path $\langle a, D', D, D'', b\rangle$. The subarc (D', D) and (D, D'') will be drawn as horizontal lines, thus D' and D'' becomes to bend-points of the arc (a, b) and segments of the arcs (u, C) and (C, v) are drawn as vertical lines. Thus, the crossing corresponding with C is drawn in orthogonal style.

Observe that the number of bend-points occurring in the final drawing can depend on the decision on which corresponding pair of line segments incident to a crossing are drawn as horizontal lines.

Definition 17 (Bend Minimization with respect to a Proper Ordered Layering)**.** Given a proper ordered layering $\mathcal{L}$ of a UPR, find an orthogonal realization $\mathcal{D}$ of $\mathcal{L}$ with minimum number of bend-points.

In this thesis, we do not investigate this problem, instead, we randomly decide on which corresponding pair of line segments incident to a crossing dummy is drawn as horizontal lines.



(a) (b)

Figure 5.20: (a) Coarse and fine layering of the subgraph of Figure 5.4 obtained after the pre-processing. Each crossing dummy C_i is split by adding two new additional dummy nodes C_i' and C_i''. This two new dummies are considered in the initial drawing as bend-points (see Figure 5.20(b)) of the corresponding arcs. (b) The initial orthogonal drawing realizing the layering of (a). The multiple incoming or multiple outgoing arcs of branching points corresponding to h_1 overlap each other. These overlapping line segments need to be merged to single line segments. Further, this drawing contains several unnecessary bend-points which also reduce the quality of the drawing.

Initial Orthogonal Drawing

Now let $\mathcal{L}$ be a proper ordered layering where we have applied the above pre-processing. Analogously to the non orthogonal layout approach of Section 5.3.3, we compute the coordinates of the nodes and bend-points. Nodes respectively bend-points on two consecutive layers can be connected via a polyline drawn in orthogonal style, that is, the polyline consists of only horizontal and vertical line segments (see Figure 5.20(b)). Since the embedding of $\mathcal{R}$ is planar, the polylines can be drawn without causing additional arc crossings. One results of this procedure is that all incoming and outgoing arcs of the hypernodes and dummy nodes end in the same point, where incoming arcs reach the nodes from below and outgoing arcs leave the nodes upwards. If there are multiple incoming or multiple outgoing arcs of a node u, the corresponding vertical line segments that touch u overlap each other. These overlapping line segments need to be merged to single line segments. As results we obtain an initial orthogonal drawing $\mathcal{D}'$.

Orthogonal compaction

The initial orthogonal drawing $\mathcal{D}'$ may contain various unnecessary bend-points which reduce the quality of the drawing. We therefore apply orthogonal

compaction techniques as final step in order to get rid of these bend-points. We first give a brief introduction to the idea of flow-based compaction techniques and then explain how we can apply them to the initial drawing in order to get pleasant drawings.

Compaction algorithms. An *orthogonal representation* of a graph G is an extension of a planar representation of G. It describes besides the topology also the "shape", that is, the bends occurring in the edges and the angles inside the faces. Informally, an *orthogonal representation* is an equivalence class of the planar orthogonal drawings of G with similar shapes. In particular, two drawings of the same orthogonal representation have the same number of bend-points [DETT99].

There are two main approaches for orthogonal compaction: approaches based on network flows and approaches based on longest paths. We consider only flow-based compaction algorithms, since they allow us to assign weights, minimal and maximal capacity to the arcs of the corresponding flow network. By choosing appropriate parameters, we can set constraints such that the upward planarity and the port constraints are not violated by the compaction process.

The flow-based compaction algorithms are based on the main idea of two flow networks N_h and N_v that are constructed with respect to a given orthogonal representation. The network N_h is associated with the horizontal line segments and the network N_v is associated with the vertical line segments of the orthogonal representation. Thus, a line segment ϵ corresponds to an arc a of N_h or N_v. The given minimal length of ϵ corresponds to the lower capacity and the given maximal length of ϵ corresponds to the upper capacity of a. If no constraints are given, then the lower capacity of a is set to zero, the upper capacity set to ∞ and the cost is set to one. After computing a minimum cost flow for both network, the flow value of N_h and N_v corresponds to the width and height of the drawing, respectively. Moreover, the assigned flow value of a gives the length of ϵ and the sum of the flow value of the arcs in N_h and N_v gives the sum of total edge length of the drawing.

However, flow-based compaction have two main drawbacks: Firstly, the orthogonal representation must represent a *4-planar* graph, that is, G is planar and for each node v of G, the degree of v does not exceed four. As depicted in [DETT99], a planar but non-4-planar graph can be expanded to a 4-planar graph by replacing each node v with degree $d > 4$ by a cycle $C = \langle v_1, \ldots, v_d, v_1 \rangle$, where each node in C is incident to exactly one edge that was formally incident to v. The arcs of the cycle are constrained not to have any bends. It can be proven that C is aways drawn as a rectangle, thus it can represent the drawing of v.

Secondly, flow-based compaction algorithms presume, that all faces of the orthogonal representation have rectangular shapes. This goal can be achieved by adding additional artificial edges and nodes to the representation.

Although this artificial elements do not appear in the final drawing, they constitute additional constraints that can lead to unpleasant drawings. Due to this facts, Bridgeman et al. [BBD+00] suggested an approach which requires a so-called *turn-regular* orthogonal representation as input. The transformation of a representation to a turn-regular orthogonal representation requires less artificial nodes and edges than a transformation to a representation where each face has rectangular shape.

In [KKM01], Klau, Klein, and Mutzel suggested an improvement heuristic that directly operates on a given layout by exploiting the visibility property of the layout. Their improvement heuristic can be used for a pre-processing step after obtained an orthogonal drawing.

Applying compaction. Since compaction algorithms preserve the embedding, we have only to ensure that the achieved upward property and the port feasibility of the drawing will not be destroyed. For this intention, we first connect in $\mathcal{D}'$ the ports of each pair of consecutive layers L_i and L_{i+1} of the coarse layering $\mathcal{L}'$ by horizontal lines. Then we connect the most left port and the most right port on these layers by lines with two new bend-points and add two vertical lines connecting the bend-points such that these additional lines form a surrounding rectangular frame (see Figure 5.21(a)). Since there may be ports which are located on the top or on the bottom of a node, not all ports can be connected by the horizontal lines. In that case, a new node u is created as representative for each such port p and we assigned u to the position where the line segment incident to p cross the corresponding horizontal line segment of the rectangular frame. In a similar manner, the bend-points (which formally were long arc dummies) on L_i and L_{i+1} are handled.

We derive a drawing $\mathcal{D}_i$ of the subgraph between L_i and L_{i+1} where the regular nodes are omitted, instead, we have nodes represents the ports and the bend-points on the both layers. We extract an orthogonal representation OR_i from $\mathcal{D}_i$ and thereafter applying flow-based orthogonal compaction algorithms to OR_i. The flow networks allow us to assign cost, lower, and upper capacity to its arcs corresponding to the lengths of line segments of $\mathcal{D}_i$.

Let $\mathcal{S}_0$ denote the set of arcs corresponding to the line segments of $\mathcal{D}_i$ adjacent to two bend-points with a 90 and a 270 degree angle in a face. The arcs in $\mathcal{S}_0$ are assigned maximal cost and zero lower capacity. Therefore, we give high priority to the compaction algorithm for reducing the length of the line segments corresponding to the arcs of $\mathcal{S}_0$. The lower and upper capacity of the arcs corresponding to the line segments of the rectangular frame are set according to their lengths, hence the frame is fixed. Thus, the upward property and the port feasibility are preserved. The lower capacity of the remaining arcs is set to the minimum lengths of their corresponding line-segment in the initial drawing and a desired spacing, making sure that the initial drawing is a feasible solution for the compaction. Each corresponding line segment of an arc in $\mathcal{S}_0$ for which the compaction achieves zero flow value

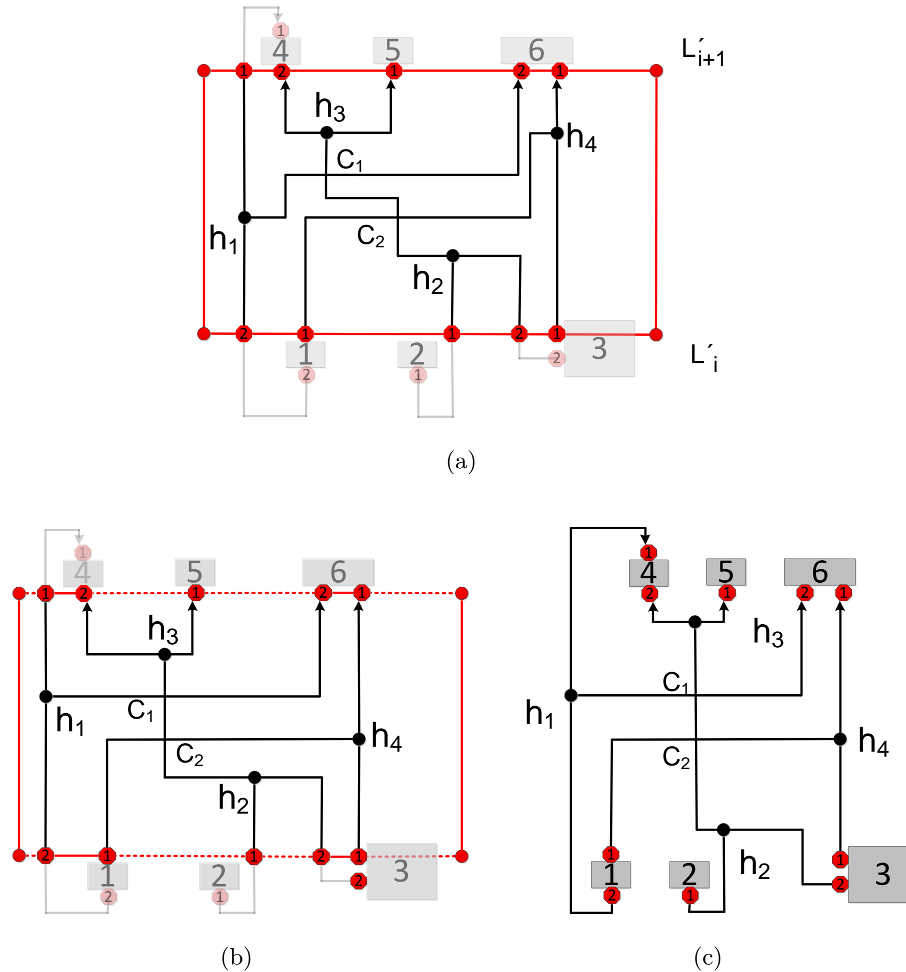


Figure 5.21: Orthogonal compaction: (a) Before starting the compaction, each drawing $\mathcal{D}_i$ (of the initial drawing $\mathcal{D}'$) of the subgraph between two consecutive layers is framed by a rectangle (red lines). The framed drawing contains no regular nodes, instead, it has nodes representing the ports and bend-points. The red lines are assigned a fixed length for compaction in order to fixe the relative position of the ports and the layers. (b) The drawing obtained after the first compaction round. We frame the drawing again for the final compaction. The red lines are assigned a fixed length, the dashed red lines are assigned only a minimal length for compaction. (c) The final drawing.

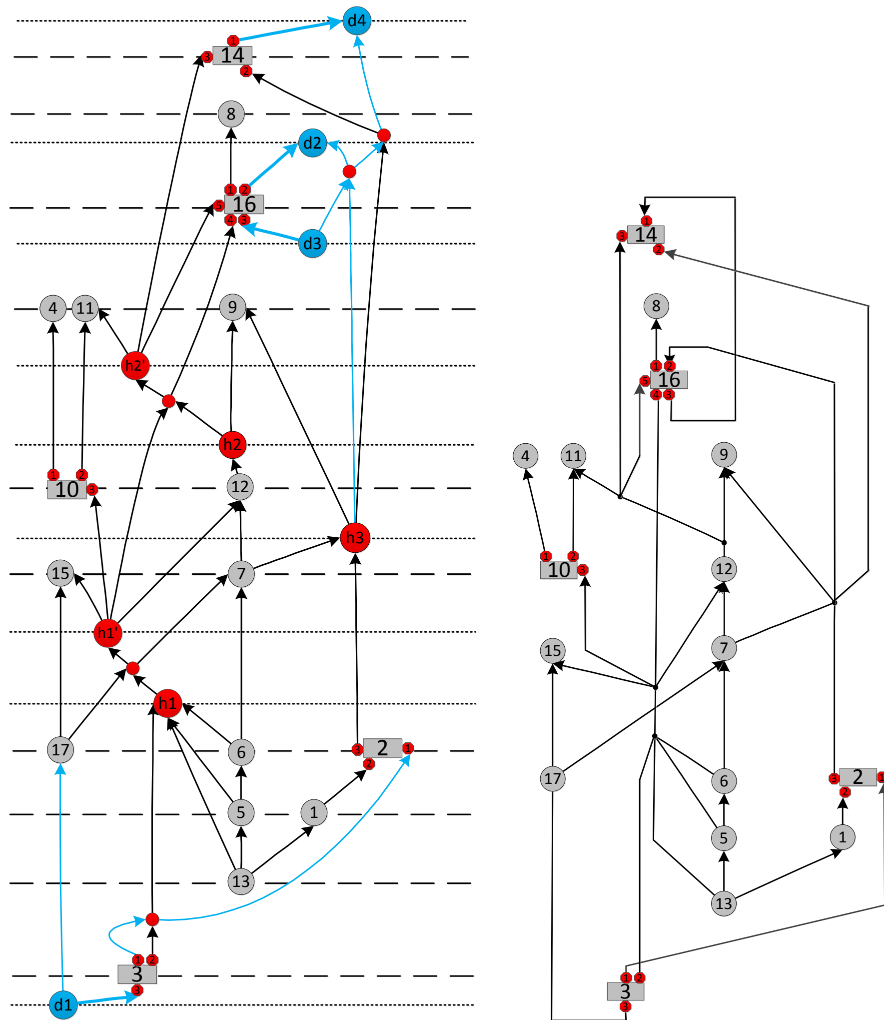
is then removed from the orthogonal representation by merging its adjacent segments.

After applying compaction to each drawings $\mathcal{D}_i$ of the subgraphs between each pair of consecutive layers, we can add an additional compaction step on the whole layout similar as before. But this time, it is applied not only to reduce the number of bend-points but also to improve the compactness of

the drawing. Notice that when considering the whole layout, we do not need to fix the x-coordinates; we only have to ensure that the horizontal distance between ports and the vertical distance of the layers of the coarse layering is fixed (see Figure 5.21(c)).

5.4 Example

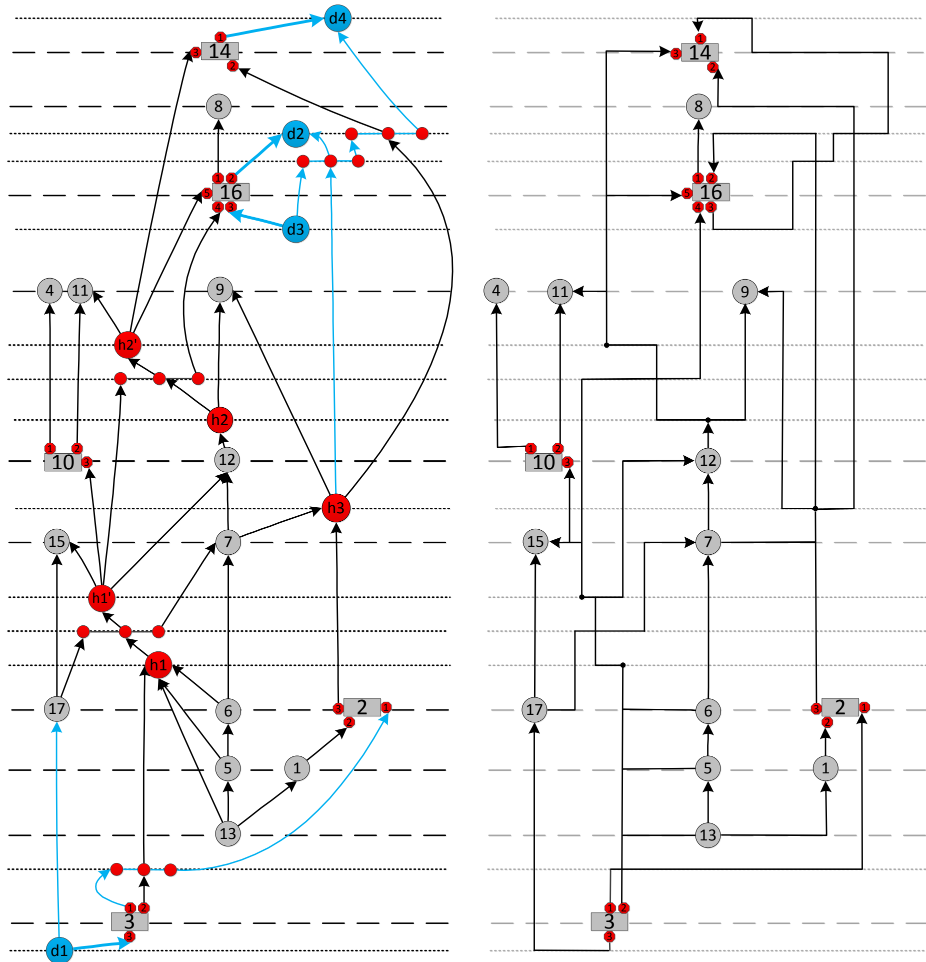
We continue the example of Section 4.6 and apply `UPL` to the UPR illustrated in Figure 4.38.



(a) The layering $\mathcal{L}$ of the final UPR U_4 (Figure 4.38) for hierarchical layout. For the sake of readability, the crossing dummies are drawn as small red cycles.

(b) The hierarchical upward drawing of the input hypergraph $\mathcal{H}$ with respect to $\mathcal{L}$. Observe that now all reversed arc are drawn according to their original direction.

Figure 5.22: (a) The coarse and fine layering of the regular nodes and hypernodes, respectively. Each node of the chains is assigned to a corresponding fine layer. The distances of the layers are dynamically computed.



(a) The layering $\mathcal{L}^*$ of the final UPR U_4 (Figure 4.38) for orthogonal layout.

(b) The initial orthogonal drawing with respect to $\mathcal{L}^*$.

Figure 5.23: (a) The coarse layering of the regular nodes and the fine layering of the hypernodes, nodes of the chains, and the crossing dummies. Unlike the layering $\mathcal{L}$, the crossing dummies are split here. (b) The arcs incident to a branching point may overlap each other. These arcs are merged together and new branching points are introduced. Then orthogonal representations for compaction are extracted from the initial drawing.

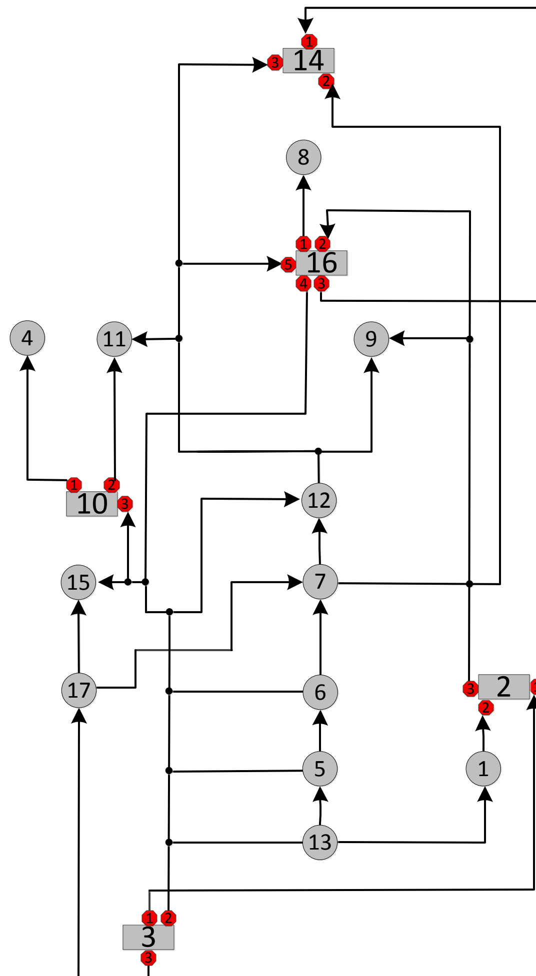


Figure 5.24: The final orthogonal drawing obtained after compaction has been applied.

5.5 Drawing Gallery

All drawings are computed using the following settings if not otherwise noted:

Sugiyama: `Barycenter+Greedy-Switch`, `GKNV-Layering`, `GKNV-Layout`

UPL: `LFUP50`, `UPL-Layering` basend `GKNV-Layering`, `GKNV-Layout`

The algorithms were conducted on: Windows XP, 2 GB, on an virtual machine (Parallels Desktop 6.0) on a Macbook Pro, Intel Core 2 Duo, 2.4GHz, 8 GB.

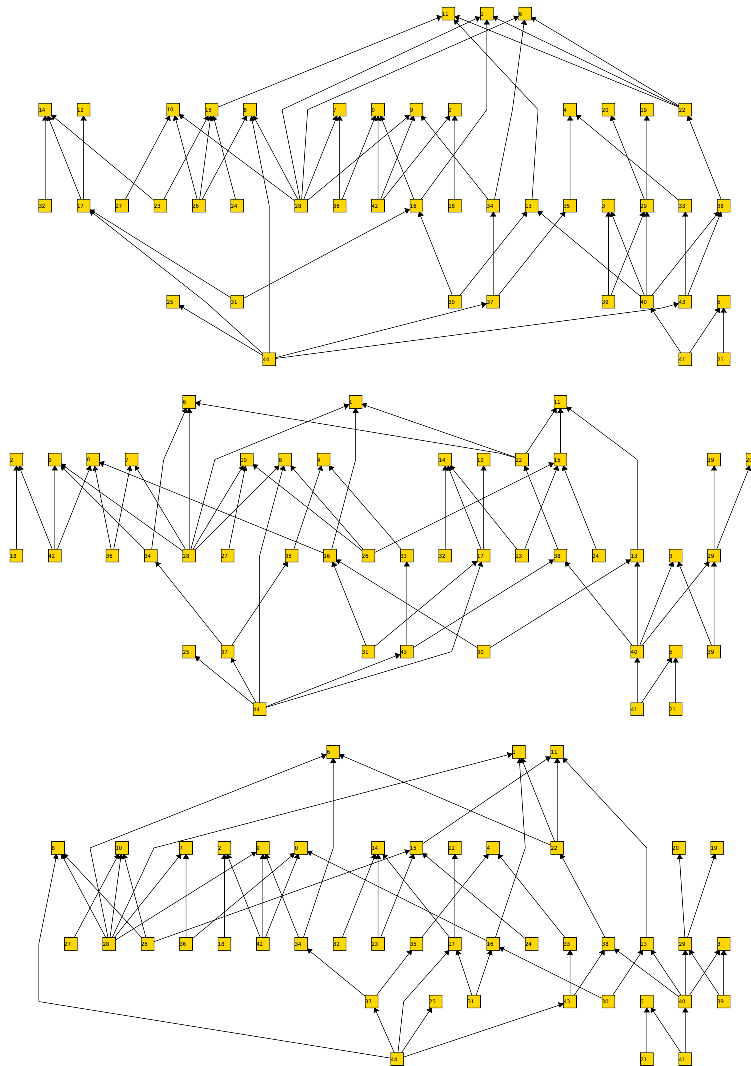


Figure 5.25: The effect of multiple random runs: The upward drawings of instance grafo3151.45 (Rome graphs) obtained from `Sugiyama`10, 20, 30, respectively.

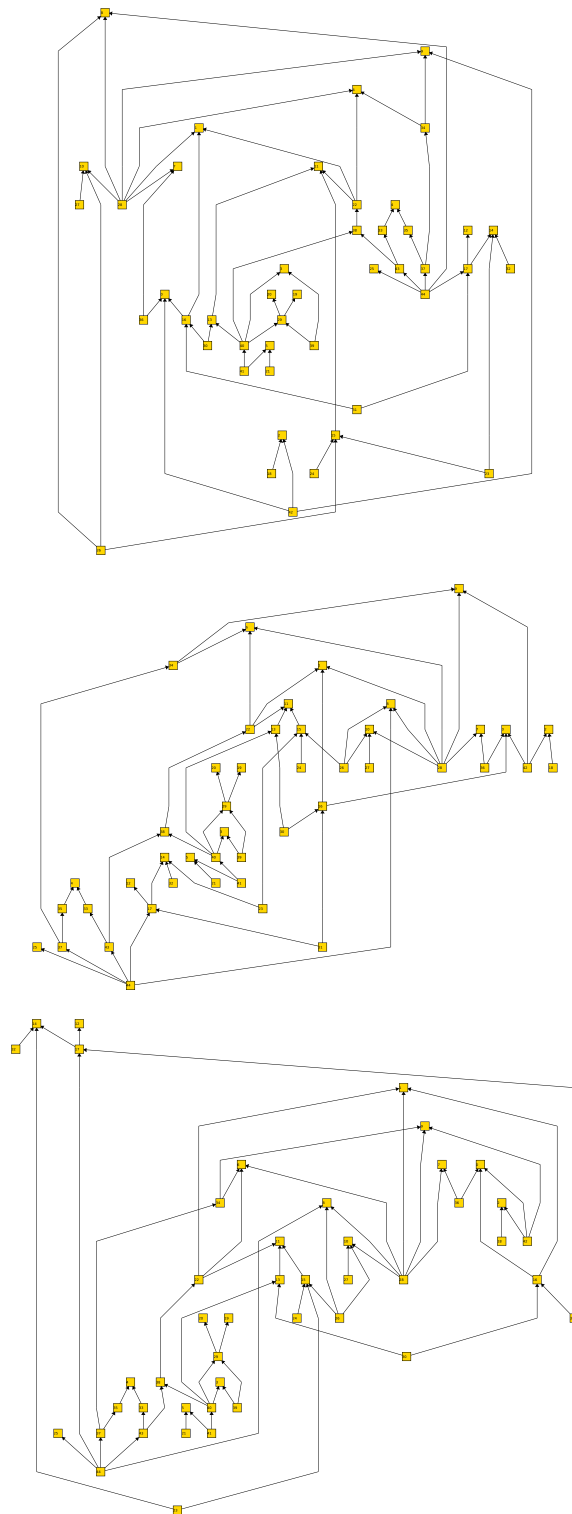
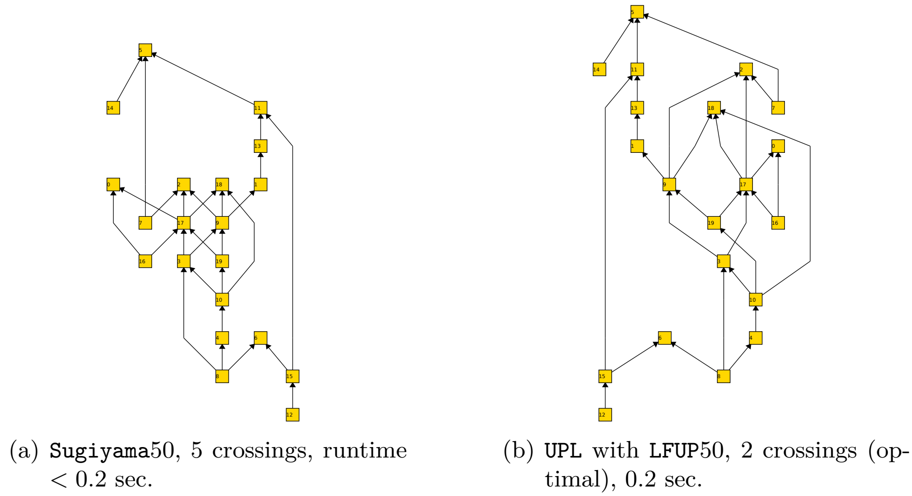


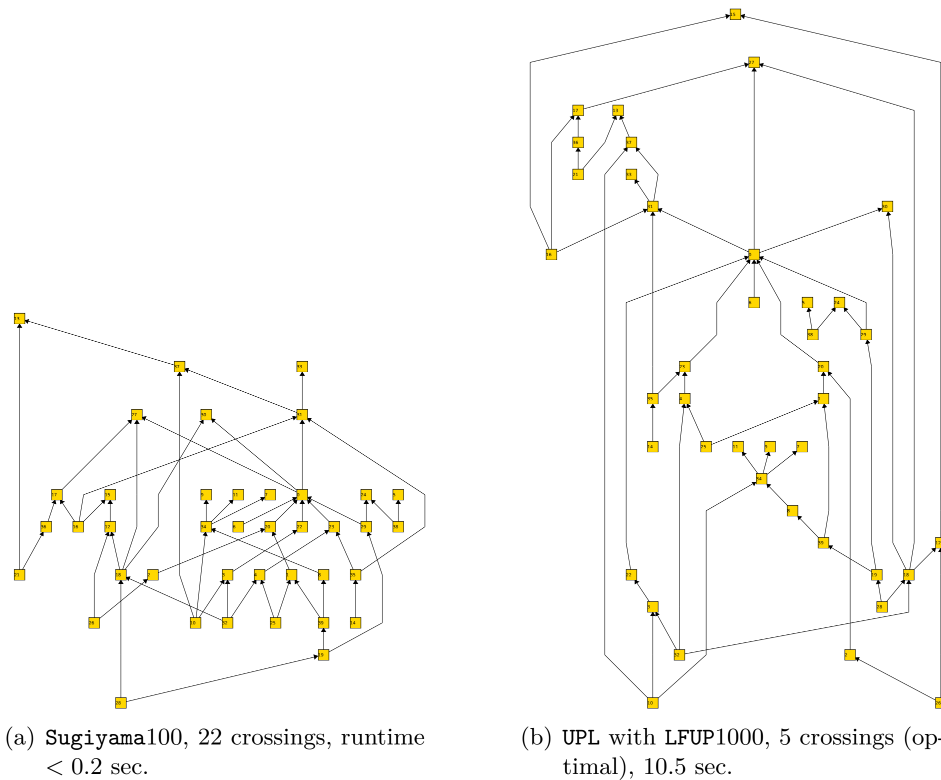
Figure 5.26: The effect of multiple random runs: The upward drawings of instance grafo3151.45 (Rome graphs) obtained from `UPL` based on `LFUP`10, 20, 30, respectively.



(a) `Sugiyama`50, 5 crossings, runtime < 0.2 sec.

(b) `UPL` with `LFUP`50, 2 crossings (optimal), 0.2 sec.

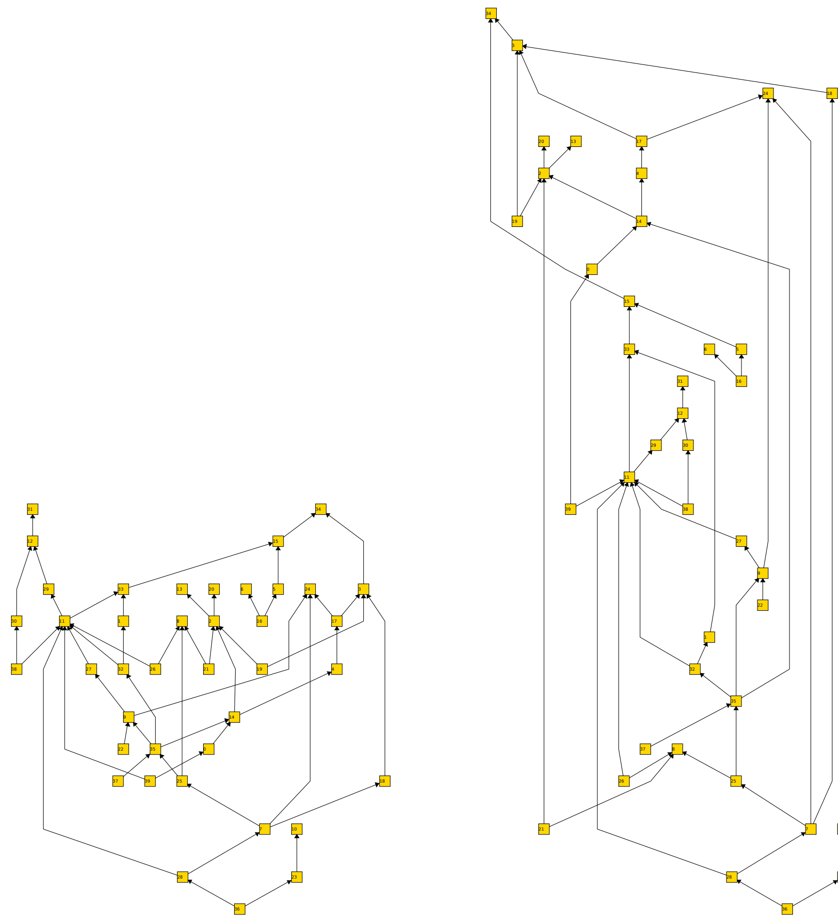
Figure 5.27: DAGs with known $ucr(G)$: Instance g2670.20.02 with $ucr(G) = 2$.



(a) `Sugiyama`100, 22 crossings, runtime < 0.2 sec.

(b) `UPL` with `LFUP`1000, 5 crossings (optimal), 10.5 sec.

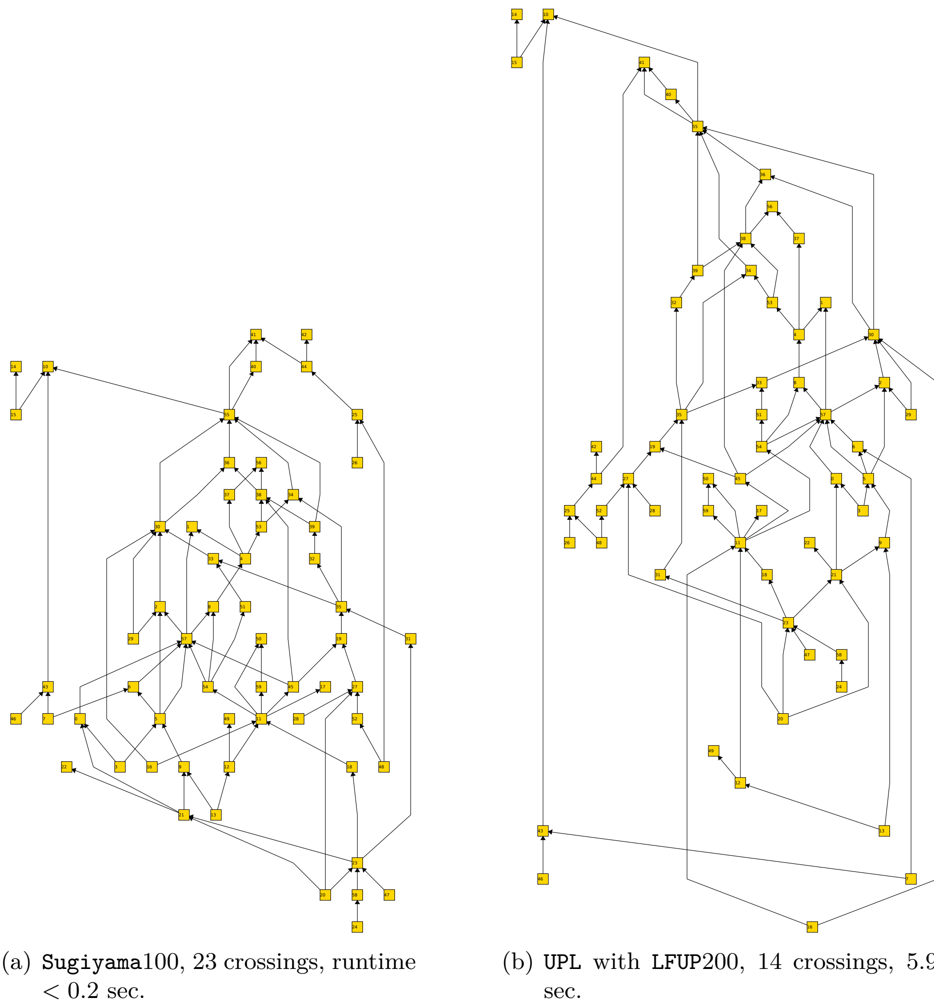
Figure 5.28: DAGs with known $ucr(G)$: Instance g11498.40.05 with $ucr(G) = 5$.



(a) Sugiyama50, 12 crossings, runtime < 0.2 sec.

(b) UPL with LFUP50, 5 crossings, 0.5 sec.

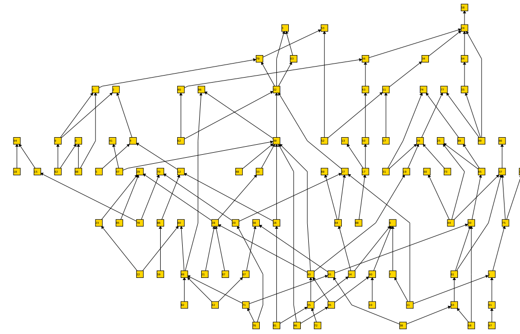
Figure 5.29: DAGs with known $ucr(G)$: Instance g10054.40.04 with $ucr(G) = 4$.



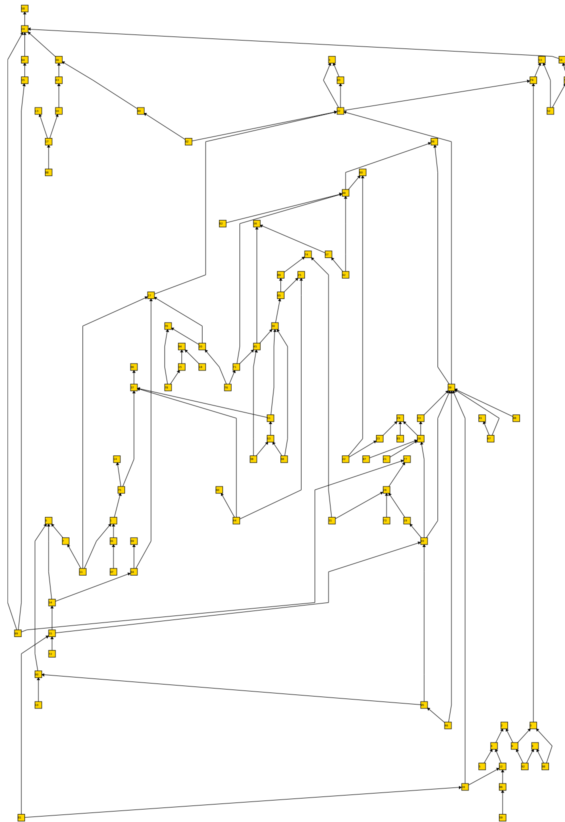
(a) `Sugiyama`100, 23 crossings, runtime < 0.2 sec.

(b) `UPL` with `LFUP`200, 14 crossings, 5.9 sec.

Figure 5.30: DAGs with known $ucr(G)$: Instance g4335.60.12 with $ucr(G) = 12$.

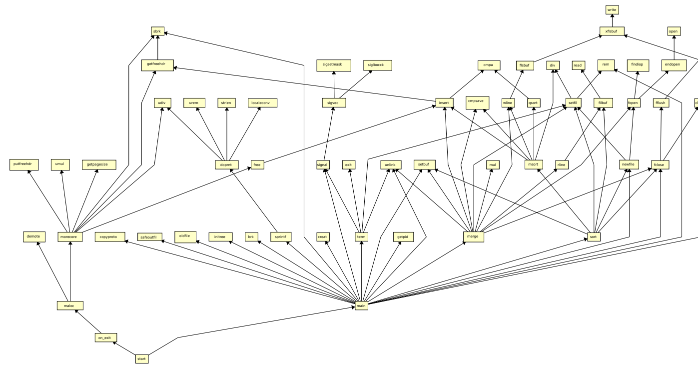


(a) `Sugiyama`100, 62 crossings, runtime < 0.2 sec.

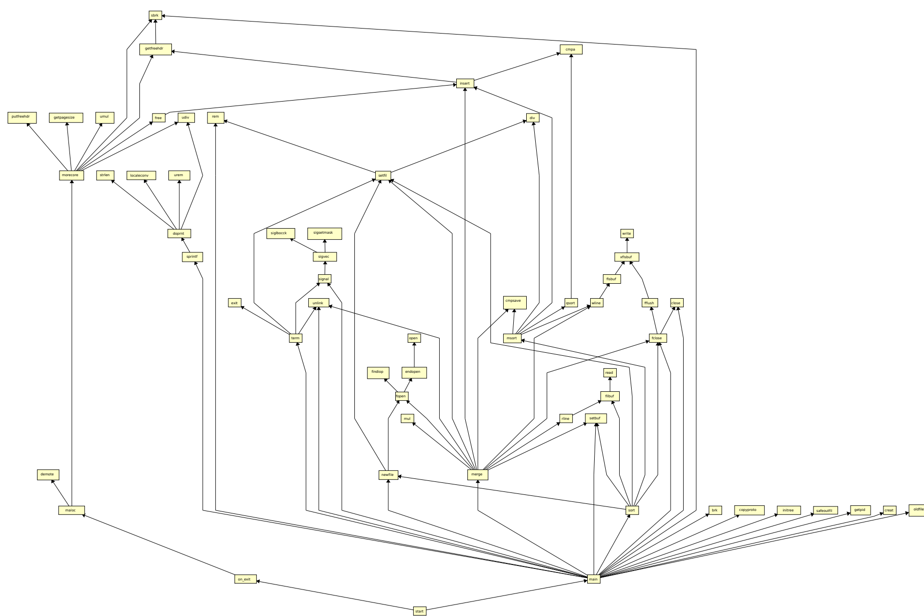


(b) `UPL` with `LFUP`500, 7 crossings, 33 sec.

Figure 5.31: DAGs with known $ucr(G)$: Instance g10106.100.03 with $ucr(G) = 3$.

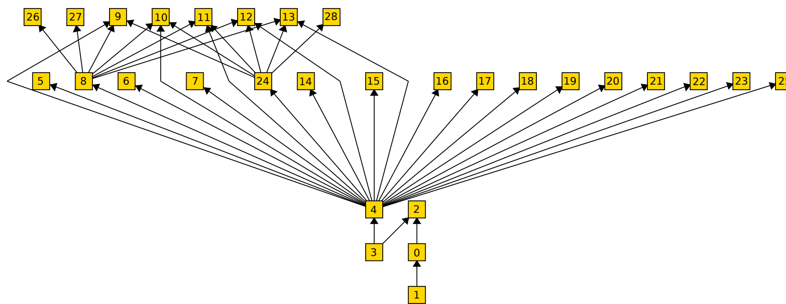


(a) Sugiyama50, 54 crossings

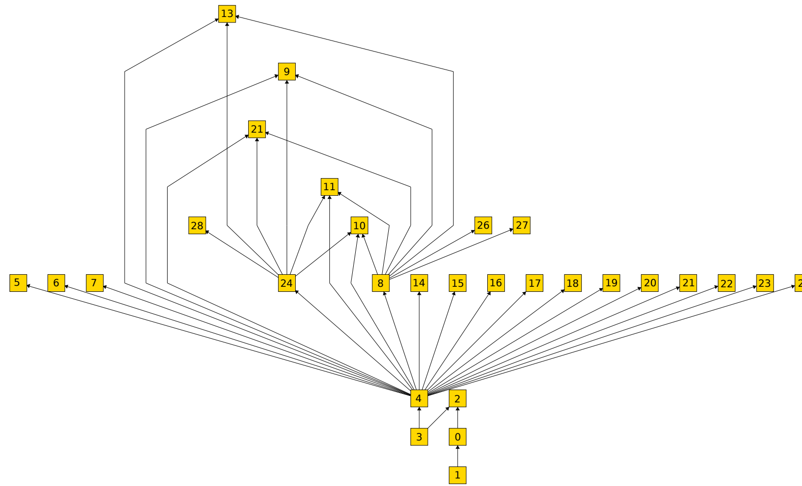


(b) UPL20, 11 crossings

Figure 5.32: Drawing of the graph "*profile*" (from the webpage www.graphviz.org). The original graph is not connected and consists of two connected components. The small component consists of two nodes and one arc is omitted, since the current OGDF UPL implementation does not support unconnected digraphs yet.

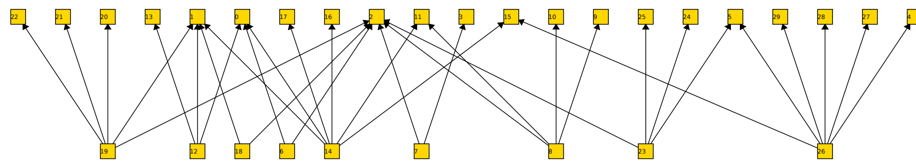


(a) `Sugiyama`20, 24 crossings. The arc crossings and the packed drawings of the subgraph on layer 4 and 5 reduce the readability.

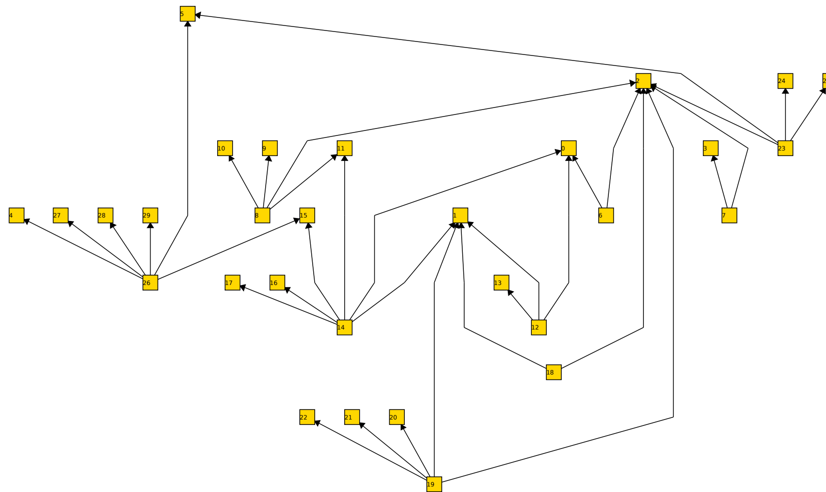


(b) `UPL`20, 4 crossings

Figure 5.33: North DAGs: Drawing of instance *g.29.16*.

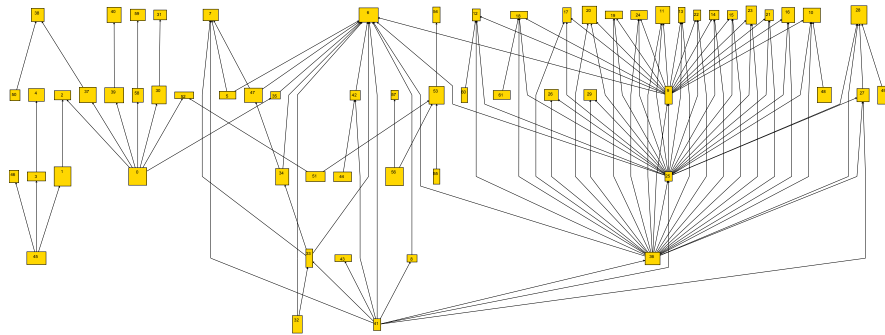


(a) `Sugiyama`20, 41 crossings.

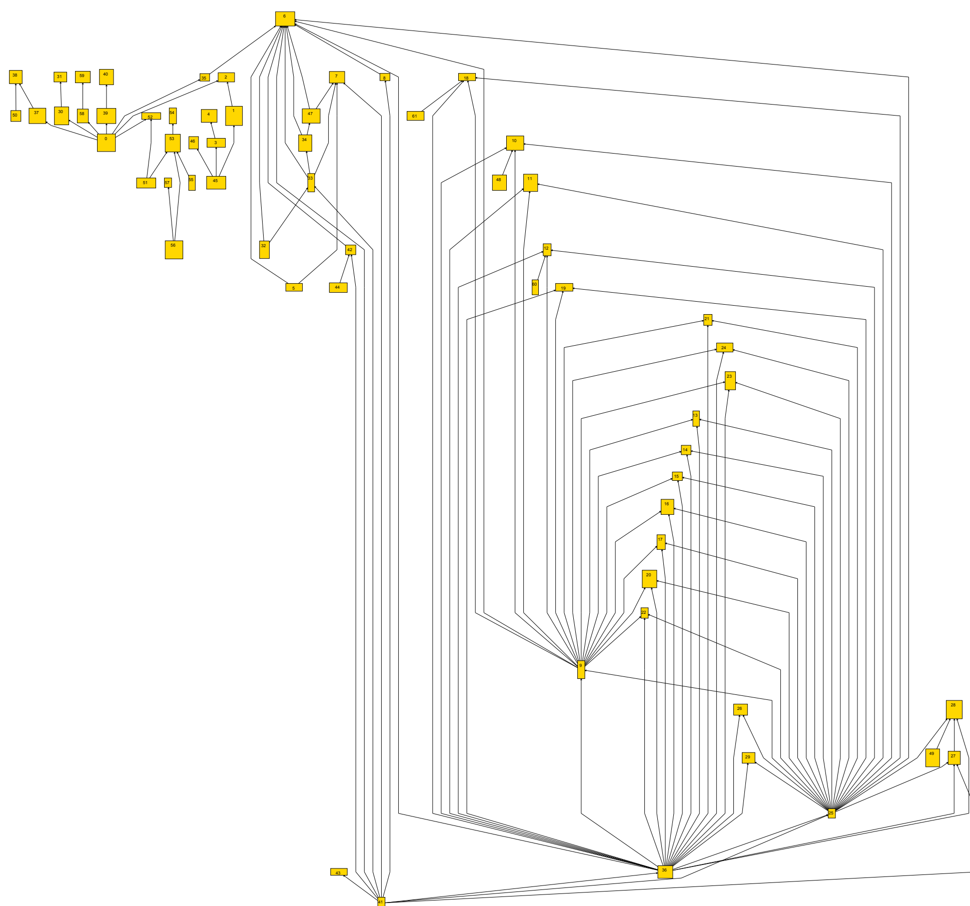


(b) `UPL`20, no crossings

Figure 5.34: North DAGs: A drawing of the upward planar instance g.30.8. Due to its moderate size and its upward planarity, this DAG can be considered as easy to draw. Yet, the drawing of `Sugiyama` (a) is very unsatisfying, no characteristics of the input DAG can be detected. In contrast to (a), the drawing in (b) is well structured and reflects the upward planar property of the DAG.

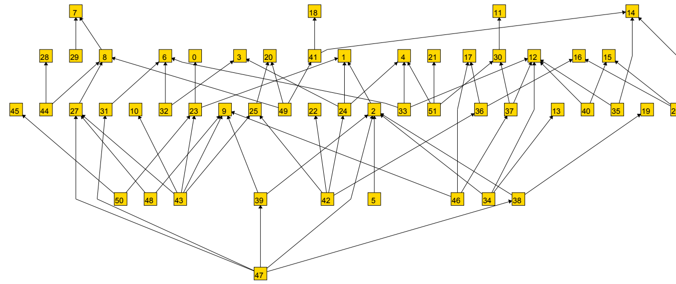


(a) `Sugiyama`, 250 crossings.

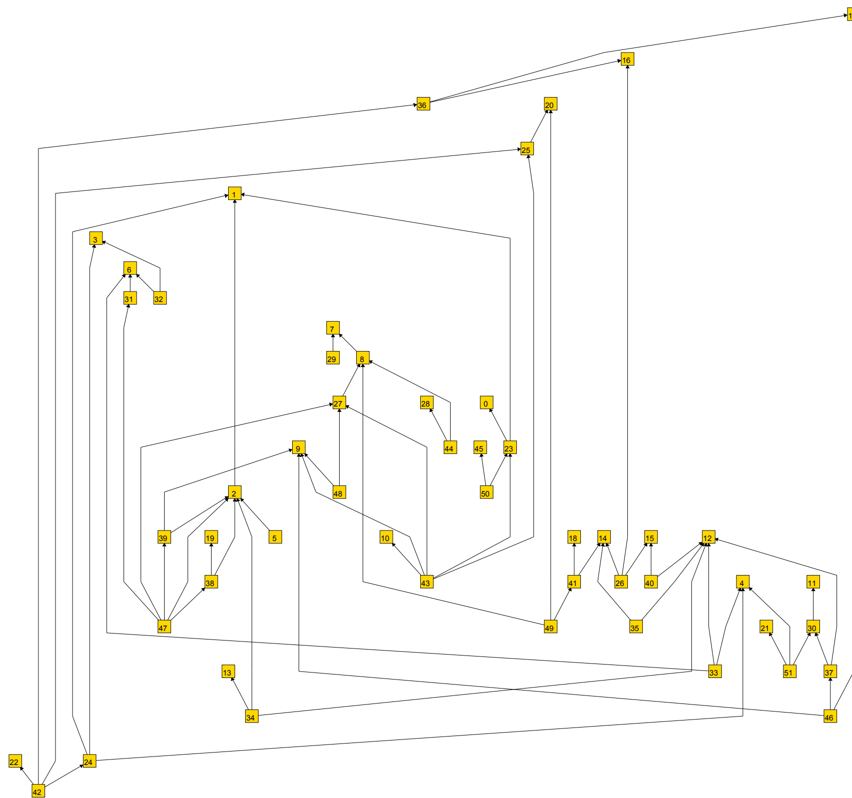


(b) `UPL`, 82 crossings

Figure 5.35: North DAGs: Drawing of the graph instance *g.60.0* with random node size.

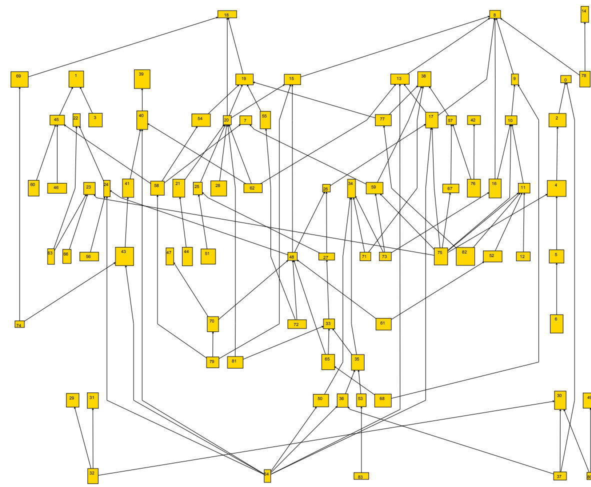


(a) `Sugiyama`20, 76 crossings

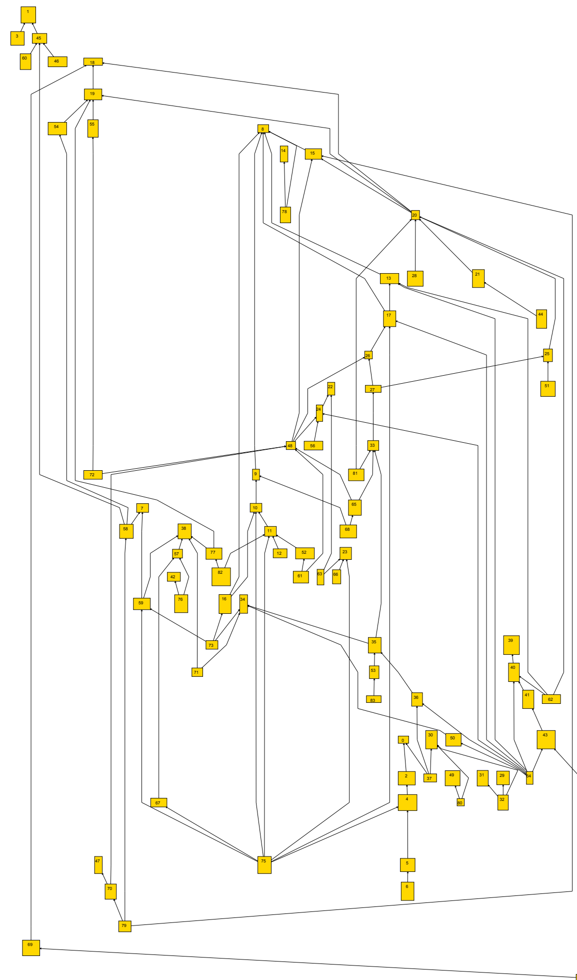


(b) `UPL`20, 9 crossings

Figure 5.36: Rome graphs. Drawing of the graph instance *grafo3579.52*. The drawing of (a) represent the typical weaknesses of `Sugiyama`. The arcs are routed in a criss-crossing fashion and the drawing is unnaturally flat, unstructured, and has many arc crossings.



(a) `Sugiyama`20, 134 crossings



(b) `UPL`20, 41 crossings

Figure 5.37: Rome graphs. Drawing of the graph instance *grafo8087.84*.

Chapter 6

Discussion

6.1 Conclusion

The main goal of this thesis was to develop a new drawing algorithm for digraphs and directed hypergraphs based on the idea of upward planarization to overcome the drawbacks of the existing approaches. In order to achieve this goal, we tackled the following two challenges:

1. Develop a new upward planarization approach which can exploit the potential of the upward planarization idea.

2. Develop a new layout approach tailored to the construction of upward drawings based on a given upward planar representation.

We gave the motivations for the first challenge by illustrating the weakness of the classical layered approach. We also showed that the mixed upward planarization approach does not exploit the potential of the idea of upward planarization, since its upward planar subgraph computation is based on a simple idea and inserting the arcs by utilizing layering techniques may limit the possible insertion paths. Therefore our goal was to develop an upward planarization approach with sophisticated techniques for the subgraph computation and without any layering techniques for the arc reinsertion step. We identified the problems that arise on the way toward the new upward planarization approach; as we illustrated that in contrast to planarization, the subgraph cannot be straight-forwardly computed (feasible subgraph problem) and a deleted arc cannot be reinserted without taking the remaining deleted arcs into account (feasible arc reinsertion problem). We tackled these two problems by modeling the hierarchies of the input DAGs by an auxiliary graph called merge graph. Using this auxiliary graph, the feasible subgraph problem can be reduced to a simple cycle test. The arc reinsertion problem was solved by a routing network using the idea of static and dynamic locking of arcs, that is, we do not allow certain arcs to be crossed.

The performance of the new layer-free upward planarization approach (`LFUP`) was evaluated by experiments based on real-world and artificial DAGs. Our experiments showed that our new approach outperforms the layered upward crossing minimization heuristics regardless of the techniques used for solving the k-level crossing minimization problem. This also included the case when the k-level crossing minimization problem was optimally solved. It also outperforms the mixed upward planarization (`MUP`) and the `Grid-Siftig` approach, which does not strictly follow the idea of the layered upward crossing minimization approach, that is, `Grid-Siftig` can modify the given layering by introducing new layers. Furthermore, the evaluation based on DAGs with known upward crossing numbers revealed that the solutions of `LFUP` for small and mid-size instances are optimal or nearly optimal. Due to these facts, we conclude:

> The layer-free upward planarization approach is a state-of-the-art upward crossing minimization heuristic. On the considered benchmark sets, it outperforms all existing upward crossing minimization heuristics.

We also gave an extension of `LFUP` for upward planarizing directed hypergraphs and for dealing with given port constraints. We combined our ideas of upward arc insertion and the known idea of inserting a single edge in the undirected minor crossing number scenario with a novel heuristic method to insert a hyperarc in a confluent fashion. Our approach not only solves the upward planarization problem for DAGs and directed hypergraphs with and without port constraints, but it is also the first port-constraint-aware upward planarization approach for these classes of graphs.

Regarding the second goal—the problem of realizing upward planar representations (UPR), that is, constructing upward drawings based on a given UPR $\mathcal{R}$—the published layout algorithms for step three of Sugiyama's framework offer good and sophisticated solutions for the coordinate assignment phase. Due to this fact, we decided to develop a new layout approach for realizing UPRs by adopting these existing algorithms. This would allow us to benefit from both, the advantage of upward planarization and the well known and elaborated layout techniques of Sugiyamas step three. In order to apply the layout algorithms, we had to derive a layering from the UPR. In our fist approach `UPSugiyama`, a layering $\mathcal{L}$ was straight-forwardly derived from $\mathcal{R}$ and we especially developed a new technique for determining the node order on each layer of $\mathcal{L}$ with respect to $\mathcal{R}$. Although `UPSugiyama` produces much better upward drawings than the alternative upward drawing approaches `Dominance` and `Visibility`, the results are still unsatisfying; in particular the drawing area requirements are too high. This flaw is due to the handling of dummy nodes, which inevitably results in larger drawing area requirements and hence, it is also symptomatic for the alternative upward drawing approaches. We tackled this problem by mapping the hierarchies

induced by the UPR into an auxiliary graph H. This auxiliary graph allows us to ignore the crossing dummies when deriving a layering of an input graph with respect to H, hence leading to a more compact drawing. We also utilized existing and new techniques for incorporating port constraints into a layering. Furthermore, we described how to deal with a UPR of a directed hypergraph; hence our new layout approach called upward planarization layout (`UPL`) can draw both digraphs and directed hypergraphs (with and without given port constraints). For further improving the quality of the final drawing, we also introduced techniques for the long arc dummy reduction and for arc bending.

We evaluated `UPL` extensively regarding runtime, several common aesthetic criteria, and by manually inspecting hundreds of drawings where we focused on the clarity, the reflected structures of the input digraphs, and tidiness of the drawings. These properties were summarized under the criterion *impression*.

Based on the results of the evaluation, we concluded that `UPL` should be preferred over the other considered algorithms, including `Sugiyama`, even when the drawing area requirement increases with increasing size of the instances. We summarized the impression of the drawings as follows:

> The drawings produced by `Sugiyama` are unnaturally flat with many arcs routed criss-crossing between the layers. Often, it seems that the nodes are arbitrary or randomly assigned to the layers. In contrast to `Sugiyama`, the upward drawings produced by `UPL` offer clarity, better reflect the structures of the digraphs, are tidy, and have relatively few arc crossings.

We also developed an approach for realizing UPRs in orthogonal style. We combined existing ideas for orthogonal arc routing with our new layering technique for obtaining an initial orthogonal drawing. Furthermore, we introduced an approach for extracting orthogonal representations from the initial drawing such that we could adopt existing compaction algorithms without violating the achieved results, that is, the number of arc crossings, the upward property, and the fulfilled port constraints. The orthogonal layout approach is the first approach in hierarchical drawing of digraphs where compaction techniques were used. In addition, by applying compaction, we addressed the problem of large drawing area requirements for large instances of the hierarchical (non-orthogonal) layout approach.

The algorithms `LFUP` and `UPL` were implemented as part of the open-source graph drawing framework OGDF. The author hopes that the access to the source code will increase the acceptance of the new drawing technique and allows practitioners to investigate the quality of the drawing algorithm by themself. The author also hopes, that `LFUP` and `UPL` will convince many practitioners to use them and inspire researchers for further investigation regarding this very interesting topic.

6.2 Future Works

We have suggested a new drawing algorithm for visualizing digraphs based on novel ideas. Although it computes very pleasant upward drawings, there are still many challenges left. Here, we briefly depict some open problems.

Maximum FUPS: A maximum FUPS U of an sT-graph G is a FUPS such that the number of arcs of U is the highest among all FUPS of G. Following the idea that if few arcs have to be reinserted, then few arc crossings should arise in the final drawing, computing a maximum FUPS sould lead to a further reduction of arc crossings.

Constraint feasible minimal UIP: We proved that we can find a minimum feasible UIP in polynomial time. Although our experiments showed that nearly always the minimum feasible UIP is also constraint feasible, no non-trivial algorithm for computing a minimal constraint feasible UIP is known.

Arc reinsertion: As reported by Gutwenger and Mutzel [GM04], there are different edge reinserting strategies which can lead to further reduction of the edge crossings. The question arises here: Can we achieve similar improvements for upward planarization of DAGs, when we adapt the reinserting strategies suggested in [GM04]?

Upward crossing number $ucr(G)$: The upward crossing number $ucr(G)$ of a DAG G is not only of theoretical interest, but knowing the $ucr(G)$ of a DAG can also help to evaluate the quality of the existing upward crossing minimization heuristics. However, only trivial exponential algorithms are known for computing $ucr(G)$.

Height of a layering: As shown by our experiments, there is a gap between Sugiyama's and the `UPL` approach regarding the height of the drawings, that is, the number of layers of the corresponding layering. We could prove that the layering is optimal with respect to the given UPR $\mathcal{R}$, if $\mathcal{R}$ represents an upward planarized sT-graph (see Lemma 19), but it is unknown how this results can be extended to general digraphs.

Bend minimization: Given a layering $\mathcal{L}$ with respect to a UPR, we are interested in a realization $\mathcal{D}$ of $\mathcal{L}$ such that the number of bends occurring in $\mathcal{D}$ is minimized (also see Definition 17). Such a drawing can improve the readability.

Compaction: As shown by the experimental evaluations in Section 5.3.4, the area requirement of the hierarchical upward drawings produced by `UPL` is relatively high, especially for large instances. A compaction approach

for upward drawing in orthogonal style (Section 5.3.5) was already introduced, but an approach for the polyline hierarchical upward drawing is still missing.

Bibliography

[BBBH10] C. Bachmaier, F.-J. Brandenburg, W. Brunner, and F. Hübner. A global k-level crossing reduction algorithm. In *WALCOM*, pages 70–81, 2010.

[BBD+00] S. S. Bridgeman, G. Di Battista, Walter Didimo, G. Liotta, R. Tamassia, and L. Vismara. Turn-regularity and optimal area drawings of orthogonal representations. *Comput. Geom.*, 16(1):53–93, 2000.

[BBG11] C. Bachmaier, W. Brunner, and A. Gleißner. Grid sifting: Leveling and crossing reduction. Technical report, University of Passau, 2011.

[BDMT98] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM J. Comput.*, 27(1):132–169, 1998.

[BFM06] D. Bokal, G. Fijavz, and B. Mohar. The minor crossing number. *SIAM J. Discrete Math.*, 20:344–356, 2006.

[BJL99] C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. In *Proc. Graph Drawing '00*, volume 1984 of *LNCS*, pages 229–240. Springer, 1999.

[BJM02] W. Barth, M. Jünger, and P. Mutzel. Simple and efficient bilayer cross counting. In *Graph Drawing*, pages 130–141, 2002.

[BK02] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In *Proc. Graph Drawing '01*, pages 31–44, London, UK, 2002. Springer-Verlag.

[BLME02] J. Branke, S. Leppert, M. Middendorf, and P. Eades. Width-restricted layering of acyclic digraphs with consideration of dummy nodes. *Inf. Process. Lett.*, 81(2):59–63, 2002.

[BN88] G. Di Battista and E. Nardelli. Hierarchies and planarity theory. *Systems, Man and Cybernetics, IEEE Transactions on*, Issue:6:1035 – 1046, Nov/Dec 1988.

[BPTT89] G. Di Battista, E. Pietrosanti, R. Tamassia, and I.G. Tollis. Automatic layout of PERT diagrams with X-PERT. In *Proc. IEEE Workshop on Visual Languages*, pages 171–176, 1989.

[BT88] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theor. Comput. Sci.*, 61:175–198, 1988.

[BTT84] C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity relationship diagrams. *J. Syst. Software*, 4:163–173, 1984.

[BWZ10] C. Buchheim, A. Wiegele, and L. Zheng. Exact Algorithms for the Quadratic Linear Ordering Problem. *INFORMS JOURNAL ON COMPUTING*, 22(1):168–177, 2010.

[CG72] E. Coffman and R. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[CG07] M. Chimani and C. Gutwenger. Algorithms for the hypergraph and the minor crossing number problems. In *Proc. ISAAC'07*, volume 4835 of *LNCS*, pages 184–195. Springer-Verlag, 2007.

[CGM+10] M. Chimani, C. Gutwenger, P. Mutzel, M. Spönemann, and H.-M. Wong. Crossing minimization and layouts of directed hypergraphs with port constraints. In *Graph Drawing*, pages 141–152, 2010.

[CGMW08] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Layer-free upward crossing minimization. In *WEA 2008: Workshop on Experimental Algorithms*, volume 5038 of *LNCS*, pages 55–68. Springer, 2008.

[CGMW10a] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Layer-free upward crossing minimization. *ACM Journal of Experimental Algorithmics*, 15, 2010.

[CGMW10b] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Upward planarization layout. In *Proc. Graph Drawing '09*, volume 5849 of *LNCS*, pages 94–106. Springer, 2010.

[CGMW11] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Upward planarization layout. *Journal of Graph Algorithms and Applications*, 15(1):127–155, 2011.

[Chi08] M. Chimani. *Computing Crossing Numbers.* PhD thesis, Technische Universität Dortmund, 2008.

[CHJM11] M. Chimani, P. Hungerländer, M. Jünger, and P. Mutzel. An sdp approach to multi-level crossing minimization. *Workshop on Algorithm Engineering and Experiments 2011 (ALENEX11)*, 2011.

[DDGS08] U. Dogrusoz, C. A. Duncan, C. Gutwenger, and G. Sander. Graph drawing contest report. In *Proc. Graph Drawing '08*, LNCS, 2008.

[DEGM03] M. T. Dickerson, D. Eppstein, M. T. Goodrich, and J. Yu Meng. Confluent drawings: visualizing non-planar diagrams in a planar way. In *Proc. 11th Int. Symp. Graph Drawing (GD 2003)*, number 2912 in Lecture Notes in Computer Science, pages 1–12. Springer-Verlag, September 2003.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[DGL+97] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7(5-6):303–325, 1997.

[DGL+00] G. Di Battista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. Drawing directed acyclic graphs: An experimental study. *Int. J. Comput. Geom. Appl.*, 10(6):623–648, 2000.

[Die05] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[DTT92] G. Di Battista, R. Tamassia, and I. G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete Comput. Geom.*, 7(4):381–401, 1992.

[EFK00] M. Eiglsperger, U. Fößmeier, and M. Kaufmann. Orthogonal graph drawing with constraints. In *Proc. SODA '00*, pages 3–11. SIAM, 2000.

[EGB06] T. Eschbach, Wolfgang Guenther, and B. Becker. Orthogonal hypergraph drawing for improved visibility. *J. Graph Algorithms Appl.*, 10(2):141–157, 2006.

[EK86] P. Eades and D. Kelly. Heuristics for drawing 2-layered networks. *Ars Combinatoria*, 21A:89–98, 1986.

[EKE03] M. Eiglsperger, M. Kaufmann, and F. Eppinger. An approach for mixed upward planarization. *J. Graph Algorithms Appl.*, 7(2):203–220, 2003.

[ELS89] P. Eades, X. Lin, and W. F. Smyth. Heuristics for the feedback arc set problem. Technical Report 1, Curtin University of Technology, Perth, Australia, 1989.

[ELS93] P. Eades, Xuemin Lin, and W. F. Smyth. A fast effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47:319–323, 1993.

[ES90] P. Eades and K. Sugiyama. How to draw a directed graph. *J. Inf. Process.*, 13(4):424–437, 1990.

[EW86] P. Eades and N.C. Wormald. The median heuristic for drawing two-layered networks. Technical Report 69, University Queensland, 1986.

[FBB+05] M. Forster, C. Bachmaier, F.-J. Brandenburg, Marcus Raitner, and Paul Holleis. Gravisto: Graph visualization toolkit. In J. Pach, editor, *Proc. Graph Drawing, GD 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 502–503. Springer, 2005.

[GA94] O. Goldschmidt and A.Takvorian. An efficient graph planarization two-phase heuristic. *Networks*, 24:69–73, 1994.

[GJ83] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM J. Algebraic and Discrete Methods*, pages 312–316, 1983.

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[GJR85] M. Grötschel, M. Jünger, and G. Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 33:28–42, 1985. 10.1007/BF01582009.

[GKNV93] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Pract. Exper.*, 19(3):214–229, 1993.

[GM04] C. Gutwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In *Proc. Graph Drawing 03*, volume 2912 of *LNCS*, pages 13–24, 2004.

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. DAG - a programm that draws directed graphs. *Software Practice and Experiments*, 18(11):1047–1062, 1988.

[GSM10] G. Gange, P. J. Stuckey, and K. Marriott. Optimal k-level planarization and crossing minimization. In *Graph Drawing 2010*, 2010.

[GT95] A. Garg and R. Tamassia. Upward planarity testing. *Order: A Journal on the Theory of Ordered Sets and Its Applications*, 12:109–133, 1995.

[GT01] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2001.

[HK99a] P. Healy and A. Kuusik. The vertex-exchange graph: A new concept for multi-level crossing minimisation. In *Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 205–216. Springer Berlin / Heidelberg, 1999.

[HK99b] P. Healy and A. Kuusik. The vertex-exchange graph and its use in multi-level graph layout. In *In Kratochvil [Kra99*, pages 205–216, 1999.

[HL96] M. D. Hutton and A. Lubiw. Upward planar drawing of single-source acyclic digraphs. *SIAM J. Comput.*, 25(2):291–311, 1996.

[HN02a] P. Healy and N. S. Nikolov. A branch-and-cut approach to the directed acyclic graph layering problem. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 98–109, London, UK, 2002. Springer-Verlag.

[HN02b] P. Healy and N. S. Nikolov. How to layer a directed acyclic graph. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, pages 16–30, London, UK, 2002. Springer-Verlag.

[HR10] P. Hungerländer and F. Rendl. Semidefinite relaxations of ordering problems. Technical report, Alpen-Adria Iniversiät Klagenfurt Institut für Mathematik Austria, August 2010.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[JLM98] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time (extended abstract). In S. H. Whitesides, editor,

Proc. 6th International Symposium on Graph Drawing '98, volume 1547 of *Lecture Notes in Computer Science*, pages 224–237. Springer, 1998.

[JLM99] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. Technical report, Angewandte Mathematik und Informatik, Universität zu Köln, 1999.

[JLMO97] M. Jünger, E. Lee, P. Mutzel, and T. Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In G. DiBattista, editor, *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 13–24. Springer Berlin / Heidelberg, 1997.

[JM96] M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 162:33–59, 1996.

[JM97] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:1–25, 1997.

[Kel87] D. Kelly. Fundamentals of planar ordered sets. *Discrete Math.*, 63(2-3):197–216, 1987.

[KKM01] G. W. Klau, K. Klein, and P. Mutzel. An experimental comparison of orthogonal compaction algorithms. In *Proc. Graph Drawing '00*, pages 37–51, London, UK, 2001. Springer-Verlag.

[KW01] M. Kaufmann and D. Wagner, editors. *Drawing graphs: methods and models.* Springer-Verlag, London, UK, 2001.

[Lin95] X. Lin. *Analysis of Algorithms for Drawing Graphs.* PhD thesis, University of Queensland, 1995.

[LS77] S. Lam and R. Sethi. Worst case analysis of two scheduling algorithms. *SIAM J. Computing*, 6:518–536, 1977.

[Luc76] C.L. Lucchesi. *A minimax equality for directed graphs.* PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1976.

[Mäk90] E. Mäkinen. Experiments in drawing two-level hierarchical graphs. *Intern. J. Computer Math.*, 37:129–135, 1990.

[Meh84] K. Mehlhorn. *Graph algorithms and NP-completeness.* Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[MSLN+09] H. Mi, F. Schreiber, N. Le Novére, S. Moodie, and A. Sorokin. Systems Biology Graphical Notation: Activity Flow language Level 1. *Nature Precedings*, (713), September 2009.

[MSM99] C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for k-layer straightline crossing minimization. In J. Kratochvíyl, editor, *Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 217–224. Springer Berlin / Heidelberg, 1999.

[Mut97] P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. In *SIAM J. Optimization*, pages 318–333. Springer Verlag, 1997.

[NHM+09] N. L. Novere, M. Hucka, H. Mi, S. Moodie, F. Schreiber, A. Sorokin, E. Demir, K. Wegner, M. I. Aladjem, S. M. Wimalaratne, F. T. Bergman, R. Gauges, P. Ghazal, H. Kawaji, L. Li, Y. Matsuoka, A. Villeger, S. E. Boyd, L. Calzone, M. Courtot, U. Dogrusoz, T. C. Freeman, A. Funahashi, S. Ghosh, A. Jouraku, S. Kim, F. Kolpakov, A. Luna, S. Sahle, E. Schmidt, S. Watterson, G. Wu, I. Goryanin, D. B. Kell, C. Sander, H. Sauro, J. L. Snoep, K. Kohn, and H. Kitano. The Systems Biology Graphical Notation. *Nature Biotechnology*, 27(8):735–741, August 2009.

[NT06] N. S. Nikolov and A. Tarassov. Graph layering by promotion of nodes. *Discrete Applied Mathematics*, 154(5):848–860, 2006.

[NTB05] N. S. Nikolov, A. Tarassov, and J. Branke. In search for efficient heuristics for minimum-width graph layering with consideration of dummy nodes. *J. Exp. Algorithmics*, 10, December 2005.

[ogd] OGDF – the Open Graph Drawing Framework. Technical University of Dortmund, Chair of Algorithm Engineering; see `http://www.ogdf.net`.

[PCJ96] H. C. Purchase, R. F. Cohen, and M. James. Validating graph drawing aesthetics. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 435–446, London, UK, 1996. Springer-Verlag.

[Pur97] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 248–261, London, UK, 1997. Springer-Verlag.

[RT86] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete Comput. Geom.*, 1(1):343–353, 1986.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[San94] G. Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, October 1994.

[San96] G. Sander. A fast heuristic for hierarchical Manhattan layout. In *Proc. Graph Drawing '95*, volume 1027 of *LNCS*, pages 447–458. Springer-Verlag, 1996.

[San99] G. Sander. Graph layout for applications in compiler construction. *Theor. Comput. Sci.*, 217(2):175–214, 1999.

[San04] G. Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *Proc. Graph Drawing '03*, volume 2912 of *LNCS*, pages 381–386. Springer-Verlag, 2004.

[SFvHM10] M. Spönemann, H. Fuhrmann, R. von Hanxleden, and P. Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proc. Graph Drawing '09*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010.

[Spö09] M. Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Sys. Man. Cyb.*, 11(2):109–125, 1981.

[TKS77] N. Tomii, Y. Kambayashi, and Y. Shuzo. On planarization algorithms of 2-level graphs. *Papers of tech. group on electronic computers, IECEJ, EC77–38*, pages 109–125, 1977.

[TT86] R. Tamassia and I. G. Tollis. A unified approach a visibility representation of planar graphs. *Discrete & Computational Geometry*, 1:321–341, 1986.

[WM99] V. E. Waddle and A. Malhotra. An e log e line crossing algorithm for levelled graphs. In *Graph Drawing*, pages 59–71, 1999.

[Zie00] T. Ziegler. *Crossing Minimization in automatic Graph Drawing.* PhD thesis, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

Index

2-LCM
- 1-sided, 28
- 2-sided, 28

Chn, 87
$ucr(G)$, 41
$f \in \gamma$, 16
k-LCM, 27
k-level crossing minimization, 27
k-level planarization, 31
sT-graph, 10
st-graph, 10
$u \prec v$, 48
$u \rightsquigarrow v$, 10
$u \not\rightsquigarrow v$, 10
$x \in f$, 16
$\mathcal{M}(\Gamma)$, 48

adjacent, 9
aesthetic criteria, 13
- angular resolution, 14
- area, 13
- aspect ratio, 15
- bend, 14
- crossings, 13
- edge length, 14
- general direction, 14

alternative upward drawing approach, 126
arc, 10
- incoming arc, 10
- outgoing arc, 10

arc reuse, 93
auxiliary routing arc, 53
averaging heuristic, 28
balancing post-processing step, 26
barycenter heuristic, 28
bend minimization, 141
bend-point , 15
BJL-Layout, 33
block, 10
block (sifting), 30

chain substitution, 87
Coffman-Graham layering, 25
component, 10
confluent
- directed hypergraph, 11
- drawing, 11
- tree, 11

connected, 10
constraint feasible upward insertion path, 48
constraint upward arc insertion with fixed-embedding, 51
ConstraintFeasibleUIP, 64
crossing dummy node, 36
crossing-arc, 53
cycle
- directed, 10
- undirected, 9

cycle removal, 22
- greedy, 22
- DFS, 22
- optimal, 23

DAG, 10
- layered, 23
- leveled, 23

DAGs with known upward crossing number, 69
degree, 9
density of a graph, 30
digraph, 10
directed hypergraph, 11
directed tree, 10
directed underlying graph
  of a directed hypergraph, 11
Dominance, 38
dominance drawing, 38
dominance drawing approach, 38
dominate, 10
dominated subgraph, 48
dominating subgraph, 48
down-sweep, 28
drawing, 11
  upward planar, 12
  grid, 13
  hierarchical, 12
  layered, 12
  orthogonal, 12
  planar, 12
  polyline, 12
  straight-line, 12
  strictly upward, 12
  upward, 12
drawing style, 12
dual graph, 17
dynamic entrance, 52
dynamic locking, 56

edge, 9
embedding
  underlying combinatorial, 16
  combinatorial, 15
  planar, 16
  upward planar, 16
end node, 9
external face, 16

face, 15
  left face of an arc, 52
  left side, 52
  right face of an arc, 52
  right side, 52
  simple, 52
face-cycle, 15
face-lock, 56
face-sink-graph, 16
Feasibility Lemma, 49
feasible upward insertion path, 48
feasible upward planar subgraph, *see* FUPS
FeasibleMinUIP, 63
feedback arc set, 22
fixed-embedding scenario, 47
fixed-port scenario, 35
forest, 10
FUPS, 48

GKNV-instance, 79
GKNV-Layering, 26
GKNV-Layout, 33
global sifting heuristic, 30
Global-Sifting, 30
graph
  4-planar, 143
  upward planar, 12
  acyclic, 9
  directed, 10
  planar, 12
  simple, 10
  undirected, 9
greedy switch heuristic, 29
Greedy-CR, 23
GreedySwitch, 29
grid sifting heuristic, 30
Grid-Sifting, 30

height
  of a layering, 24
hierarchical drawing
  w.r.t. a layering, 24
hyperarc, 11
hypernode, 11
hypernode splitting, 93

ILP for k-LCM, 30
in-degree, 10

incident, 9
induce
  combinatorial embedding of a planar drawing, 15
  upward planar embedding, 46
initial orthogonal drawing, 142
inner node of a tree, 10
inner subarc, 33
insertion path, 17
intermediate UPR, 47

layer-by-layer sweep, 28
layer-free upward planarization, 42
layered drawing
  w.r.t. a layering, 24
layered upward crossing minimization, 41
layering, 23
  proper ordered, 113
  coarse, 117
  fine, 117
  proper, 24
leave of a tree, 10
left incoming arc, 114
left path, 114
length
  of a path, 9
length function, 9
level planar, 27
leveling, *see* layering
LFUP, 42
locks, 55
long arc dummies, 24
long arc dummy reduction, 118
longest path layering, 24
LongestPath, 24
loop, 55

median heuristic, 28
merge graph, 48
minimum feedback arc set, 22
MinWidth layering, 27
mixed upward planarization, 37
multi-level crossing minimization, 27
multi-run, 29
multi-set, 10
multigraph
  directed, 10
  undirected, 10
MUP, 37

node, 9
node-arc crossing, 122
North DAGs, 69

orthogonal compaction, 143
orthogonal representation, 143
out-degree, 10

path
  directed, 10
  undirected, 9
pc-valid
  arc order, 86
  embedding, 86
  insertion path, 86
planarization, 18
port, 35
port constraint valid, *see* pc-valid
promotion layering, 25

random DAG set, 69
realization
  of a layering, 113
  of a pc-valid path, 89
  of a UPR, 110
  of an insertion path, 17
realization of a layering (hypergraph), 117
regular node, 24
resolution rules, 13
Rome graphs, 68
routing network, 52

SBGN, 2
SDP for k-LCM, 30
self-loop, 9
sifting heuristic, 29
sifting trial, 30
sink, 10

sink-switch, 16
source, 10
source node, 10
source-arc, 88
source-switch, 16
spaghetti effect, 15
static locking, 56
StretchWidth layering, 27
subarc
  of a chain, 87
  of a long arc, 24
subgraph, 10
  maximum planar, 17
Sugiyama's drawing framework, 21

target node, 10
top-sink-switch, 16
tree, 10
two-cycle, 23

UIP, 46
  fixed-port scenario, 89
UML, 3
underlying graph
  of a directed graph, 10
  of a directed hypergraph, 11
  star-based, 11
underlying tree
  of a hyperarc, 11
  star-based, 11
  tree-based, 11
up-sweep, 28
UPL, 109
UPL-instance, 79
UPL-Layering, 114
UPSugiyama, 111
upward crossing minimization, 41
upward crossing number $ucr(G)$, 41
upward insertion path, 46
upward insertion sequence, 47
upward planarization, 36

Visibility, 39
visibility drawing approach, 38

width
  of a layer, 24
  of a layering, 24