Karlsruhe Institute of Technology

Automatic Layout Generation for Argument Maps

Diploma Thesis of

Christof Doll

Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Dorothea Wagner
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dr. Ignaz Rutter
Advisor:	Dipl.-Inform. Andreas Gemsa

September 2011 – February 2012

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Christof Doll
Karlsruhe, 22nd of February 2012

Deutsche Zusammenfassung

Die Diplomarbeit beschäftigt sich mit der automatischen Generierung von Layouts von Argumentkarten. Diese stellen Argumente einzelner Texte oder ganzer Debatten sowie deren Beziehungen zueinander als Graphstruktur dar und finden hauptsächlich innerhalb der Geisteswissenschaft ihre Anwendung.

Am Anfang der Arbeit steht eine empirische Analyse der graphentheoretischen Eigenschaften von Argumentkarten. Ein hoher Prozentsatz derer Zusammenhangskomponenten ist azyklisch und planar. Die verbleibenden Zusammenhangskomponenten können durch das Entfernen weniger Kanten in einen azyklischen und planaren Graphen überführt werden. Auf diese Analyse aufbauend definieren wir den grundlegenden Zeichenstil sowie Nebenbedingungen und Optimierungsziele für Layouts von Argumentkarten. Aufgrund der Azyklizität der Eingabedaten fällt die Wahl hierbei auf orthogonale Aufwärtszeichnungen. Die Knoten des Graphen werden als Boxen uniformer Breite gezeichnet, während die Höhe von dem zugehörigen Label abhängt. Zu minimieren sind dabei die Kreuzungs- und Knickzahl, die Gesamtkantenlänge sowie die Distanz von Quellen und Senken zur äußeren Facette.

Das Hauptergebnis dieser Arbeit ist ein auf dem Topology-Shape-Metrics-Framework aufbauender Algorithmus, der die gewünschten Layouts berechnet. Hierfür definieren wir die drei Schritte Topology, Shape und Metrics formal, untersuchen sie einzeln in Bezug auf ihre Komplexität und geben Algorithmen zu deren Lösung an. Im Topology-Schritt optimieren wir sowohl die Anzahl der Kreuzung als auch die Distanz von Quellen und Senken zur äußeren Facette. Dabei erzwingen wir, dass alle Kanten aufwärtsgerichtet sind. Im darauf folgenden Shape-Schritt weisen wir den Kanten 90°-Knicke zu. Dabei kann eine einzelne Kante höchstens vier Knicke erhalten. Zuletzt wird im Metrics-Schritt die Höhe und anschließend die Breite der Layouts minimiert.
Der Topology- und Metrics-Schritt erweisen sich als NP-vollständig, während wir für den Shape-Schritt die NP-Vollständigkeit nur vermuten können. Aus diesen Gründen werden die wichtigsten Schritte der Layoutberechnung heuristisch gelöst. In einer anschließenden Untersuchung der berechneten Layouts konnte gezeigt werden, dass die Optimierungskriterien hinreichend gut minimiert werden. Darüber hinaus überzeugen diese Layouts auch aufgrund ihrer Ästhetik und der übersichtlichen Struktur.

Während des Shape-Schrittes unterteilen wir die Zeichenfläche in disjunkte Spalten. Diesen werden dann einzelne oder mehrere Knoten und Kanten zugeteilt. Daher bezeichnen wir die auf diese Weise berechneten Layouts als Spalten-basiert. Dies ist jedoch nicht zu verwechseln mit Ebenen-basierten Layouts, wie sie mit der Sugiyama-Methode berechnet werden können. Während Ebenen-basierten Layouts eine topologische Ordnung der Knoten zugrunde liegt, Kanten also nur von höheren zu niedrigeren Ebenen verlaufen, existiert eine solche Beschränkung bei Spalten-basierten Layouts nicht. Dadurch wird die Positionierung mehrerer flacher Boxen neben einer hohen ermöglicht. Die Layouts wirken somit besonders kompakt.
Des Weiteren weisen wir die Knoten und Kanten so den disjunkten Spalten zu, dass für

jeden Knoten gilt, dass die ihm zugewiesene Spalte sowohl der Median der Spalten seiner eingehenden Kanten als auch der Median der Spalten seiner ausgehenden Kanten ist. Dadurch zeichnen sich die Layouts durch einen hohen Grad an Symmetrie aus, der sich wiederum positiv auf ihre Ästhetik auswirkt.

Contents

1. Introduction

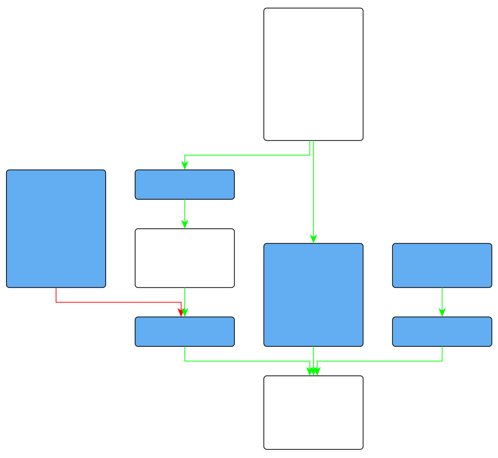
Over the last decades graphs have become a well-known and widely used modelling technique for objects and the relationships among them. Not only within computer science but across all fields of science – even within the humanities – data is represented by graphs. In most cases the work with graphs is computer-assisted but not fully automatic, i.e. the user still needs to get an overview of the graph's internal structure. This turns out to be a challenging and long-lasting task as the number of nodes and edges often is large, e.g. in the context of social networks or data structures [FMF$^+$01].
Thus, there is a significant need for computer-supported perception of data represented by graphs. This need resulted in an own branch of research within the algorithmic community: graph drawing. A lot of work has been spent on this topic and there is a variety of algorithms dealing with different input graphs or having different constraints on the layout that they compute: orthogonal drawings of planar graphs with maximum degree four [Tam87], upward drawings of DAGs [STT81], symmetric drawings of series-parallel digraphs [HEH04] – just to name a few.

In this work we cover the problem of visualising argument maps. Argument maps present the arguments given in a debate or a book together with two binary relations among them, i.e. support and attack. Thus, argument maps can be regarded as graphs. They originate from argumentation theory but are used in many more fields like philosophy and politics [Bet10]. We have the chance to work with real-life data having access to 51 argument maps. This has two major advantages: (i) we can define the desired layout style with respect to these input instances and (ii) we can evaluate our algorithm using real-world data.
In a layout of an argument map the arguments are drawn as boxes that have uniform width and are labelled by a short summary of the argument. The support and attack relations are drawn as green and red arrows, respectively. There are several constraints on layouts of argument maps. The most important ones are, that the edges are orthogonal and that they are directed downwards. Drawings in which all edges have the same directions are called *upward drawings* in the graph drawing community. For the sake of compatibility we use the upward terminology as well. Nevertheless, we draw all edges such that they are directed downwards.
Sinks and sources are exit and entry points of argument maps and need to be found quickly by the user. Therefore, we require that above a source and below a sink no other box is positioned.
While fulfilling these constraints we optimise the following four criteria:

- Number of edge crossings
- Number of bends
- Total edge length
- Distance of sinks and sources to the external face

In a series of experimental studies Purchase et al. found out that minimising the number of edge crossings and the number of bends improves human perception of graph drawings [PCJ96, PCJ97, Pur97]. Furthermore, we want to minimise the total edge length as thereby the layout becomes compact. These optimisation goals are frequently used when drawing graphs [ET88]. The last optimisation criterion is specific to layouts of argument maps and is closely related to the constraint that above a source and below sink no other box may be positioned. Despite this constraint there can run edges above a source or below a sink. We measure the distance to the external face by the minimum number of crossings a downward directed edge starting at the sink or ending at the source, respectively, needs to create in order to reach the external face.

Additionally, the algorithm we present enforces two further properties: (i) It computes so-called *column based* layouts. In a column based layout the canvas is divided into disjoint columns of uniform width that corresponds to the uniform width of the boxes. The boxes and edges are then assigned to these columns. Thereby, vertical edge segments always run within a column, whereas horizontal edge segments can span over several columns. Due to the columns the final layouts have a clear look which is exemplified in Figure 1.1a. We emphasise that column based layouts are not to be confused with layered layouts as computed using the famous Sugiyama method [STT81]. Layered layouts are based on a topological order of the nodes. Edges may be only direct from a higher layer to a lower one (see Figure 1.1b). For column based layouts we do not have such a restriction and, therefore, we can position two shallow boxes beside a single high box (see Figure 1.1a).
(ii) Our algorithm furthermore satisfies a property which we denote by *local symmetry*, i.e. for each node v the column assigned to v is the median of the columns assigned to the incoming edges of v as well as the median of its outgoing edges. Because of local symmetry the resulting layouts have a clear and well-structured look [LNS85].


(a) A column based layout of an argument map.

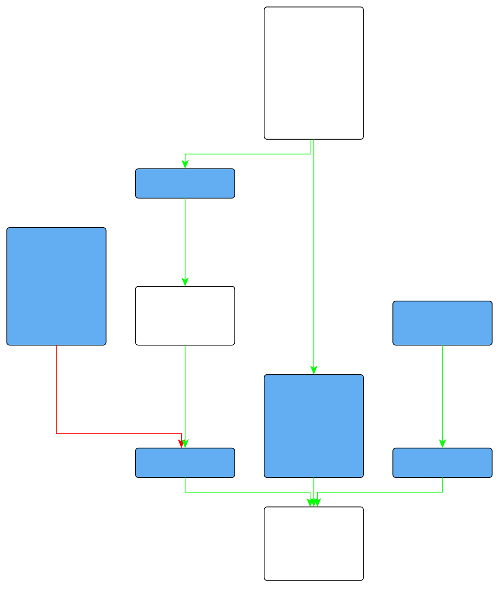

(b) A layer based layout of an argument map.

Figure 1.1: A layer based and a column based layout of an argument map.

The already mentioned algorithm is based on the *topology-shape-metrics framework* that originally was invented by Tamassia [Tam87] in 1987. The name topology-shape-metrics approach was introduced by Tollis et al. in 1998 [TDET98]. The key idea behind the framework is that reducing the number of crossings is the most important aesthetic criterion and, therefore, it is minimised in the first step without regarding any other criteria. Thereby, an embedding of the input graph is computed. If the graph is non-planar, the crossings are replaced by dummy nodes having degree four. The planarisation together with its embedding is denoted by *topology*. In general the topology is fixed throughout the remaining steps of the framework.
In the second step the *shape* of the final layout is optimised with respect to the fixed topology. Shape denotes the assignment of bends to edges and the angle of these bends. For orthogonal edges there are only two types of bends – $90°$ bends to the left and $90°$ bends to the right.
Finally, in the *metrics* step the edge lengths and node dimensions are computed. Usually, the goal of this step is to minimise the total edge length.
Each of the three steps of the topology-shape-metrics framework can be treated individually. In this work we minimise the number of crossings as well as the distance of sinks and sources to the external face in the topology step while preserving that all edges are directed upwards. Therefor, we use the layer-free upward crossing minimisation heuristic recently published by Chimani et al. [CGMW10].
For the shape step we suggest a set of algorithms. None of them minimises the number of bends to the optimum, but we can give an upper bound for the number of created bends. All algorithms have in common that they compute a column assignment of the nodes and edges of the input graph. First, we present an algorithm that assigns at most two bends per edge as long as the graph is planar. However, this algorithm is only of theoretical interest as the resulting layouts are confusing. For layouts of argument maps we use an algorithm that is based on a graph drawing technique by Biedl and Kant [BK94]. Using this algorithm the resulting layouts have at most four bends per edge independent of whether the graph is planar or non-planar. Note that due to the column assignment the topology computed in the first step can be relaxed again, i.e. the edge crossings are not determined anymore.
This is of advantage when the total edge length is minimised in the last step of the topology-shape-metrics framework. For a given column assignment we first minimise vertical edge length and, afterwards, compact the width of the layouts. We present two approaches for vertical edge length minimisation. A network flow first arbitrarily fixes a topology that is compatible with the column assignment and then computes the optimal solution subject to the fixed topology. Furthermore, we present a greedy approach that can exploit the open choices concerning the topology. The subsequent width compaction is based on a network flow idea.

We analyse the computational complexity of each of the steps that our algorithm performs. The upward crossing minimisation in the topology step is already known to be $\mathcal{NP}$-complete, because checking upon upward planarity is $\mathcal{NP}$-complete [GT94, GT02]. However, for specific parameter sets our approach can be reduced to upward crossing minimisation of an s-t-graph. Since the test upon upward planarity can be performed efficiently for s-t-graphs [BDMT98], we prove that upward crossing minimisation remains $\mathcal{NP}$-complete for s-t-graphs.
We guess that bend minimisation is $\mathcal{NP}$-complete as well. However, we cannot offer a proof of this conjecture.
Furthermore, we show that vertical edge length minimisation for a given column assignment is $\mathcal{NP}$-complete. In contrast, the subsequent width compaction can be solved optimally in polynomial time.

1.1 Outline

This work is structured as follows: In the subsequent section we give an overview about the related work. In Chapter 2 we introduce the reader to argument maps and analyse the graph-theoretic properties of a set of 51 argument maps. Afterwards, we define the drawing style of argument maps and present the corresponding formal problem statement along with complexity considerations in Chapter 3. In Chapter 4 we discuss a preliminary approach to this problem based on integer linear programming. This approach played an important role while we formulated the problem statement but is not efficient enough for the daily use. Therefore, we present an efficient algorithm building upon the topology-shape-metrics framework in Chapter 5. We discuss the final layouts in Chapter 6 before concluding this work in Chapter 7.

1.2 Related Work

As already mentioned, a lot of work has been invested within the field of graph drawing. In this section we give a short overview of the related work and present some of the most important results.

As at the beginning of this work the desired layout was not formally specified the question how to characterise a good layout arose. In the mid-nineties Purchase et al. performed a series of empirical studies testing how different formal criteria affect the aesthetic of graph layouts [PCJ96, PCJ97, Pur97]. The criteria they examined are minimising edge crossings, minimising bends, maximising symmetry, maximising the minimum angle between edges and fixing edges and nodes to an orthogonal grid. They concluded that minimising arc crossings has the most important effect on human perception. Furthermore, minimising bends and maximising symmetry improve perception as well. However, maximising the minimum angle and introducing an orthogonal grid have no statistically relevant effect.

In 1987 in his seminal paper Tamassia introduced a new approach to the field of graph drawing that inspired many other researchers to adapt and extend his approach. The innovation was to use network flows in order to compute layouts of graphs. His algorithm efficiently embeds a planar graph with a fixed combinatorial embedding and a maximum vertex degree of four on an orthogonal grid with the minimum number of bends [Tam87]. This algorithm can be seen as one of the first three-phase-methods for orthogonal graph drawing as later described by Biedl et al. [BMT97]. The three phases are (i) fixing a planar embedding, (ii) computing an orthogonal description and (iii) compaction of the layout. The term topology-shape-metrics framework was introduced by Tollis et al. in 1998 [TDET98].
The restrictions to the input data of Tamassia's algorithm are quite strong. However, there are extensions of his algorithm that can deal with graphs with no restriction on the maximum degree by replacing the vertices with boxes such that several edges can attach to the box on each side [FK95]. Eiglsperger et al. transformed the network flow approach to integer linear programs and thereby gave the opportunity to incorporate every constraint on the layout that can be formulated as a set of linear inequalities [EFK00].

As we find out in Chapter 2 that argument maps are close to directed acyclic graphs (DAGs) the famous visualisation method by Sugiyama et al. [STT81] is important in this context. They presented an approach to visualise hierarchical system structures. This approach consists of three steps: (i) layer assignment, (ii) determining relative positions within each layer to reduce edge crossings and (iii) positioning the vertices and edges. For each of the steps there are several ways how to perform it. Therefore, this approach can be seen as a general framework that can easily be modified. However, it has one drawback that none of the further developments could overcome: Choosing a wrong layer assignment in step (i)

can require many edge crossings in step (ii) which would not be necessary, if another layer assignment had been chosen (see Figure 5.1 on Page 48 for an example).
Having this problem in mind Chimani et al. recently developed an approach to layer-free upward crossing minimisation [CGMW10]. This approach adapts the planarisation approach by Batini et al. [BTT84] which is the most successful heuristic for minimising crossings in undirected graphs [GM04] to upward crossing minimisation of directed graphs. In a follow-up paper they included this approach in the Sugiyama framework [CGMW11]. After the crossing minimisation they compute a layering of the nodes that is compatible with the given topology and then assign coordinates to the nodes and edges, i.e. they replace the first two steps of the Sugiyama framework with the layer-free upward crossing minimisation method. In contrast to Chimani et al. we integrate the layer-free upward crossing minimisation method into the topology-shape-metrics framework.

The drawing of argument maps has some similarities to layout UML class diagrams. In UML class diagrams there are two types of edges: (i) generalisations and (ii) associations. Mostly, the generalisations are required to be drawn upward, whereas the associations can be directed in any direction. There are approaches that extend the Sugiyama approach to UML class diagrams by first drawing the directed edges using the classical Sugiyama framework and afterwards routing the associations [See97].
Eiglsperger et al. introduced the concept of mixed-upward planarity and developed algorithms for the creation of mixed-upward drawings [Eig03, EKS03, EEK03]. However, their heuristic for upward edge insertion is still based on a layering of the graph and, therefore, has the same drawback as the Sugiyama framework.

2. Input Characterisation

In this chapter we introduce the reader to the field of argument maps – what they are used for and how they are constructed. Furthermore, we analyse their graph-theoretic properties. The goal of this analysis is to characterise graph-theoretic properties that are special to the domain of argument maps. These are the properties we will later try to exploit in two ways: On the one hand we can use it for the specification of the layout we want to compute, e.g. for acyclic graphs we can consider upward drawings. On the other hand we can try to exploit the gathered information in terms of efficiency of the algorithm that computes the specified layout, e.g. some problems are $\mathcal{NP}$-complete for general graphs but can efficiently be solved for special classes of graphs.

First we formally define argument maps in Section 2.1. Afterwards, in Section 2.2 we briefly give summaries of the three sources that build the empirical basis for the following analysis. On the one hand these sources differ in the purpose of the argument maps and on the other hand in the experience of the creator in dealing with argument maps. In Section 2.3 we list the analysis methods we use and depict the collected data in form of several diagrams.

2.1 Argument Maps

In this section we first give an overview about the terminology used in the context of argument maps including formal definitions of the terms *argument*, *thesis*, *support*, *attack* and *argument map*. Thereby, we show how argument maps are associated with graphs. Afterwards, we discuss the use cases of argument maps and present a simple example.

Definition 2.1 (Argument)**.** *An* argument *is a propositional formula of the form:*

$$\text{premise}_1 \wedge \ldots \wedge \text{premise}_n \Longrightarrow \text{conclusion}$$

We assume that an argument is deductively valid.

Definition 2.2 (Thesis)**.** *A* thesis *is a single term in the propositional logic:*

$$\text{thesis}$$

Arguments and theses will form singular entities in argument maps and can be in a supporting or mitigating relation.

Definition 2.3 (Support)**.** Support *is a binary relation on the set of arguments and theses. In general this relation is asymmetric. A* supporting *relation can exist between two arguments or between a thesis and an argument. There is a support from argument A_1 to argument A_2 if the conclusion of A_1 is logically equivalent to one of the premises of A_2. If A_1 (or A_2) is a thesis instead of an argument the term* thesis *needs to be equivalent to a premise of A_2 (the conclusion of A_1).*

Definition 2.4 (Attack)**.** Attack *is defined analogously to support. Instead of logical equivalence we require logical antivalence.*

Basing on Definition 2.1-2.4 we can now define argument maps:

Definition 2.5 (Argument Map)**.** *An* argument map *consists of a set V of arguments and theses and the two relations $A_{\text{support}}, A_{\text{attack}} \subseteq V \times V$.*

Obviously, an argument map can be interpreted as a graph structure $G = (V, A)$ where $A = A_{\text{support}} \cup A_{\text{attack}}$. Arguments and theses form the nodes of G, whereas supports or attacks between them are represented by directed edges.

When the graph representing an argument is drawn, the nodes are drawn as rectangular boxes containing a summary of the argument. Supporting edges are drawn as green arrows, whereas mitigating edges are red. In Figure 2.1 we show a rather small example of a layouted argument map. In this work we do neither distinguish between theses and arguments nor between supporting and attacking edges when computing layouts for argument maps. However, in Chapter 7 we show, that this distinction could be part of future work.

The approach to visualise arguments in an argument map originates from argumentation theory but is used as a tool in vast fields of other domains like philosophy in general, education and politics. In the academic context they are most frequently used in order to represent the arguments that are given in one or in several texts. However, they are applied in order to visualise whole debates as well, e.g. in a political context. There even have been experiments in live reconstruction of debates, e.g. during a party conference of the German party Bündnis 90/Die Grünen the audience could observe the argument map representing the discussion that emerged among them on a large screen. For more detailed information about argument maps we refer the reader to the book “Theorie dialektischer Strukturen” by Betz [Bet10].

The reconstruction of debates using argument maps is supported by a tool called argunet[1]. However, this tool does not supply any layout techniques. The user needs to place the boxes on his own and edges are modelled as straight lines connecting the centre points of two boxes. Especially the straight line constraint on edges imposes a restriction on the size of the argument maps, because maps get confusing if the number of arguments increases. Furthermore, it is time consuming to manually create a good layout. Creating the layout of the poster mentioned in the paragraph about “The Moral Controversy about Climate Engineering” in the next section was an effort of several hours. Therefore, we would like to give an algorithmic support for the layout of argument maps. Before we come to the algorithms, we turn towards the properties that distinguish argument maps from general graphs. In order to analyse these properties, we gathered realistic input data from three sources. Their description is the content of the following section.

2.2 Sources

In this section we present the three sources that supply the input data we use for the analysis in the subsequent section.

[1]See `http://www.argunet.org`

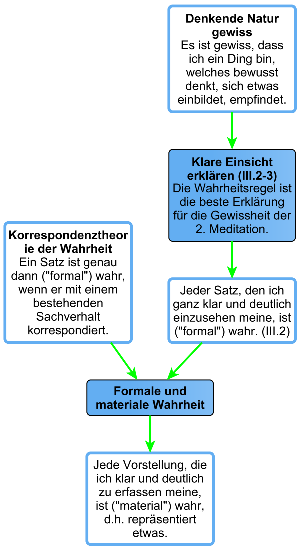

Denkende Natur gewiss
Es ist gewiss, dass ich ein Ding bin, welches bewusst denkt, sich etwas einbildet, empfindet.
Klare Einsicht erklären (III.2-3)
Die Wahrheitsregel ist die beste Erklärung für die Gewissheit der 2. Meditation.
Korrespondenztheorie der Wahrheit
Ein Satz ist genau dann ("formal") wahr, wenn er mit einem bestehenden Sachverhalt korrespondiert.
Jeder Satz, den ich ganz klar und deutlich einzusehen meine, ist ("formal") wahr. (III.2)
Formale und materiale Wahrheit
Jede Vorstellung, die ich klar und deutlich zu erfassen meine, ist ("material") wahr, d.h. repräsentiert etwas.

Figure 2.1: A simple argument map.

The Moral Controversy about Climate Engineering

A reconstruction of the controversy about research into, development and implementation of Climate Engineering-technologies. This reconstruction consists of six argument maps. The first five represent only parts of the whole controversy. These five maps are then composed into one big map representing the whole controversy. This map has been manually layouted by Betz and Cacean for a poster [BC11].

Philosophisches Argumentieren I

A set of reconstructions that have been created during a course given by Betz at the Free University of Berlin during the winter term 2007/2008[2]. Between the weekly sessions the students created argument maps representing the arguments given in a philosophical text and presented them to their fellow students in the next session.

René Descartes: Meditationen über die Grundlagen der Philosophie

A collection of 34 argument maps consisting of 4 up to 31 arguments. These maps have been reconstructed from the meditations by Descartes [Des08]. Each meditation is covered by a set of several argument maps. These maps have been used by Betz during the lecture "René Descartes: Meditationen über die Grundlagen der Philosophie" he gave at the University of Stuttgart in winter term 2008/2009[3].

In total the three sources supply 51 argument maps. Together they form a small but sound sample whose analysis is the subject of the following section.

2.3 Analysis

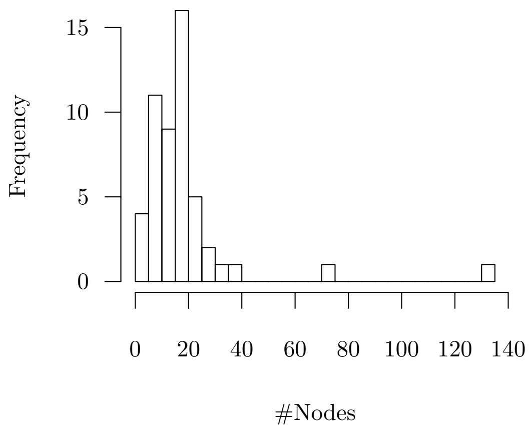
After presenting the sources of the argument maps in the foregoing section we now turn towards their analysis. In this section we discuss our analysis methods and apply them to the supplied argument maps. Thereby, we focus on the pure graph structure corresponding

[2]See http://myvv.fu-berlin.de/vorlesungsverzeichnis/myvvq.php?dozent=betz&keyword=&lvnummer=&kommentar=1&fachbereich=007001001001001001&semester=WS0708

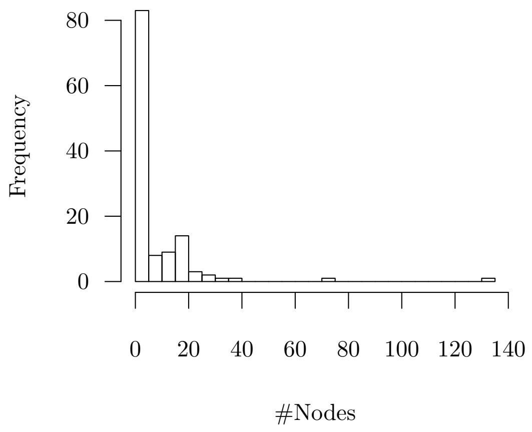
[3]See http://www.uni-stuttgart.de/philo/index.php?id=375&no_cache=1&tx_ttnews%5Btt_news%5D=734&cHash=94989b0c1092b61dc4cdf3ca98b5a80f

to an argument map and do not distinguish between arguments and theses as different types of nodes or support and attack as different types of edges.

The first property we analyse is the size of the argument maps measured in terms of the number of nodes. The vast majority of argument maps turned out to be of modest size, i.e. 96% of the argument maps consist of at most 40 nodes and 78% have at most 20 nodes (see Figure 2.2a). As most layout algorithms compute the layouts of each connected component of a graph separately, we also examined their size. Within this analysis we focused on weakly connected components, i.e. the arc directions do not matter when determining connectivity. Over all argument maps there were 123 weakly connected components. Roughly the half of them are singletons, meaning that they consist of a single node. Furthermore, 93% of all components consist of at most 20 nodes. See Figure 2.2b for more information on the size of connected components. We will later take a closer look on the connected components and examine whether they belong to some special classes of graphs like trees or s-t-graphs.



(a) Broken down by graph size.



(b) Broken down by component size.

Figure 2.2: Distribution of the size of argument maps.

We also consider the density of the graphs, i.e. the ratio of the number of edges to the number of nodes. Again, argument maps show a pretty simple structure as can be seen in Figure 2.3. The number of edges appears to be linear with respect to the number of nodes with a factor of proportionality between 0.7 and 1.3. There are some exceptions having less or even no edges at all.

Now, we investigate the degree distribution of the argument map. Figure 2.4 shows the degree distribution over all 51 argument maps. Only little nodes have a degree greater than five and a degree between one and three is most common.

We now turn to the already mentioned analysis whether the connected components belong to some specific graph classes. We consider the following eight classes of graphs:

- s-t-graphs
- series-parallel graphs
- singletons
- rooted trees
- trees
- acyclic graphs
- planar graphs
- simple graphs

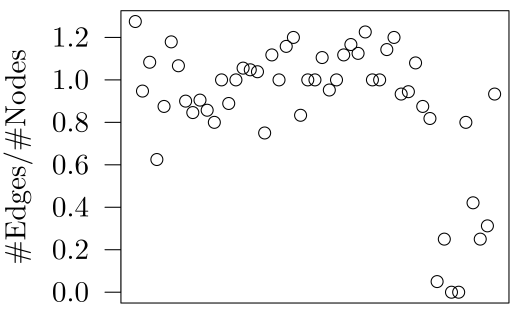


Figure 2.3: Density of argument maps.

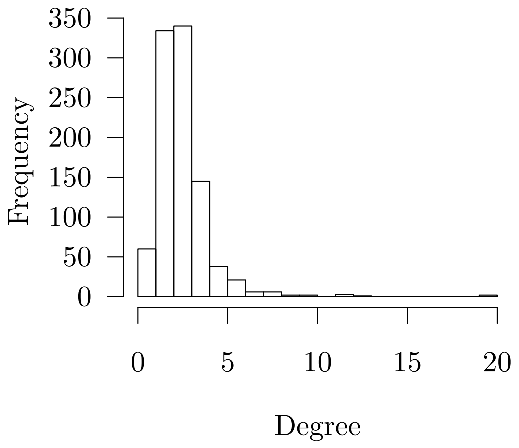


Figure 2.4: Degree distribution over all argument maps.

We plot the results in Figure 2.5a. All components are simple and 96% and 95% are planar or acyclic, respectively. At the first glance one might find the high percentage of trees and rooted trees remarkable as well. But taking into consideration that roughly half of all components are singletons the percentage of non-singleton trees and rooted trees decreases to 24% and 13%, respectively. Thus, we cannot focus on the properties of trees, rooted trees, series parallel graphs and *s*-*t*-graphs. Instead, we take a closer look on planarity and acyclicity. In Figure 2.5b we again plot the percentage of planar components and acyclic components and additionally the percentage of components that are both, planar and acyclic, which is 93%.

As there are some components that are not planar or not acyclic we will investigate their distance to planarity or acyclicity. The distance measure we use here is the number of edges one needs to delete in order to get a planar or acyclic graph, respectively. In the case of acyclicity such a set of edges is denoted by *feedback arc set*; in the case of planarity this problem is called *maximum planar subgraph problem.* Both, finding a minimum feedback arc set and a maximum planar subgraph are $\mathcal{NP}$-complete problems as shown by Karp [Kar72] and Garey and Johnson [GJ79], respectively. For our analysis we used the heuristics implemented in the methods `y.algo.Cycles.findCycleEdges()` and

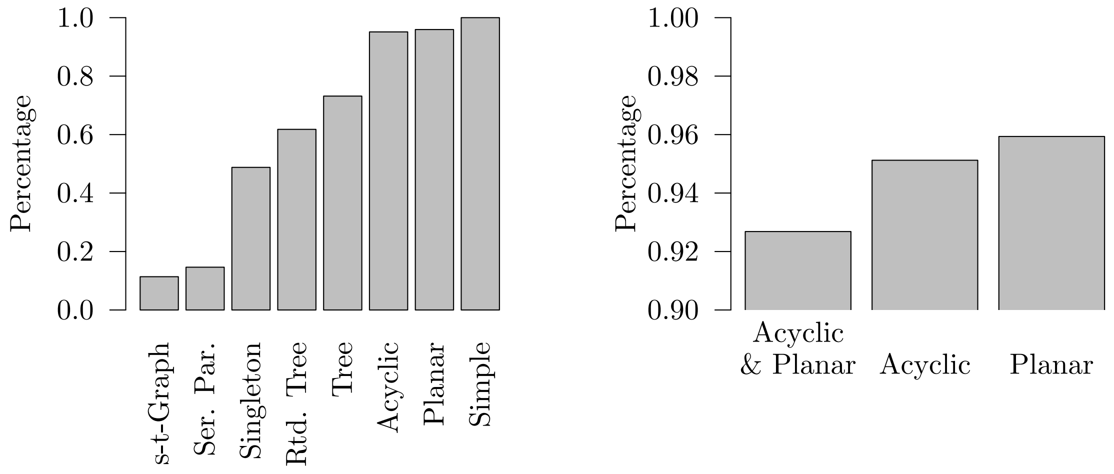


Figure 2.5: Analysis of the components of argument maps.

`y.layout.planar.GT.createPlanarization()` of the *yFiles* library[4].
As can be seen in Figure 2.6 the connected components are very close to acyclicity or planarity, respectively. Thus, it is perfectly legitimate to apply a special treatment to the edges prohibiting acyclicity or planarity in order to gain acyclic and planar input data. We will come back to this point in the context of the layout algorithm in Chapter 5.

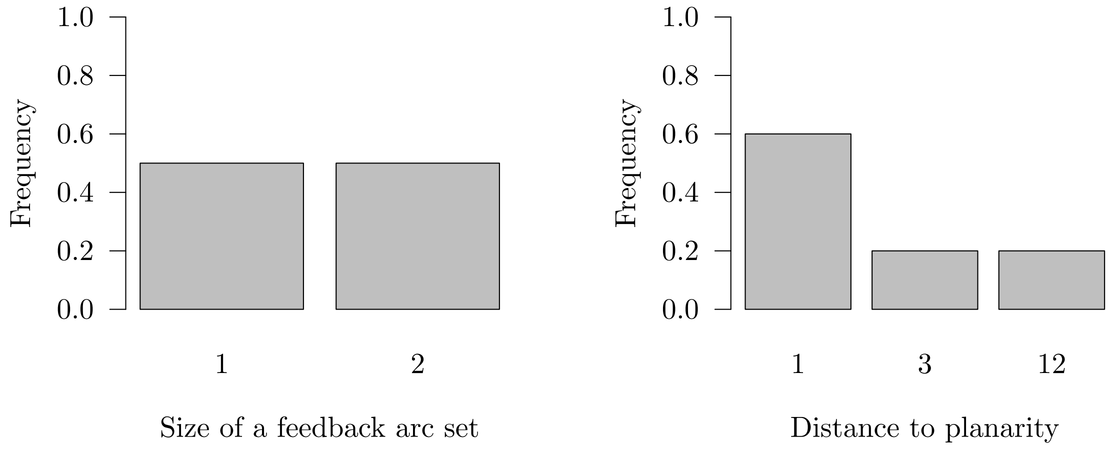


Figure 2.6: Examination of cyclic and non-planar components.

We conclude that argument maps have a pretty simple structure. The components of an argument map usually have at most 40 nodes and are of maximum degree 5. Furthermore, the components usually are planar and acyclic. If they are not planar, they are close to a planar subgraph, i.e. one need to delete only few edges in order to gain planarity. The same holds for acyclicity.

[4]See `http://www.yworks.com`

3. Problem Description

In the previous chapter we described argument maps and analysed their graph-theoretic properties. In this chapter we shift the focus to layouts of argument maps. In Section 3.1 we define the basic drawing style, whereas we discuss the properties a layout of an argument map needs to fulfil in Section 3.2. Together with the optimisation goals, which we describe in Section 3.3, we carve out the formal problem statement ARGUMENT MAP in Section 3.4. We close this chapter with complexity considerations of two subproblems of ARGUMENT MAP and two closely related problems in Section 3.5.

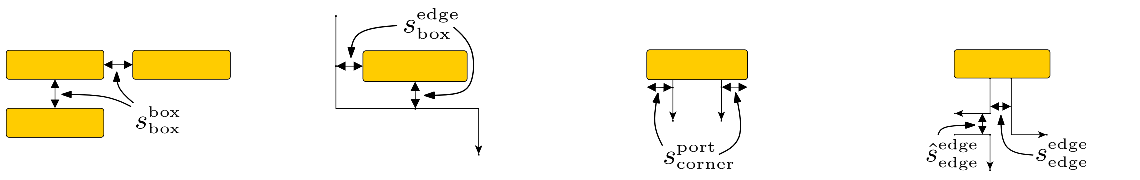
3.1 Basic Drawing Style

In this section we define the basic drawing style of the desired layouts. First, all boxes need to be of *uniform width* w. The heights of the boxes can differ from each other and depend on the size of the label describing the corresponding argument. Thus, we prescribe the height $h(b_i)$ for each box b_i.
We require that the edges are routed *orthogonally*, i.e. each edge consists of an alternating sequence of horizontal and vertical edge segments.
Furthermore, all edges need to be drawn *upwards*. In the context of orthogonal edges this means that no edge segment is directed downwards. Note that we use the term upward drawing for compatibility reasons with the vocabulary used in the graph drawing community. However, all figures of argument maps are downward drawings as this is the preferred drawing style for argument maps. Edges enter their targets at the top and leave their sources at the bottom. Ports on the left or right side of a box are not allowed.
In the following section we discuss minor properties layouts of argument maps shall satisfy.

3.2 Properties of Layouts of Argument Maps

In this section we focus on minor properties we require for a layout of an argument map. These are minimum spacing between boxes and edges, alignment of boxes, grouping and free sources and sinks.

We require five different types of *spacing constraints* (see Figure 3.1). As shown in Figure 3.1a we require a minimum horizontal or vertical distance $s_{\text{box}}^{\text{box}}$ between each pair of boxes. Furthermore, edges need to have a minimum distance $s_{\text{box}}^{\text{edge}}$ to boxes (see Figure 3.1b) and the edge ports must be at least $s_{\text{corner}}^{\text{port}}$ units distanced from the corners of the incident box (see Figure 3.1c). The last type of constraint concerns the distances between two segments of different edges. We require minimum distances between two parallel edge segments (see Figure 3.1d). If two edges have a common source or target the distance shall be at least $s_{\text{edge}}^{\text{edge}}$ and otherwise the distance must be at least $\hat{s}_{\text{edge}}^{\text{edge}}$. By using these two different minimum spacings we enable edge bundling. Note that the spacing constraints on edge segments do not forbid edge crossings as the constraints are only required for parallel edge segments.


$s_{\text{box}}^{\text{box}}$
$s_{\text{box}}^{\text{edge}}$
$s_{\text{corner}}^{\text{port}}$
$\hat{s}_{\text{edge}}^{\text{edge}}$
$s_{\text{edge}}^{\text{edge}}$

(a) Spacing between boxes.

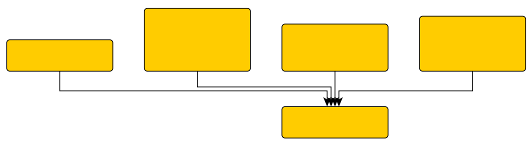
(b) Spacing between an edge and a box.

(c) Spacing between the ports of a box and its corners.

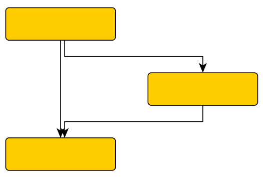
(d) Spacing between two edges.

Figure 3.1: Illustration of the five types of spacing constraints.

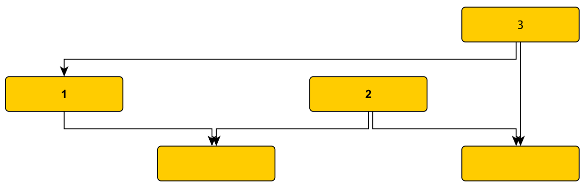
When looking at a single argument, it is important that its predecessors can be found quickly. In order to give them a unifying look, we require the predecessors of each box to be *vertically aligned* at their bottom as shown in Figure 3.2a. In general this is not possible: Let p_1 and p_2 be two predecessors of a box b. If there is a directed path from p_1 to p_2 (or vice versa) the alignment of p_1 and p_2 conflicts the upward drawing constraints (see Figure 3.2b). Therefore, we only require vertical alignment of so called *immediate predecessors*. A node p is an immediate predecessor of b if there exists an edge (p, b) and (p, b) is the only directed path from p to b.
However, restricting to immediate predecessors does not fully resolve the problem with directed paths between aligned predecessors. Consider a box that is a predecessor of two boxes, like Box 2 in Figure 3.2c. Box 2 should be aligned with Box 1 as they have a common successor and for the same reason it should be aligned with Box 3. However, we cannot align all three boxes together as there exists a directed path from Box 3 to Box 1. However, we can use the groups $\{1, 2\}$ and $\{3\}$ or $\{1\}$ and $\{2, 3\}$. In such a situation we allow to resolve the conflict arbitrarily by choosing one possible grouping for alignment. Furthermore, we require alignment of all sinks.



(a) The predecessors of a box need to be aligned.



(b) In general alignment is not possible.



(c) The groups of aligned boxes are not well-defined.

Figure 3.2: Alignment of boxes.

The user can provide some information about how boxes shall be grouped in the layout. Such a *grouping* can visualise semantics, e.g. common authorship. Each grouping is related with a *grouping box*. A grouping box is a rectangular region in the drawing which contains the boxes that shall be grouped and no other one. Edges that are not adjacent to any box in the grouping box may not cut through the grouping box. See Figure 3.3 for a simple example: The Boxes 2-5 are grouped by the blue grouping box. Boxes 1 and 6 may not be positioned in the grouping box. Furthermore, edge $(1, 6)$ may not cut through the grouping box as it is not adjacent to any box within it. Note that a grouping box can be used for a further grouping, i.e. we allow nested groupings as well.

The information about all groupings can be stored in a tree T that consists of two types of nodes, *singleton nodes* and *grouping nodes*. The singleton nodes need to be the leafs of T, whereas the grouping nodes can only be inner nodes. For each node of the argument map there is exactly one corresponding singleton node in T. The number of grouping nodes is equivalent to the number of groupings that shall be enforced in the final layout. For each grouping node g in T the boxes that are children of g shall be enforced to be in one grouping box. In the most simple case the root is a grouping node and all singleton nodes are direct children of the root, i.e. no grouping is enforced at all.

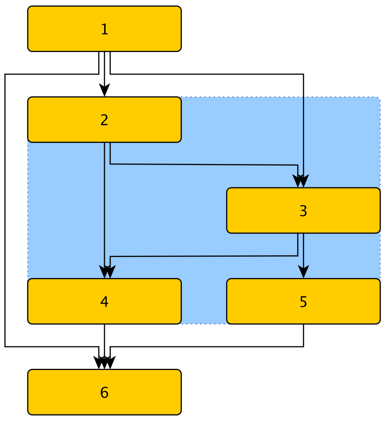


Figure 3.3: A grouping box (blue region) of four boxes.

The sources of an argument map are an entry point into the flow of the argumentation, whereas the sinks are exit points. Therefore, these boxes need to be found quickly. Thus, we require that above a source there is no other box and, the other way round, below a sink should be no other box. We denote this property by *free sources and sinks*.

3.3 Optimisation Goals for Layouts of Argument Maps

In the foregoing section we discussed the desired minor properties that extend the basic drawing style defined in Section 3.1. In this section we come to the optimisation goals. We present four optimisation criteria that we aim to minimise. Three of them are optimisation goals that are typical to graph drawing, whereas the fourth one is special to layouts of argument maps. We will use these optimisation goals in order to define the cost function in the formal problem statement Argument Map in Section 3.4.

As common in the field of graph drawing we want to minimise the *number of crossings* as well as the *number of bends*. Purchase et al. have shown in a series of user studies that the number of crossings in a layout has the most important impact on human perception among a set of different aesthetic criteria for graph drawing [PCJ96, Pur97, PCJ97]. Furthermore, they showed that reducing the number of bends increases the readability as well.
Furthermore, we minimise the *total edge length*, i.e. the sum over all edge lengths. This yields compact layouts with little free space [FMF$^+$01].

The last optimisation goal we pursue is specific to layouts of argument maps and we denote it by *total source/sink distance minimisation.* The sources of an argument map are an entry point into the flow of the argumentation, whereas the sinks are exit points. Therefore, these boxes need to be found quickly. In Section 3.2 we already demanded that above a source and below a sink should be no other box. However, there can be edges. We further want the sources and sinks to be layouted close to the external face of the layout. The goal is to minimise the distance between a source or sink and the external face.
Let t_i be a sink and f_0 the external face of the layout. We define the *sink distance* of t_i as the minimum number of crossings an imaginary upward directed edge starting at t_i needs in order to reach the external face f_0. In Figure 3.4 we show that the sink distance does not correspond to the shortest path from the face of t_i to f_0 in the dual graph. We cannot route the imaginary edge along this shortest path to the external face as it would conflict with the upward drawing constraint.
Analogously, we define the *source distance* of a source s_i as the number of necessary crossings of an imaginary edge starting somewhere in the external face f_0 and ending at s_i. The *total source/sink distance* is the sum of all source distances and all sink distances in a graph.

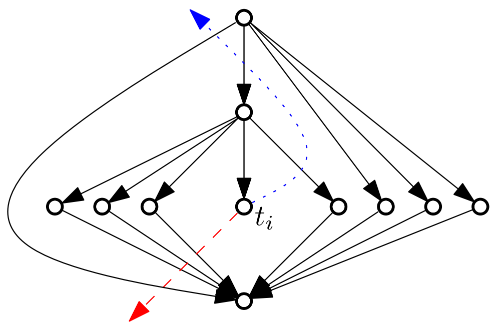


Figure 3.4: The shortest path in the dual graph (blue dotted) has length 3, whereas the imaginary edge (red dashed) needs at least 4 crossings in order to reach the external face f_0.

After defining the basic drawing style, stating further constraints a layout needs to satisfy and discussing the optimisation goals we are now ready to give a formal problem statement in the following section.

3.4 The Problem Argument Map

In this section we introduce the formal problem statement Argument Map. Argument Map wraps up the properties described in Section 3.2 and the optimisation goals presented in Section 3.3:

Instance: A DAG $G = (V, A)$, each vertex is associated with an integer height $\mathrm{h}(v)$, a uniform integer box width w, a tree T defining the grouping boxes, a set of spacing constraints, four real parameters $\alpha, \beta, \gamma, \delta \geq 0$ and one real number k.
Question: Does there exist a layout of G that satisfies the requirements described in Section 3.2 and optimises the criteria presented in Section 3.3 such that the following inequality holds:

$$\alpha \cdot \#\text{edge crossings} + \beta \cdot \#\text{bends} + \gamma \cdot \text{total edge length} + \delta \cdot \text{total source/sink distance} \leq k$$

Obviously, this problem statement is a quite complex one and it is not surprising that Argument Map is $\mathcal{NP}$-complete. Therefore, we analyse the complexity of two subproblems of Argument Map and two closely related problems in the next section.

3.5 Complexity Considerations

In this section we firstly analyse two subproblems of Argument Map where the optimisation is restricted to bend minimisation and crossing minimisation, respectively, and prove $\mathcal{NP}$-completeness for them in Section 3.5.1 and Section 3.5.2. Afterwards, we examine the complexity of two problems that are closely related to Argument Map. In Section 3.5.3 we show $\mathcal{NP}$-completeness of Rectilinear Upward Drawing and in Section 3.5.4 we prove that Directed Visibility Representation is $\mathcal{NP}$-complete.

3.5.1 Argument Map is $\mathcal{NP}$-complete for $\beta > 0$ and $\alpha = \gamma = \delta = 0$

In this section we show that Argument Map is already $\mathcal{NP}$-complete if we restrict to bend minimisation:

Theorem 1. Argument Map *is $\mathcal{NP}$-complete for $\beta > 0$ and $\alpha = \gamma = \delta = 0$.*

We prove Theorem 1 by reduction from 3-Partition, which Garey and Johnson defined in their famous book about complexity theory published in 1979 as follows [GJ79]:

Instance: A finite set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$ and a "size" $s(a) \in \mathbb{Z}^+$ for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that the equation $\sum_{a \in A} s(a) = mB$ holds.
Question: Can A be partitioned into m disjoint sets $S_1, S_2, \dots, S_m$ such that for $1 \leq i \leq m$ the equation $\sum_{a \in S_i} s(a) = B$ holds? (Note that the above constraints on the item sizes imply that every such S_i must contain exactly three elements from A.)

Garey and Johnson proved that this problem is not only $\mathcal{NP}$-complete but $\mathcal{NP}$-complete in the strong sense as well. This means that 3-Partition remains $\mathcal{NP}$-complete if the numeric values encoded in the input data are polynomially bounded by the length of the input.

While transforming a 3-Partition instance to an Argument Map instance we basically construct gadgets of three types: (i) a frame with dimensions depending on mB, (ii) one number gadget for each $a_i \in A$ and (iii) two synchroniser gadgets that synchronise the number gadgets. The basic idea of the transformation is that a rectangular frame is separated into m cells. Each of these cells will represent one of the sets S_i in the solution of the 3-Partition instance. The dimensions of the frame are rigid as long as no bends are introduced to the edges belonging to the frame. As already mentioned, there are $|A|$ number gadgets – each representing an element $a_i \in A$. The number gadgets consist of a head, a linear chain and a foot. The head is only one box wide, whereas the width of the foot is a_i. While the head and foot are rigid, the linear chain of a number gadget is flexible, i.e. the head and foot can move relative to each other. We will restrict to three number gadgets per cell and to a total width of at most B boxes for the foots of the three number gadgets in each cell. In order to enforce these constraints we need two synchronisers. One of them synchronises the heads of the number gadgets together with a part of the frame, whereas the other one synchronises the foots. In Figure 3.5 we depict this prove schematically. In this figure as well as in the remaining part of this section, we use the 3-Partition instance $A = \{2, 2, 2, 2, 3, 3\}, B = 7$ as an example.
For the constructed Argument Map instance we use the following minimum spacing constraints: $s^{\text{box}}_{\text{box}} = 40$, $s^{\text{edge}}_{\text{edge}} = \hat{s}^{\text{edge}}_{\text{edge}} = 5$, $s^{\text{edge}}_{\text{box}} = 20$ and $s^{\text{port}}_{\text{corner}} = 20$. Furthermore, we require a uniform box width $w = 135$.

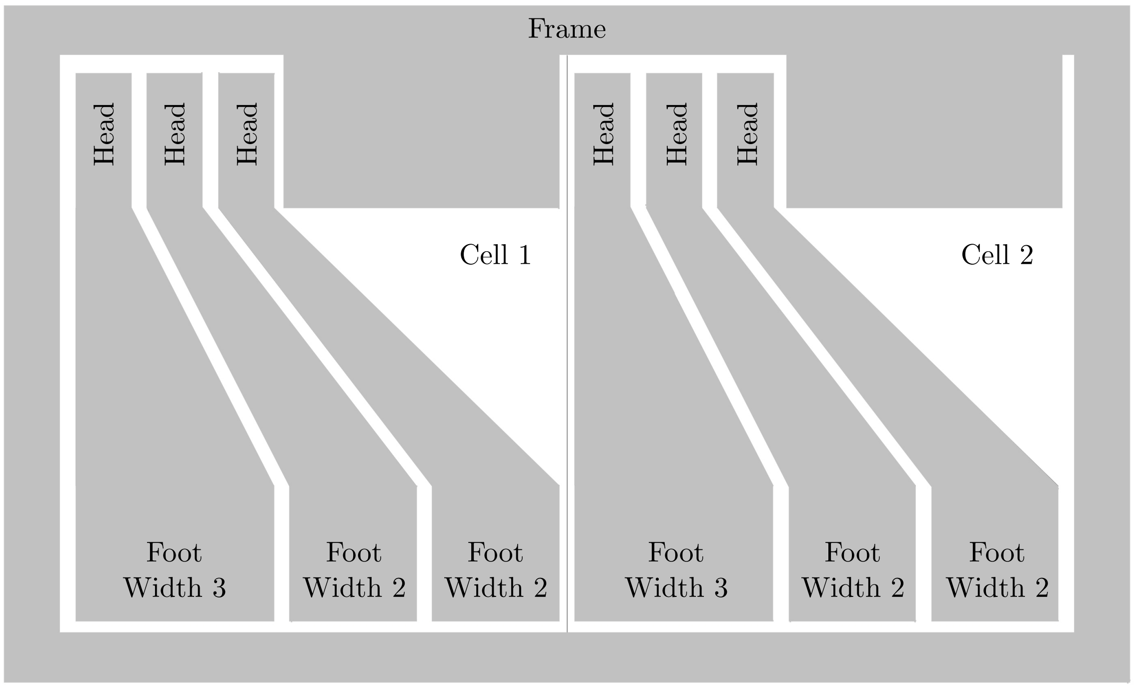


Figure 3.5: Schema of the Argument Map instance corresponding to the 3-Partition instance $A = \{2, 2, 2, 2, 3, 3\}, B = 7$. The synchronisers are missing.

The Frame

Figure 3.5 already depicts the frame schematically. It basically has a rectangular shape and is separated into m cells. The top-right corner of each cell is occupied by a couple of *blocks* (we will define this term later). We now describe in detail, how the frame depicted in Figure 3.6 is constructed.

First we describe the construction of the upper border – the construction of the lower border is analogue. On the upper border we have two layers of boxes, $P = \{p_1, \dots, p_{mB+1}\}$ and $Q = \{q_1, \dots, q_{mB}\}$ where P is the upper layer and Q the lower one. Layer P contains $mB + 1$ boxes, whereas Q contains mB boxes. We add two edges from p_i to q_i and three edges from p_{i+1} to q_i for $i \in \{1, \dots, mB\}$. Since each pair p_i and p_{i+1} are shared predecessors of q_i, they need to be aligned. Thus, all boxes in P need to be aligned.
Assume that the edges between P and Q are not bent. Due to the spacing constraints between the ports and the corner of the boxes ($s^{\text{port}}_{\text{corner}}$) and the spacing constraints between the duplicate edges ($s^{\text{edge}}_{\text{edge}}$), the boxes in P and the boxes in Q need to have a horizontal distance of at most 40 to each other. However, all boxes in P are side by side and, therefore, the horizontal distance between them must be at least $s^{\text{box}}_{\text{box}} = 40$. Thus, the boxes in P are *horizontally rigid* as long as no bends are added, i.e. the x-coordinate of one box determines the x-coordinates of all other boxes. This property is conveyed to the boxes in Q, i.e. for the whole upper border. Due to analogy the lower border is horizontally rigid as well.
Furthermore, there exist $m + 1$ edges starting at the upper layer of the upper border and ending at the lower layer of the lower border. For i from 0 to m, they connect the $(i \cdot B + 1)$-th boxes of those layers with each other. Thereby, the inside of the frame is separated into m cells of uniform width B. These cells correspond to the m sets S_i in the solution of the 3-Partition instance.
Each cell contains $B - 3$ *blocks* that are attached to the upper border. A block consists of two boxes that are connected with as many edges as possible with respect to the spacing constraints, i.e. 19 edges. Thus, the two boxes of a block are horizontally rigid as well.

The $B-3$ blocks are attached to the $B-3$ rightmost boxes of the lower layer of the upper border. Again, we use 19 edges to connect the blocks to the upper border and, thereby, make this connection horizontally rigid as well.

Now, we turn towards the left and the right border of the frame. The vertical borders consist of one grouping box each. In each grouping box there are three boxes which are connected by edges such that their vertical order is fixed. We denote these boxes from top to bottom by ℓ_i and r_i for $i \in \{1, 2, 3\}$. The boxes ℓ_1 and r_1 are connected with the upper border, whereas ℓ_3 and r_3 are connected to the lower border. Furthermore, there are the two edges (ℓ_1, r_2) and (r_1, ℓ_2). These edges enforce that ℓ_1 and r_1 are aligned. The boxes ℓ_3 and r_3 need to be aligned as well, because they are predecessors of nodes in the lower layer of the lower border. Thus, the two grouping boxes are of exactly the same height.
Since the source and target of the edges (ℓ_1, r_2) and (r_1, ℓ_2) lie in different grouping boxes that are layouted side by side, each of these edges will be bent at least two times. Thus, in total the minimum bend count of the frame is 4.

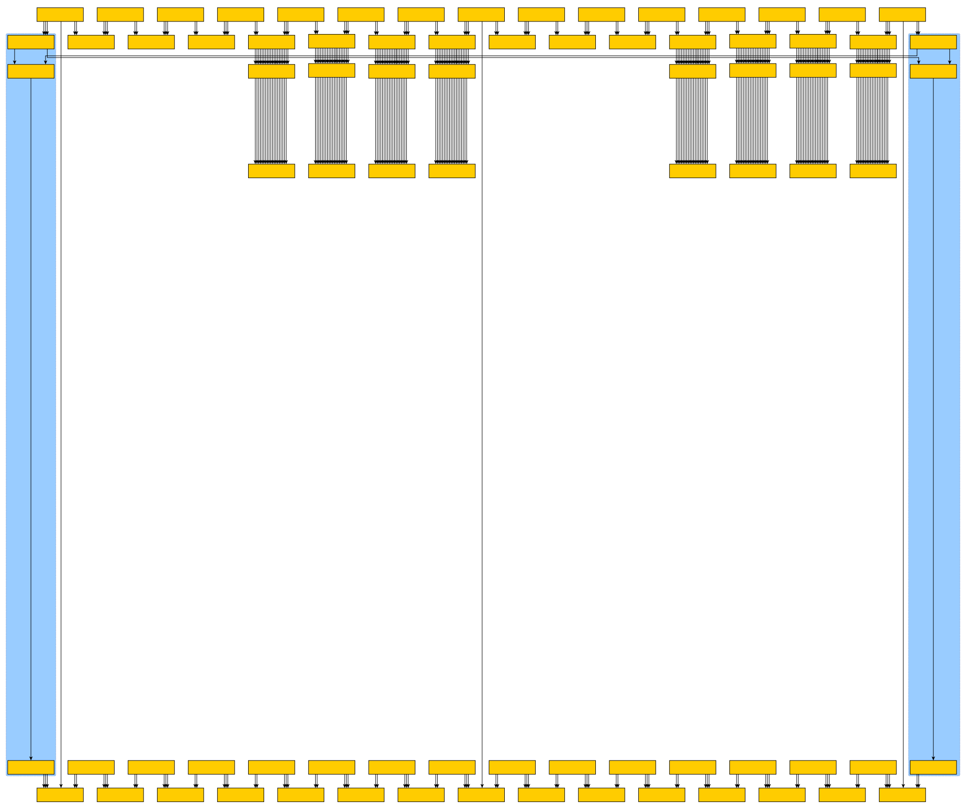


Figure 3.6: The frame for the 3-Partition instance $A = \{2, 2, 2, 2, 3, 3\}, B = 7$.

The Number Gadgets

We construct one number gadget for each $a_i \in A$. A number gadget consists of three parts that are vertically ordered. We denote the first part by *head*, the second one by *linear chain* and the last one by *foot*. Only the foot depends on the value of a_i. The head and the foot are horizontally rigid, whereas the linear chain is horizontally flexible. We will now present the three parts in detail. Note that the last box of part one (two) is the same as the first box of part two (three).

The *head* is a block like the other $B-3$ blocks belonging to each cell of the frame.
The *linear chain* consists of $2 \cdot \lceil 2/3 \cdot B \rceil + 1$ boxes. We denote these boxes by c_j. For j

being odd we add three edges from c_j to c_{j+1}, whereas we add two edges for even values of j. This part can either be layouted in a vertical manner like in Figure 3.7a or with a horizontal shift as depicted in Figure 3.7b. We measure the horizontal shift in the number of boxes that fit into the gap we introduced by shifting. Due to the spacing constraints, the maximum possible shift is $\lceil 2/3 \cdot B \rceil$. Note that a maximum shift of $\lceil 2/3 \cdot B \rceil$ is sufficient, because the number gadget in a cell that corresponds to the biggest number can always be layouted as the right most number gadget in the cell.
The *foot* consists of a_i blocks which are attached to a triangle-shaped horizontally rigid *connector*. The connector consists of $2 \cdot (a_i - 1)$ layers of which the ℓth layer contains $\lfloor (\ell + 1)/2 \rfloor$ boxes. The single box in the first layer is the last box of the linear chain. The $a_i - 1$ boxes in the last layer are connected to the blocks. We add edges between neighbouring layers such that the connector becomes rigid.
Furthermore, we enforce alignment of the lower boxes of the a_i blocks by adding further $a_i - 1$ boxes and adding edges from the lower boxes of the blocks to these boxes. Thus, the spacing constraint $s^{\text{box}}_{\text{box}}$ enforces a minimum width of the lower boxes of the blocks and, thereby, of the whole foot. In contrary the spacing constraint $s^{\text{port}}_{\text{corner}}$ enforces a maximum width. We have chosen the spacing constraints such that these two bounds are equal, i.e. the foot is horizontally rigid.

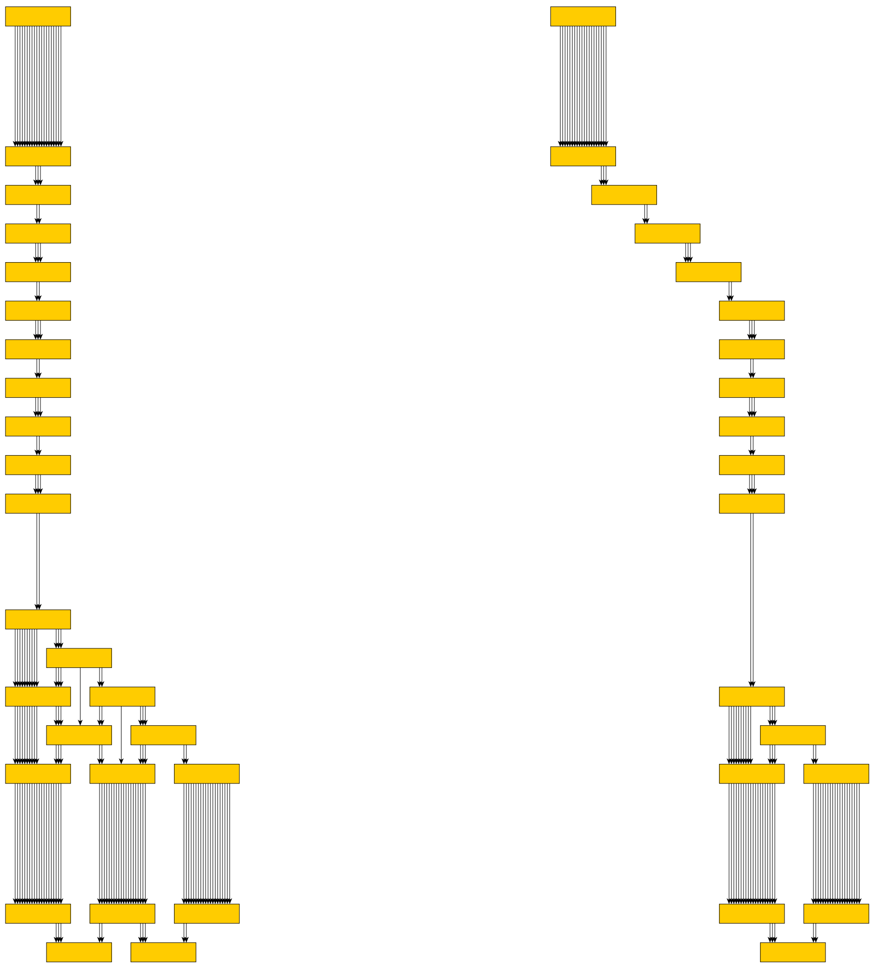

(a) The number gadget for $a = 3$.
(b) The number gadget for $a = 2$ shifted to the right by 2.

Figure 3.7: The number gadget.

The Synchroniser Gadgets

So far, we constructed a frame having m cells of width B and one number gadget for each $a_i \in A$. It remains to model the following two constraints: (i) there are exactly three number gadgets per cell and (ii) the three numbers corresponding to the three number gadgets in a cell sum up to B. In order to enforce these constraints, we add two synchroniser gadgets. We position one of them in the grouping box on the right side of the frame and another one on the left side. We replace the edge (r_2, r_3) by two edges. One of them goes from r_2 to the right synchroniser gadget and the other one from the synchroniser gadget to r_3. Thus, the synchroniser gadget needs to be positioned between r_2 and r_3. Analogously, we position the left synchroniser gadget. Later, we will connect each number gadget to both synchroniser. Thus, the number gadgets need to be positioned between the two grouping boxes, i.e. the number gadgets need to lie inside a cell of the frame. We connect them in such a way that constraint (i) and (ii) are enforced.

The purpose of a synchroniser gadget is to force a couple of blocks to have an overlapping y-range due to the upward drawing constraint. At first, we remove one of the 19 edges of each block. Instead we add one edge going from the top of each block to the synchroniser gadget and one edge from the synchroniser gadget to the bottom of each block. Then the synchroniser gadget defines a horizontal lines which needs to be crossed by all edges belonging to the blocks. We depict this situation in Figure 3.8a for the most simple case of a synchroniser gadget – a single box.

In order to synchronise r blocks, the synchroniser gadget needs an in- and out-degree of r, respectively. Thus, a single box serves only up to $r \leq 19$. However, for $r > 19$ we can extend the synchroniser gadget by adding four boxes. We add edges from two new boxes to the single box and edges from the single box to the remaining two new boxes. Then, the synchroniser gadget has both, an in-degree and an out-degree, of ≤ 38 (see Figure 3.8b). If $r > 38$ holds, we can extend the synchroniser gadget by adding further boxes (see Figure 3.8b) and so on.

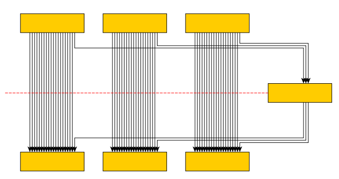
Now, we can enforce constraint (i) and (ii). We let the right synchroniser gadget synchronise the heads of all number gadgets as well as the $B - 3$ blocks in each cell. Thus, the heads of the number gadgets are forced into the three free slots of each cell and constraint (i) is satisfied. The left synchroniser gadget synchronises all the blocks in the foots of the number gadgets. This enforces, that the sum of the numbers corresponding to the number gadgets in one cell is at most B, i.e. constraint (ii).

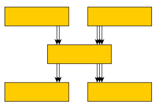
Since the synchronisers are placed inside grouping boxes and the synchronised blocks are not in these grouping box, the edges connecting the blocks and the synchroniser gadgets need to be bent two times. Thus, the two synchroniser gadgets together contribute $8mB$ bends to a proper layout.

The Complete Transformation

In Figure 3.9 we depict the transformed 3-Partition instance for the example instance $A = \{2, 2, 2, 2, 3, 3\}$, $B = 7$. The question is whether this instance can be layouted with at most $k = 4 + 8mB$ bends. Since we need at least 4 bends for the frame and $8mB$ bends for the synchronisation, this is the minimum required number of bends. If and only if there is a layout with exactly k bends, the original 3-Partition instance is solvable. We can reconstruct its solution by simply checking which number gadget lies in which cell of the frame and building up the sets S_i.

The remaining step of the proof is to show that the described transformation is polynomial in time. Therefore, we only need to bound the number of boxes created during the transformation. As the in- and out-degree of each box is at most 19 due to the spacing


(a) The most simple synchroniser gadget – a single box – and three synchronised blocks. The red dashed horizontal line needs to be crossed by all edges belonging to the synchronised blocks.


(b) The synchroniser gadget for $19 < mB \leq 38$.

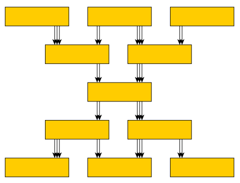

(c) The synchroniser gadget for $38 < mB \leq 57$.

Figure 3.8: The synchroniser gadget.

constraints the number of created edges is linear in the number of created boxes. We bound the number of boxes for the three constructed gadget types separately: (i) the frame, (ii) the number gadgets and (iii) the synchroniser gadgets.

(i) The upper and lower border of the frame consist of $2mB+1$ boxes each. The grouping boxes on the right and the left side contain 3 boxes each. Furthermore, there are $B-3$ blocks in each cell which. As each summand is linear in mB the number of all boxes in the frame is linear in mB as well.

(ii) There are $3m$ number gadgets each containing a block in the head and a linear chain of $2 \cdot \lceil 2/3 \cdot B \rceil + 1$ boxes. We further need to bound the number of boxes in the foot. There are a_i blocks and a connector containing $a_i \cdot (a_i - 1)$ boxes. Note that $a_i < B/2$ holds. Since each summand is in $\mathcal{O}(m^2 B^2)$, the number of boxes contributed by the number gadgets is a quadratic function in mB.

(iii) The number of boxes needed for the synchroniser gadgets that synchronise mB blocks is quadratic in mB as well.

Summing up all the terms we gain a function that is polynomial in mB. Since 3-Partition is $\mathcal{NP}$-complete in the strong sense, it is still $\mathcal{NP}$-complete if restricted to instances with B being polynomial in m. Thus, the presented transformation is polynomial in m as well and, therefore, Argument Map with $\beta > 0$ and $\alpha = \gamma = \delta = 0$ is $\mathcal{NP}$-hard. As it is easy to see that Argument Map $\in \mathcal{NP}$, we conclude that Argument Map with $\beta > 0$ and $\alpha = \gamma = \delta = 0$ is $\mathcal{NP}$-complete.

3.5.2 Argument Map is $\mathcal{NP}$-complete for $\alpha > 0$ and $\beta = \gamma = \delta = 0$

After we analysed the aspect of bend minimisation in the foregoing section, we now turn towards crossing minimisation and prove the following theorem:

Theorem 2. Argument Map *is $\mathcal{NP}$-complete for $\alpha > 0$ and $\beta = \gamma = \delta = 0$.*

We reduce the problem Upward Planarity Testing to this restricted version of Argument Map. Upward Planarity Testing is defined as follows:

Instance: A DAG $G = (V, A)$.
Question: Does there exist an upward drawing of G such that no pair of edges cross?

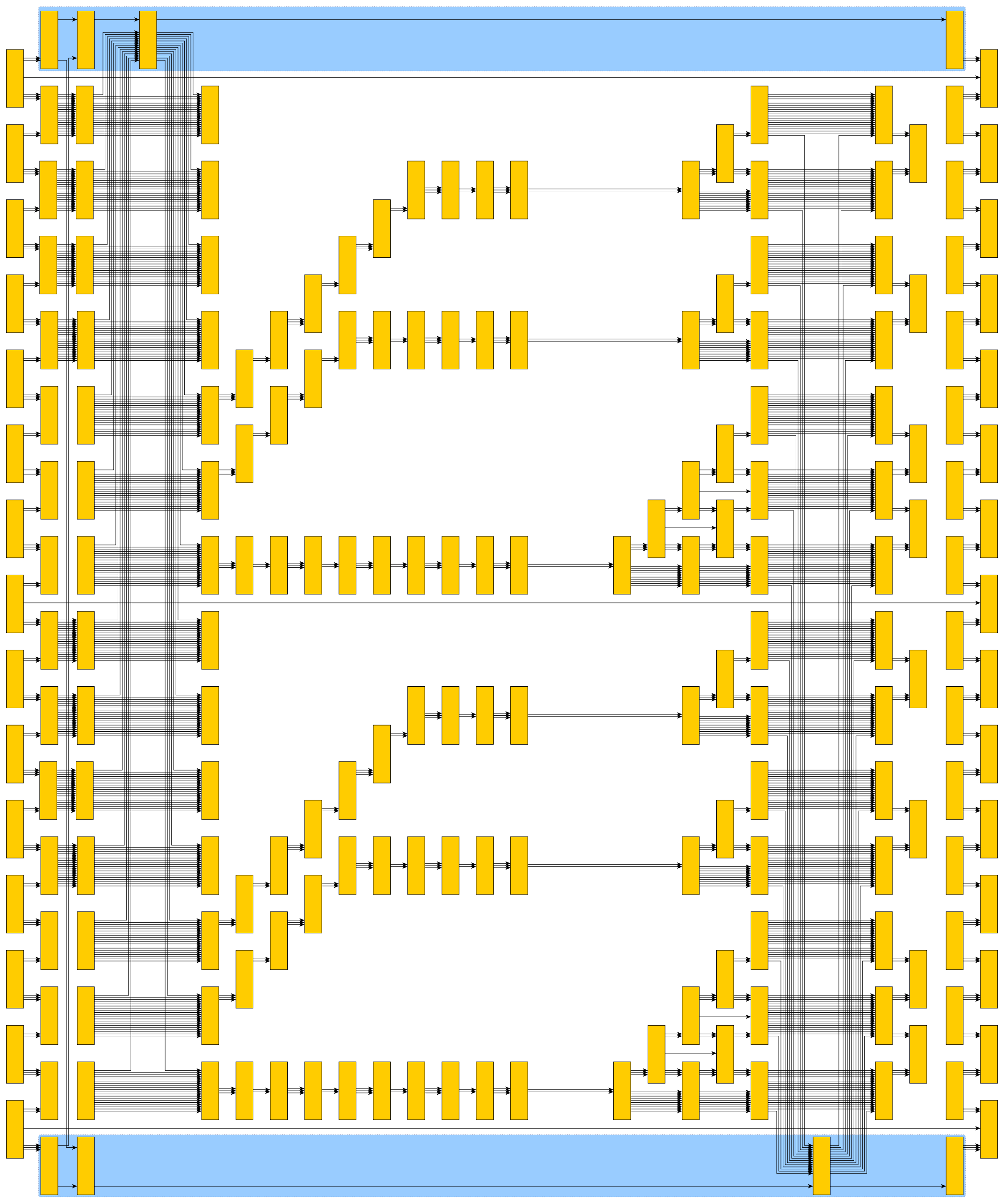


Figure 3.9: A 3-Partition instance ($A = \{2, 2, 2, 2, 3, 3\}, B = 7$) transformed to a Argument Map instance (rotated counterclockwise by 90 degrees).

Garg and Tamassia proved that Upward Planarity Testing is $\mathcal{NP}$-complete [GT94, GT02]. The transformation from an Upward Planarity Testing instance to an Argument Map instance is simple. Let G denote the input graph of an Upward Planarity Testing instance. We simply use G as input for the Argument Map instance. Furthermore, we set all box dimensions and all minimum spacings to zero and require no grouping. We assign α with an arbitrary positive value set and $\beta = \gamma = \delta = 0$. Furthermore, we set $k = 0$, i.e. we are looking for a layout free of crossings. Obviously, this transformation is polynomial. We now need to show, that the Argument Map instance has a solution if and only if G has an upward planar drawing.

It is easy to see, that G has an upward planar drawing, if the Argument Map instance can be solved. The solution of the Argument Map instance needs to satisfy the upward drawing constraint. Furthermore, since $\alpha > 0$ and $k = 0$, it is planar. Thus, the solution of the Argument Map instance is an upward planar layout of G.
Furthermore, we need to show that an upward planar drawing of G can be transformed to a solution of the corresponding Argument Map instance such that no crossings are added. Firstly, we modify the drawing of G by adding edges. We add edges from all sinks that lie in an internal face to the sink of the corresponding face. Analogously, we add edges from the face's source to the sources that lie within this face. Thereby, we gain an s-t-graph. Then, we can apply Algorithm 4 which we will introduce in Section 5.3.2. This algorithm assigns bends to the edges and fixes the x-coordinates of the boxes. Afterwards, we need to compute the y-coordinates according to a topological ordering of the s-t-graph. We ensure the alignment by moving nodes upward or downward. As last step, we delete the edges we added in order to get an s-t-graph. The resulting layout is still crossing-free and satisfies all properties required in Argument Map.

Thus, the Argument Map instance has a solution if and only if G is upward planar and, therefore, Argument Map is $\mathcal{NP}$-complete for $\alpha > 0$ and $\beta = \gamma = \delta = 0$.

3.5.3 Rectilinear Upward Drawing

In this section we show that a very basic graph drawing problem that is closely related to Argument Map is already $\mathcal{NP}$-complete. The problem is whether a DAG with maximum degree 4 has a rectilinear upward drawing, i.e. it can be layouted on a orthogonal grid without bends such that no edge is directed downwards. Firstly, we introduce this problem formally and denote it by Rectilinear Upward Drawing:

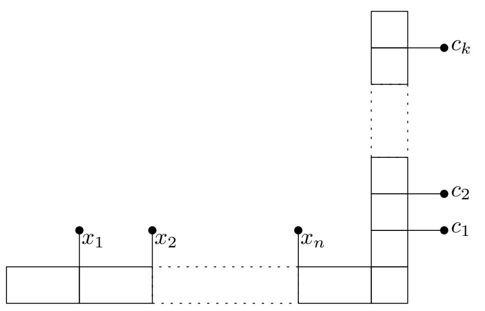
Instance: A DAG $G = (V, A)$ with maximum degree 4.
Question: Can G be embedded on an orthogonal grid such that no edge is bent and no edge is directed downwards?

Formann et al. discussed the same problem without the restriction on the edge directions. We denote their version by Rectilinear Drawing. They showed that Rectilinear Drawing is $\mathcal{NP}$-complete by reduction from 3-SAT [FHH$^+$90]. We firstly introduce the reader to the proof by Formann et al. and then modify it in order to prove the following theorem:

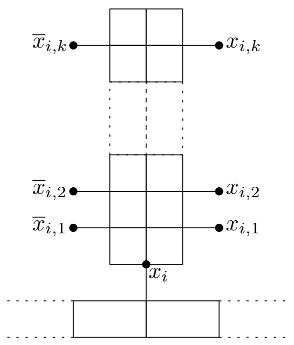
Theorem 3. Rectilinear Upward Drawing *is $\mathcal{NP}$-complete.*

In the following, we cite the proof by Formann et al. We transform a 3-SAT instance in three steps to a Rectilinear Drawing instance. Let S be an instance of 3-SAT with the variables $\{x_1, \ldots, x_n\}$ and the clauses $\{c_1, \ldots, c_k\}$. At first, we build up a *skeleton* with the shape of a lying L. The skeleton consists of two sequences of rectangular faces – one horizontal sequence of $n + 1$ faces and one vertical sequence having $k + 1$ faces. These two sequences are connected by another rectangular face that builds the corner of the L.

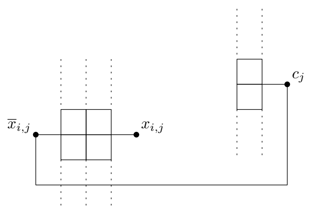
We attach a node for each x_i to the horizontal arm of the L and connect a node per c_i to the vertical arm. The skeleton is depicted in Figure 3.10a. Note that up to mirrorings, stretchings and rotations the depicted layout is the only valid layout of the skeleton. In the following we assume that the skeleton is oriented as shown in Figure 3.10a.
For each x_i we add a *tower gadget* that is shown in Figure 3.10b. The tower gadget consists of two parallel sequences of rectangular faces. We attach k nodes to each side of the tower and denote them by $x_{i,1}, \dots, x_{i,k}$ and $\overline{x}_{i,1}, \dots, \overline{x}_{i,k}$, respectively. The topology of the tower gadget enforces that all positive literals are on one side and all negated literals are on the other one. Since we connect the tower gadget and the skeleton only by a single edge, the tower gadget can be flipped vertically. The truth value of the corresponding variable x_i is determined by which literals are on which side. If the positive literals are on the right side, then x_i is interpreted to be true and otherwise, it needs to be false.
Up to now, it is not encoded which literal belongs to which clause. For each clause c_j we connect the node c_j with the nodes corresponding to the literals in the clause, e.g. for $c_j = \{x_k, \overline{x_l}, x_m\}$ we connect c_j with $x_{k,j}, \overline{x_{l,j}}$ and $x_{m,j}$ via an edge chain of length three. As shown in Figures 3.10c-3.10e there are three possibilities how an edge can be attached to the node c_j. The edge chains leaving c_j upwards or downwards can reach the right and the left hand side of a tower. However, the edge chain leaving c_j rightwards can only reach the right side of a tower. Thereby, we enforce that at least one of the three literals per clause evaluates to true.
Thus, if a valid layout has been computed, the solution of the 3-SAT instance can easily be reconstructed by looking at the orientation of the towers.



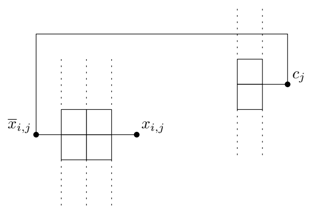
(a) The skeleton.



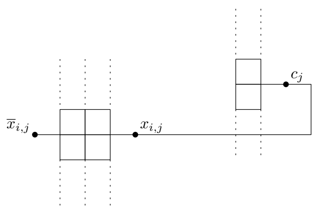
(b) The tower of x_i.



(c) Clause c_j and the left side of the tower of x_i are connected.



(d) Clause c_j and the left side of the tower of x_i are connected.

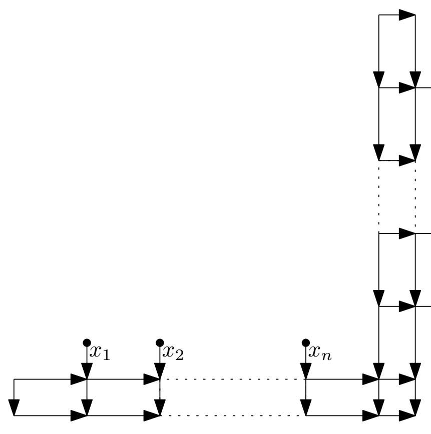


(e) Clause c_j and the right side of the tower of x_i are connected.

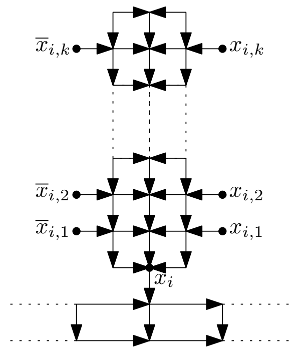
Figure 3.10: Illustration of the proof by Formann et al.

We now modify the proof by Formann et al. such that it holds for Rectilinear Upward Drawing as well. Therefore, we need to assign a direction to each edge such that the

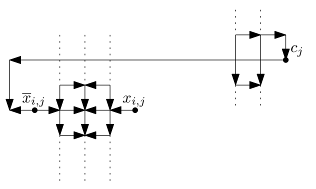
resulting graph remains acyclic. Furthermore, all drawings representing a valid solution of the 3-SAT instance need to be upward drawings. For the skeleton and the towers this can be done without problems as can be seen in Figures 3.11a and 3.11b. Problems arise when directing the edges in the edge chains connecting the nodes c_j and the associated literals. As the first edges of these edge chains may be attached at the top or at the bottom of c_j each direction assignment would conflict with the upward drawing constraint. Therefore, we attach the nodes c_j to the skeleton by using two edges instead of one. Now, this connection is attached to the top of c_j, i.e. the edge chains connecting the clauses and the literals can be attached at either the left, the bottom or the right of the nodes c_j. The possible edge chains after this modification are shown in Figures 3.11c-3.11e.
These modifications are sufficient in order to prove $\mathcal{NP}$-completeness of Rectilinear Upward Drawing.



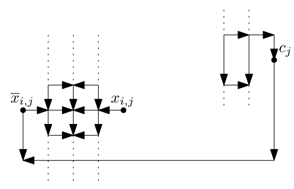
(a) The skeleton.



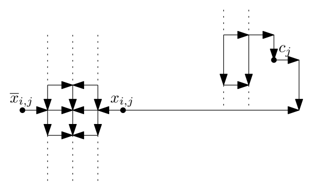
(b) The tower of x_i.



(c) Clause c_j and the left side of the tower of x_i are connected.



(d) Clause c_j and the left side of the tower of x_i are connected.



(e) Clause c_j and the right side of the tower of x_i are connected.

Figure 3.11: Illustration of the modified proof.

3.5.4 Directed Visibility Representation

In the previous section we discussed whether the problem of embedding a DAG with maximum degree 4 in a rectilinear way such that no edge is directed downwards is $\mathcal{NP}$-complete. In this section we consider a closely related problem using the visibility representation. When using the visibility representation, we draw nodes as horizontal lines and there exists an edge between two nodes, if and only if a vertical line can be drawn between two nodes such that no other node is crossed. We denote this problem by Directed Visibility Representation:

Instance: A DAG $G = (V, A)$.
Question: Does there exist a drawing of G in the visibility representation such that all edges are directed upwards?

Theorem 4. Directed Visibility Representation *is $\mathcal{NP}$-complete.*

We prove Theorem 4 by showing that it is equivalent to an $\mathcal{NP}$-complete problem. Di Battista and Tamassia showed that a graph admits a directed visibility representation, if and only if it admits an upward planar drawing [DT88]. Thus, Directed Visibility Representation turns out to be equivalent to the problem Upward Planarity Testing of which Garg and Tamassia proved that it is $\mathcal{NP}$-complete [GT94, GT02]. Thus, Directed Visibility Representation is $\mathcal{NP}$-complete. However, Bertolazzi et al. found a polynomial algorithm if the embedding of G is fixed [BDLM94].

3.6 Concluding Remarks

In this chapter we presented the formal problem statement Argument Map and discussed the constraints on layouts of argument maps as well as the pursued optimisation goals. In the remaining part of this work we present two approaches to automatic layout of argument maps: (i) an ILP-based approach and (ii) a topology-shape-metrics approach.

ILP Approach

In a first phase we modelled the layout computation as ILP problem. The advantage of ILPs is that one can easily add or omit a single type of constraints. Thereby, we could test a variety of different layouts with only little effort. Having first automatic layouts at hand it is easier to specify the constraints and optimisation goals of a layout. Thus, the ILP approach was an important step during the requirements definition. As the ILP approach was developed in an early phase of this work and some requirements came up later it is not complete with respect to the properties in Section 3.2 and the optimisation goals in Section 3.3. Furthermore, it is not made for practical use due to efficiency reasons. Running on a low-level computer of an end-user the computation times would exceed any feasible values.

Topology-Shape-Metrics Approach

After defining the requirements using the ILP approach we turned towards an approach that yields an efficient algorithm yielding good results. We thereby build on the well known topology-shape-metrics framework of which we give an overview in Section 5.1. We dedicate one of the Sections 5.2-5.4 to each of the three steps in the framework. Within this framework we only heuristically optimise the mentioned criteria. Nevertheless, we get resulting layouts of a high quality. In Chapter 6 we discuss these layouts and identify possibilities for further improvements.

4. ILP Approach

The first algorithm we present for layouts of argument maps is based on an integer linear program (ILP). The main advantage of ILPs is that they are modular. One can easily add more constraints to the ILP or omit existing ones, i.e. one can easily modify which properties of the layout shall be enforced and which criteria are to be optimised. Therefore, the ILP approach was important for carving out the formal problem statement Argument Map (see Section 3.4). Since the ILP approach was developed in an early phase of this work and some requirements came up later, it is not complete with respect to the properties in Section 3.2 and the optimisation goals in Section 3.3.
Furthermore, this approach is not efficient enough for practical use. For small instances up to 15 nodes, optimal solutions can be computed within a feasible time. For larger instances a feasible solution is found in most cases. However, the ILP solver does not terminate with the optimal solution within the given time. Thus, we were not able to compute gold standard layouts for all argument maps. Nevertheless, the computed layouts give an impression of the different constraints that should be taken into the formal problem statement.

In Section 4.1 we describe how to build an ILP for a given argument map and, afterwards, discuss the generated layouts in Section 4.2. We conclude this chapter with a discussion how the missing properties and optimisation goals can be added to the ILP in Section 4.3.

4.1 ILP Construction

In this section we describe the construction of an ILP instance for a given argument map. Firstly, we discuss some preliminaries to the construction. Then we describe how some of the properties stated in Section 3.2 can be modelled as linear inequalities. However, we do not model all of them – alignment of predecessors as well as free sources and sinks are missing. Furthermore, we do not distinguish between the two minimum spacings $s_{\text{edge}}^{\text{edge}}$ and $\hat{s}_{\text{edge}}^{\text{edge}}$ between a pair of edges. The same holds for the optimisation goals described in Section 3.3. In this approach we only optimise the total edge length and the number of bends.
Furthermore, we restrict the edges to have at most four bends. In Section 5.3.1 we show that two bends per edge are sufficient. Nevertheless, we allow four bends, because layouts with four bends per edge are easier to perceive. Additionally, to the properties described in Section 3.2 we enforce a maximum layout size and so called "path layout".
After describing all constraints of the ILP we present the target function that is to be

minimised. We, furthermore, improve the performance of the ILP solver by giving an initial solution.

Preliminaries

In the ILP approach we model all boxes and edges using points. A point `p` is a tuple of two coordinates `p.x` and `p.y`. These ILP variables are integer variables. Thereby, we simulate the embedding on a grid. Note that the coordinate system is oriented as usual in computer drawing applications – the y-axis points downwards. Beside the integer variables representing the coordinates, there are many binary variables in the domain $\{0, 1\}$. We interpret 0 as `false` and 1 as `true`.

Throughout the ILP we use the concept of *relative position*s from one point on the grid to another one. By a relative position we denote a set of four binary variables. They are named `upwards`, `downwards`, `rightwards` and `leftwards`. We denote the relative position from point `p` to point `q` by $\texttt{rp}(\texttt{p}, \texttt{q})$. Obviously, this may introduce semantically equivalent variables if we work with the relative positions from `p` to `q` as well as from `q` to `p`. In this case we add equality constraints for semantically equivalent variables. Note that due to relative positions the coordinates of `p` and `q` are not related with the variables of $\texttt{rp}(\texttt{p}, \texttt{q})$. Since the minimum distances between a pair of points depends on the type of graphical item the points belong to, we will later come back to this point.

The Boxes

Each box `b` is represented by two points $\texttt{b}^{\nwarrow}$ and $\texttt{b}^{\searrow}$, which form the top left and the bottom right corner of the rectangle that represents the box. The vertical and horizontal distances between $\texttt{b}^{\nwarrow}$ and $\texttt{b}^{\searrow}$ are fixed according to the dimensions of `b` specified in the input data by the following two constraints:

$$\begin{aligned}\texttt{b}^{\searrow}.\texttt{x} - \texttt{b}^{\nwarrow}.\texttt{x} &= w \\ \texttt{b}^{\searrow}.\texttt{y} - \texttt{b}^{\nwarrow}.\texttt{y} &= h(\texttt{b})\end{aligned}$$

In order to forbid overlapping of boxes we use the relative positions. Here, we describe how to enforce disjointness for a pair of boxes `b` and `c`. We can express the relative positions of the two boxes `b` and `c` using the relative positions $\texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow})$ and $\texttt{rp}(\texttt{b}^{\searrow}, \texttt{c}^{\nwarrow})$. For example, box `b` is right of `c` if $\texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow}).\texttt{leftwards} = 1$. For disjointness of `b` and `c` we need the constraint that `b` is to at least one side of `c` which is expressed in the following constraint:

$$\begin{aligned}\texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow}).\texttt{leftwards} + \texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow}).\texttt{upwards} &\;+ \\ \texttt{rp}(\texttt{b}^{\searrow}, \texttt{c}^{\nwarrow}).\texttt{rightwards} + \texttt{rp}(\texttt{b}^{\searrow}, \texttt{c}^{\nwarrow}).\texttt{downwards} &\;\geq\; 1\end{aligned}$$

As already mentioned in the paragraph about relative positions this constraint does not yet enforce the boxes to be disjoint. We furthermore need a constraint that relates the relative positions $\texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow})$ and $\texttt{rp}(\texttt{b}^{\searrow}, \texttt{c}^{\nwarrow})$ with the coordinates of the four points $\texttt{b}^{\nwarrow}$, $\texttt{b}^{\searrow}$, $\texttt{c}^{\nwarrow}$ and $\texttt{c}^{\searrow}$:

$$\begin{aligned}\texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow}).\texttt{leftwards} = 1 &\implies \texttt{b}^{\nwarrow}.\texttt{x} \geq \texttt{c}^{\searrow}.\texttt{x} + s^{\text{box}}_{\text{box}} && (4.1)\\ \texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow}).\texttt{upwards} = 1 &\implies \texttt{b}^{\nwarrow}.\texttt{y} \geq \texttt{c}^{\searrow}.\texttt{y} + s^{\text{box}}_{\text{box}} && (4.2)\\ \texttt{rp}(\texttt{b}^{\searrow}, \texttt{c}^{\nwarrow}).\texttt{rightwards} = 1 &\implies \texttt{b}^{\searrow}.\texttt{x} + s^{\text{box}}_{\text{box}} \leq \texttt{c}^{\nwarrow}.\texttt{x} && (4.3)\\ \texttt{rp}(\texttt{b}^{\searrow}, \texttt{c}^{\nwarrow}).\texttt{downwards} = 1 &\implies \texttt{b}^{\searrow}.\texttt{y} + s^{\text{box}}_{\text{box}} \leq \texttt{c}^{\nwarrow}.\texttt{y} && (4.4)\end{aligned}$$

Note that it is sufficient to use an implication in these four constraints, because we force one of the four variables on the left hand sides to be assigned with 1. To enforce equivalence between the left hand and the right hand sides would work as well. However, then the transformation to linear inequalities would be more complicated. We show how the four constraints can be formulated as a linear inequality by using $\texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow})\texttt{.leftwards}$ as an example:

$$\texttt{b}^{\nwarrow}\texttt{.x} - \texttt{c}^{\searrow}\texttt{.x} - \texttt{MAX} \cdot \texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow})\texttt{.leftwards} \geq -\texttt{MAX} + s_{\text{box}}^{\text{box}}$$

Hereby, $\texttt{MAX}$ is an integer bigger than any other value that might occur in the ILP. In the case $\texttt{rp}(\texttt{b}^{\nwarrow}, \texttt{c}^{\searrow})\texttt{.leftwards} = 1$ this inequality evaluates to $\texttt{b}^{\nwarrow}\texttt{.x} - \texttt{c}^{\searrow}\texttt{.x} \geq s_{\text{box}}^{\text{box}}$. Otherwise, it evaluates to $\texttt{b}^{\nwarrow}\texttt{.x} - \texttt{c}^{\searrow}\texttt{.x} \geq -\texttt{MAX} + s_{\text{box}}^{\text{box}}$, i.e. $\texttt{b}^{\nwarrow}\texttt{.x} - \texttt{c}^{\searrow}\texttt{.x}$ is unconstrained.

The Edges

We model the edges as a sequence of six points on the grid. The first point is located on the boundary of the source box, whereas the last point is on the boundary of the target box. The four points in between represent the bends of the edges, i.e. we restrict the layouts to have at most four bends per edge. In Section 5.3.1 we show that two bends per edge are sufficient. Nevertheless, we allow four bends, because layouts with four bends per edge are easier to perceive. We require that edges are layouted orthogonally and no edge segment is directed upwards.
In the ILP an edge $\texttt{e}$ is associated with the following variables:

- The six coordinate pairs $\texttt{p}(\texttt{e}, \texttt{i})$ for $\texttt{i} = 0, \ldots, 5$ representing the start point, the bends and the target point of $\texttt{e}$.
- Four binary variables `upwards`, `downwards`, `rightwards` and `leftwards` per edge segment $\texttt{s}(\texttt{e}, \texttt{i})$ for $\texttt{i} = 0, \ldots, 4$ that determine the direction of the corresponding edge segment.
- One integer variable $\texttt{l}(\texttt{e}, \texttt{i})$ for $\texttt{i} = 0, \ldots, 4$ per edge segment corresponding to its length.
- One binary variable $\texttt{b}(\texttt{e}, \texttt{i})$ for $\texttt{i} = 0, \ldots, 3$ per possible bend, i.e. for the four points $\texttt{p}(\texttt{e}, 1), \ldots, \texttt{p}(\texttt{e}, 4)$. These variable are used for counting the bends.

Since the edges shall be layouted orthogonally we have some redundancy in the above mentioned variables. We restrict the edge segments to alternate between vertical and horizontal such that the first segment is a vertical one. Thus, we can set the direction of edge segments $0, 2$ and 4 to `downwards` as no upward directed segments are allowed and the directions of segments 1 and 3 are either `leftwards` or `rightwards`.

$$\begin{aligned}
\forall \texttt{i} \in \{0, 2, 4\} &: \texttt{s}(\texttt{e}, \texttt{i})\texttt{.downwards} = 1 \\
\forall \texttt{i} \in \{0, 2, 4\} &: \texttt{s}(\texttt{e}, \texttt{i})\texttt{.upwards} = \texttt{s}(\texttt{e}, \texttt{i})\texttt{.leftwards} = \texttt{s}(\texttt{e}, \texttt{i})\texttt{.rightwards} = 0 \\
\forall \texttt{i} \in \{1, 3\} &: \texttt{s}(\texttt{e}, \texttt{i})\texttt{.leftwards} + \texttt{s}(\texttt{e}, \texttt{i})\texttt{.rightwards} = 1 \\
\forall \texttt{i} \in \{1, 3\} &: \texttt{s}(\texttt{e}, \texttt{i})\texttt{.upwards} = \texttt{s}(\texttt{e}, \texttt{i})\texttt{.downwards} = 0
\end{aligned}$$

Furthermore, we need to relate the directions of the edge segments with the coordinates of the incident points. Therefore, we need the following two sets of constraints:

$$\begin{aligned}
\forall \texttt{i} \in \{0, 2, 4\} &: \texttt{p}(\texttt{e}, \texttt{i} + 1)\texttt{.y} \geq \texttt{p}(\texttt{e}, \texttt{i})\texttt{.y} \\
\forall \texttt{i} \in \{0, 2, 4\} &: \texttt{p}(\texttt{e}, \texttt{i} + 1)\texttt{.x} = \texttt{p}(\texttt{e}, \texttt{i})\texttt{.x}
\end{aligned}$$

$$\begin{aligned}
\forall \texttt{i} \in \{1, 3\} &: \texttt{p}(\texttt{e}, \texttt{i})\texttt{.y} = \texttt{p}(\texttt{e}, \texttt{i} + 1)\texttt{.y} \\
\forall \texttt{i} \in \{1, 3\} &: \texttt{s}(\texttt{e}, \texttt{i})\texttt{.rightwards} = 1 \Longrightarrow \texttt{p}(\texttt{e}, \texttt{i} + 1)\texttt{.x} \geq \texttt{p}(\texttt{e}, \texttt{i})\texttt{.x} \\
\forall \texttt{i} \in \{1, 3\} &: \texttt{s}(\texttt{e}, \texttt{i})\texttt{.leftwards} = 1 \Longrightarrow \texttt{p}(\texttt{e}, \texttt{i})\texttt{.x} \geq \texttt{p}(\texttt{e}, \texttt{i} + 1)\texttt{.x}
\end{aligned}$$

Since the edge segments $\texttt{s}(\texttt{e},\texttt{i})$ for $i = 0, 2, 4$ are directed downwards we can directly impose constraints on the coordinates of the incident points. They need to have the same x-coordinate and the y-coordinate of $\texttt{p}(\texttt{e},\texttt{i}+1)$ must be greater than or equal to the y-coordinate of $\texttt{p}(\texttt{e},\texttt{i})$. Thus, edge segments may have zero length.
For the edge segments $\texttt{s}(\texttt{e},\texttt{i})$ for $i = 1, 3$ we only know that they are horizontal and do not know whether they are directed rightwards or leftwards. Thus, we can directly enforce equal y-coordinates of the incident points. However, we constrain their x-coordinates depending on $\texttt{s}(\texttt{e},\texttt{i}).\texttt{rightwards}$ and $\texttt{s}(\texttt{e},\texttt{i}).\texttt{leftwards}$ by the last two constraints. These constraints are an implication with an inequality as implicature. We have already seen how to formulate this type of constraint as linear inequality in the context of non-overlapping boxes.

As already mentioned, we require the end points of edges to be on the border of the source or target box, respectively. Incoming edges attach at the top of the box and outgoing edges attach at the bottom. Let `s` be the box corresponding to the source of edge `e` and let `t` be its target box. For the y-coordinates of the first and last point we use two simple constraints:

$$\texttt{p}(\texttt{e},0).\texttt{y} = \texttt{s}^{\searrow}.\texttt{y}$$
$$\texttt{p}(\texttt{e},5).\texttt{y} = \texttt{t}^{\nwarrow}.\texttt{y}$$

However, the x-coordinates are not fixed. They may vary in the x-range of the corresponding box. The minimum distance $s^{\text{port}}_{\text{corner}}$ between the port and the corner of the box is required:

$$\texttt{s}^{\nwarrow}.\texttt{x} + s^{\text{port}}_{\text{corner}} \leq \texttt{p}(\texttt{e},0).\texttt{x} \leq \texttt{s}^{\searrow}.\texttt{x} - s^{\text{port}}_{\text{corner}}$$
$$\texttt{t}^{\nwarrow}.\texttt{x} + s^{\text{port}}_{\text{corner}} \leq \texttt{p}(\texttt{e},5).\texttt{x} \leq \texttt{t}^{\searrow}.\texttt{x} - s^{\text{port}}_{\text{corner}}$$

We do not discuss the variables $\texttt{l}(\texttt{e},\texttt{i})$ and $\texttt{b}(\texttt{e},\texttt{i})$ right now but postpone their discussion to the paragraphs "Total Edge Length Minimisation" and "Bend Minimisation".

Downward Arcs

According to the constraints presented in the foregoing paragraph, the target of an edge can only be positioned below the source of the edge. This is due to the following constraints:

- No upward directed edge segments are allowed.
- The first and the last segment of an edge are directed downwards.

However, additionally constraining the relative position of two boxes that are connected via an edge turned out to improve the performance of the ILP approach. For each edge $\texttt{e} = (\texttt{s},\texttt{t})$ we add the following constraint:

$$\texttt{rp}(\texttt{s}^{\searrow},\texttt{t}^{\nwarrow}).\texttt{downwards} = 1$$

Overlapping of Edges and Boxes

In order to avoid edges that cut through boxes we need two types of constraints: (i) edge points may not be inside boxes and (ii) the points incident to an edge segment may not be on opposite sides of a box.
Constraint (i): The concept of how to avoid that an edge point $\texttt{p}(\texttt{e},\texttt{i})$ is inside of a box `b` is already familiar to the reader:

$$\begin{aligned}\texttt{rp}(\texttt{p}(\texttt{e},\texttt{i}),\texttt{b}^{\nwarrow}).\texttt{downwards} + \texttt{rp}(\texttt{p}(\texttt{e},\texttt{i}),\texttt{b}^{\nwarrow}).\texttt{rightwards} &\; + \\ \texttt{rp}(\texttt{p}(\texttt{e},\texttt{i}),\texttt{b}^{\searrow}).\texttt{upwards} + \texttt{rp}(\texttt{p}(\texttt{e},\texttt{i}),\texttt{b}^{\searrow}).\texttt{leftwards} &\; \geq \; 1\end{aligned}$$

Furthermore, we need the spacing constraints for each direction. They enforce a distance of at least $s_{\text{box}}^{\text{edge}}$ between the edge point and the border of box. For example, we present the `downwards` case:

$$\mathtt{rp(p(e,i),b^{\nwarrow}).downwards} = 1 \Longrightarrow \mathtt{b^{\nwarrow}.y} \geq \mathtt{p(e,i).y} + s_{\text{box}}^{\text{edge}}$$

Constraint (ii): Up to now it is possible that two edge points lie on opposite sides of a box and the segment connecting them cuts through the box. We now formulate constraints such that a vertical edge segment that starts above a box needs to end above this box as well. We do not treat vertical segments that start below a box. Since they are directed downward they cannot cut through this box.
For horizontal segments we enforce the following: If a segment starts on the left (right) side of a box, then it needs to end on the left (right) side of this box as well. Thus, the segment cannot cut through the box.

$$\begin{aligned}
\forall \mathtt{i} \in \{0,2,4\} : (\mathtt{rp(p(e,i),b^{\nwarrow}).downwards} = 1 \quad &\wedge \\
\mathtt{rp(p(e,i),b^{\nwarrow}).rightwards} = 0 \quad &\wedge \\
\mathtt{rp(p(e,i),b^{\searrow}).leftwards} = 0) \quad &\Longrightarrow \quad \mathtt{rp(p(e,i+1),b^{\nwarrow}).downwards} = 1
\end{aligned}$$

$$\begin{aligned}
\forall \mathtt{i} \in \{1,3\} : (\mathtt{rp(p(e,i),b^{\searrow}).leftwards} = 1 \quad &\wedge \\
\mathtt{rp(p(e,i),b^{\nwarrow}).downwards} = 0 \quad &\wedge \\
\mathtt{rp(p(e,i),b^{\searrow}).upwards} = 0) \quad &\Longrightarrow \quad \mathtt{rp(p(e,i+1),b^{\searrow}).leftwards} = 1
\end{aligned}$$

$$\begin{aligned}
\forall \mathtt{i} \in \{1,3\} : (\mathtt{rp(p(e,i),b^{\nwarrow}).rightwards} = 1 \quad &\wedge \\
\mathtt{rp(p(e,i),b^{\nwarrow}).downwards} = 0 \quad &\wedge \\
\mathtt{rp(p(e,i),b^{\searrow}).upwards} = 0) \quad &\Longrightarrow \quad \mathtt{rp(p(e,i+1),b^{\nwarrow}).rightwards} = 1
\end{aligned}$$

These constraints are all of the same type:

$$\mathtt{a} = 1 \wedge \mathtt{b} = 0 \wedge \mathtt{c} = 0 \Longrightarrow \mathtt{d} = 1$$

We reformulate them as linear inequality in the following way:

$$\mathtt{d} - \mathtt{a} + \mathtt{b} + \mathtt{c} \geq 0$$

We do not need to add disjointness constraints for all pairs of edges and boxes. Due to the vertical ordering implied by the edge directions some pairs of edges and boxes cannot overlap. Therefore, we first compute the transitive hull of the partial ordering represented by the edges using the Floyd-Warshall-algorithm. Afterwards we can easily check whether an edge and a box might overlap. We only add disjointness constraints between a box and an edge if the box is in the ordering neither before the edge's source nor after the edge's target.

Overlapping of Edges

To avoid overlapping of edges is the most complex set of constraints we use in the ILP approach. In this paragraph we describe how to avoid overlapping between two edges `e` and `f`. At first, we introduce the relative positions between each pair of points where one point belongs to `e` and the other one to `f`. For each relative position we add the constraint that at least one of the binary variables is true:

$$\begin{aligned}
\forall \mathtt{i},\mathtt{j} \in \{0,\ldots,5\} \quad : \quad & \mathtt{rp(p(e,i),p(f,j)).upwards} + \\
& \mathtt{rp(p(e,i),p(f,j)).downwards} + \\
& \mathtt{rp(p(e,i),p(f,j)).rightwards} + \\
& \mathtt{rp(p(e,i),p(f,j)).leftwards} \geq 1
\end{aligned}$$

Again, we need the spacing constraints in order to relate the relative positions with the real coordinates. In this approach we do not distinguish the different kinds of spacing between a pair of edges described in Section 3.2 but use $s^{\text{edge}}_{\text{edge}}$ as minimum spacing between all pairs of edges. For example, we present the `downwards` case:

$$\texttt{rp(p(e,i),p(f,j)).downwards} = 1 \Longrightarrow \texttt{p(f,j).y} \geq \texttt{p(e,i).y} + s^{\text{edge}}_{\text{edge}}$$

Since the edge segments are orthogonally a lot of these binary variables are redundant. For a vertical (horizontal) segment the incident points have always the same horizontal (vertical) relative position to all other points:

$$\begin{aligned}
\forall \texttt{i} \in \{0,2,4\}, \texttt{j} \in \{0,\ldots,5\} &: \texttt{rp(p(e,i),p(f,j)).leftwards}\\
&= \texttt{rp(p(e,i+1),p(f,j)).leftwards}\\
\forall \texttt{i} \in \{0,2,4\}, \texttt{j} \in \{0,\ldots,5\} &: \texttt{rp(p(e,i),p(f,j)).rightwards}\\
&= \texttt{rp(p(e,i+1),p(f,j)).rightwards}
\end{aligned}$$

$$\begin{aligned}
\forall \texttt{i} \in \{1,3\}, \texttt{j} \in \{0,\ldots,5\} &: \texttt{rp(p(e,i),p(f,j)).downwards}\\
&= \texttt{rp(p(e,i+1),p(f,j)).downwards}\\
\forall \texttt{i} \in \{1,3\}, \texttt{j} \in \{0,\ldots,5\} &: \texttt{rp(p(e,i),p(f,j)).upwards}\\
&= \texttt{rp(p(e,i+1),p(f,j)).upwards}
\end{aligned}$$

Note that we need to add these constraints with `e` and `f` being exchanged as well. Finally, we can add the constraints that no edge point of `f` lies on an edge segment of `e`:

$$\begin{aligned}
\forall \texttt{i} \in \{0,2,4\}, \texttt{j} \in \{0,\ldots,5\} &: \texttt{rp(p(f,j),p(e,i)).downwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i+1)).upwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i)).leftwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i)).rightwards} = 1
\end{aligned}$$

$$\begin{aligned}
\forall \texttt{i} \in \{1,3\}, \texttt{j} \in \{0,\ldots,5\} &: \texttt{s(e,i).rightwards} = 1\\
&\Longrightarrow (\texttt{rp(p(f,j),p(e,i)).rightwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i+1)).leftwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i)).upwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i)).downwards} = 1)
\end{aligned}$$

$$\begin{aligned}
\forall \texttt{i} \in \{1,3\}, \texttt{j} \in \{0,\ldots,5\} &: \texttt{s(e,i).leftwards} = 1\\
&\Longrightarrow (\texttt{rp(p(f,j),p(e,i)).leftwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i+1)).rightwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i)).upwards} = 1\\
&\vee\ \texttt{rp(p(f,j),p(e,i)).downwards} = 1)
\end{aligned}$$

Again, we need these constraints as well with exchanged roles of `e` and `f`. The second and third constraint are of a new type we did not handle yet:

$$\texttt{a} = 1 \Longrightarrow (\texttt{b} = 1 \vee \texttt{c} = 1 \vee \texttt{d} = 1 \vee \texttt{e} = 1)$$

We transform them to the following linear inequality:

$$-\texttt{a} + \texttt{b} + \texttt{c} + \texttt{d} + \texttt{e} \geq 0$$

Maximum Layout Size

Additionally to the properties described in Section 3.2, we use constraints for bounding the maximum layout size which is given by a maximum width and a maximum height. For boxes we only need to enforce that the bottom right point is inside the rectangle $[0, \texttt{max\_width}] \times [0, \texttt{max\_height}]$:

$$\begin{aligned} \texttt{b}^{\searrow}\texttt{.x} &\leq \texttt{max\_width} \\ \texttt{b}^{\searrow}\texttt{.y} &\leq \texttt{max\_height} \end{aligned}$$

In the case of edges we need to distinguish the x- and y-coordinates. For the x-coordinates we give the bound for all bend points. As the first and the last point are on the boundary of the source or target box, respectively, their x-coordinate is in the interval $[0, \texttt{max\_width}]$ due to the constraints we just stated.

$$\forall \texttt{i} \in \{1, \dots, 4\} : \texttt{b(e,i).x} \leq \texttt{max\_width}$$

For the y-coordinates it would be sufficient to bound the y-coordinate of the last point as it is the point with the highest y-coordinate. However, again the corresponding point is on the boundary of the edge's target box and, therefore, it is within the prescribed rectangle.

Path Layout

While carving out the formal problem statement, we tested another layout property which is not part of the problem Argument Map. Let `b` and `c` be boxes such that `b` has a single outgoing edge `e` that leads to `c`. Furthermore, let `e` be the only incoming edge of `c`. We say that `b` and `c` together with `e` form a *path*. During the development of the ILP-based algorithm we found out that layouts look more convenient if a path is layouted as a straight vertical line and the edge ports are centred. Therefore, we add constraints that enforce equal x-coordinates for the two boxes `b` and `c` and requires the connecting edge to be centred horizontally:

$$\begin{aligned} \texttt{b}^{\nwarrow}\texttt{.x} &= \texttt{c}^{\nwarrow}\texttt{.x} \\ \forall \texttt{i} \in \{0, \dots, 5\} : \texttt{p(e,i).x} &= \texttt{b}^{\nwarrow}\texttt{.x} + \lfloor w/2 \rfloor \end{aligned}$$

Furthermore, we require all inner points to be the same. As we already imposed the constraints for the x-coordinates we only need to add constraints for equal y-coordinates:

$$\forall \texttt{i} \in \{2, \dots, 4\} : \texttt{p(e,i).y} = \texttt{p(e,1).y}$$

Grouping

Grouping of boxes needs only one new type of constraint. Nevertheless, it is a complex operation as many of the disjointness constraints described above can be disposed after grouping and some new disjointness constraints need to be added. The new type of constraint is that a box `b` should lie inside a grouping box `c`:

$$\begin{aligned} \texttt{b}^{\searrow}\texttt{.x} &\leq \texttt{c}^{\searrow}\texttt{.x} \\ \texttt{c}^{\nwarrow}\texttt{.x} &\leq \texttt{b}^{\nwarrow}\texttt{.x} \\ \texttt{b}^{\searrow}\texttt{.y} &\leq \texttt{c}^{\searrow}\texttt{.y} \\ \texttt{c}^{\nwarrow}\texttt{.y} &\leq \texttt{b}^{\nwarrow}\texttt{.y} \end{aligned}$$

We now describe how the bounding boxes defined by a tree T (see Section 3.2) can be enforced in the layout. For each grouping node of T we introduce a new grouping box

to the ILP. In the following we do not distinguish between a node in the tree and the corresponding box in the layout. The grouping boxes will not be visible in the final layout but help to model the grouping of the boxes. For all children $\mathtt{b_i}$ of a grouping node $\mathtt{g}$ in T we require that the box $\mathtt{b_i}$ is within the grouping box $\mathtt{g}$ by adding the above set of constraint with $\mathtt{b} = \mathtt{b_i}$ and $\mathtt{c} = \mathtt{g}$. Furthermore, we require that all boxes that are not forced to be in the grouping box do not overlap it. Of course, we then can dispose some of the box disjointness constraints we already added, i.e. we do not need the disjointness constraints between all $\mathtt{b_i}$ and all boxes outside of $\mathtt{g}$ anymore.
Additionally, we add disjointness constraints between the grouping box and all edges that neither have their source nor their sink inside the grouping box. Afterwards, we can dispose the disjointness constraints between these edges and the boxes inside the grouping box.
Furthermore, we can simplify the constraints for disjointness of edges. We dispose the disjointness constraints for pairs of edges where one edge has source and sink inside the grouping box and the other edge has source and sink outside the grouping box.

The grouping feature was introduced for two purposes: (i) it can visualise semantic groups that exist in the input data and are defined using the input parameter T as described in Chapter 3 and (ii) it was supposed to speed up the layout computation because it reduces the number of constraints.
For the case, that the user does not provide a grouping structure T, we present an algorithm that computes one. The grouping our algorithm computes depends on the articulation points of the graph. If there is at least one articulation point in the graph one of them is selected and removed from the graph. Then the graph breaks into at least two connected components. For each connected component a grouping box is added and afterwards the components are treated recursively.
An instance might get insolvable if the algorithm uses an articulation point that is part of a path. Assume, that the articulation point is the source (sink) of a path. Let C denote the connected component containing the sink (source) of the path after the removal of the articulation point. If there exists a directed path from C to the articulation point as well as a directed path from the articulation point to C (as shown in Figure 4.1), then the articulation point needs to be position to the left or the right of the grouping box of C. Thus, it cannot be positioned above (below) the sink (source) of the path.
In order to avoid this, the algorithm simply does not use articulation points that are part of a path. However, the selection of the articulation points is still not fully determined. Hence, we suggest different strategies to select them. These strategies apply different centrality measures: (i) highest closeness first [OAS10], (ii) highest node betweenness first [OAS10], (iii) highest degree first and (iv) lowest degree first. In Section 4.2 we discuss their empirical impact on the runtime of the ILP solver.

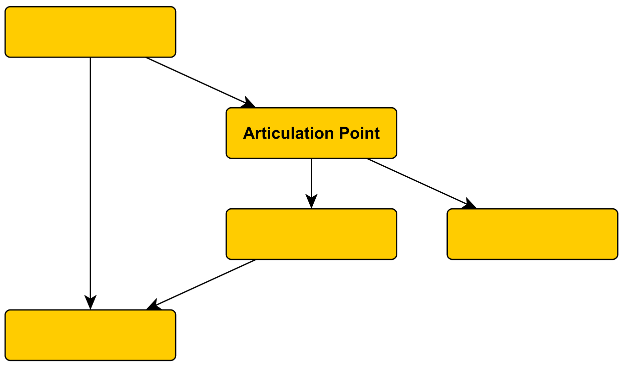


Figure 4.1: Grouping can lead to insolvable instances.

Total Edge Length Minimisation

For each edge segment of an edge $\mathtt{e}$ exists an integer variable $\mathtt{l(e,i)}$ for $\mathtt{i} = 0,\ldots,4$ that will be used for measuring the length of the corresponding edge segment. Since we know the direction of the vertical segments, we can require that $\mathtt{l(e,i)}$ equals the absolute difference of the y-coordinates of the two points defining the segment. For the horizontal segments we enforce, that the variable $\mathtt{l(e,i)}$ is at least as big as the segment length. Due to the length minimisation $\mathtt{l(e,i)}$ will be assigned with the smallest possible value, i.e. the correct segment length. This simplification improves the performance of the ILP approach.

$$\forall \mathtt{i} \in \{0,2,4\} \quad : \quad \mathtt{l(e,i)} = \mathtt{p(e,i+1).y} - \mathtt{p(e,i).y}$$

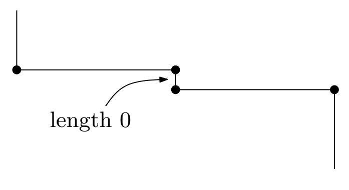
$$\begin{aligned}\forall \mathtt{i} \in \{1,3\} \quad &: \quad \mathtt{l(e,i)} \geq \mathtt{p(e,i+1).x} - \mathtt{p(e,i).x}\\ \forall \mathtt{i} \in \{1,3\} \quad &: \quad \mathtt{l(e,i)} \geq \mathtt{p(e,i).x} - \mathtt{p(e,i+1).x}\end{aligned}$$

For the total edge length minimisation we introduce the variable `total_edge_length` and the following constraint:

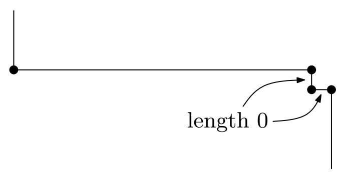
$$\mathtt{total\_edge\_length} = \sum_{\mathtt{e}\in A} \sum_{\mathtt{i=0}}^{4} \mathtt{l(e,i)}$$

Bend Minimisation

Remember that each edge is associated with four binary bend variables $\mathtt{b(e,i)}$ for $\mathtt{i} = 0,\ldots,3$ indicating whether a possible bend is indeed a bend. For counting the bends we assume that before and after each horizontal segment of non-zero length always is a vertical segment of non-zero length. Then each non-zero length horizontal segment induces two bends. However, without the assumption of vertical segments of non-zero length this counting strategy leads to an erroneous number of counted bends. In Figure 4.2a the actual bend number is 2, whereas we count 4 bends using the strategy mentioned above. However, this layout looks exactly like the layout in Figure 4.2b where our bend counting strategy works correctly. Summing up, the bends counted by our counting strategy can be more than there are actual bends in the layout. However, if too many bends have been counted, then there is an equivalent layout for that the counting strategy works correctly for it. Thus, in combination with bend minimisation this counting strategy leads to correct results. We benefit from a large performance impact in comparison to counting strategies that count correctly all the time.



(a) Erroneous counting if the vertical segment has zero length.



(b) Correct counting due to bend minimisation.

Figure 4.2: How to count bends.

Now, we turn towards the constraints we need in order to implement this counting strategy.

$$\begin{aligned}\forall \mathtt{i} \in \{1,3\} \quad &: \quad \mathtt{p(e,i).x} \neq \mathtt{p(e,i+1).x} \Longrightarrow \mathtt{b(e,i)} = 1\\ \forall \mathtt{i} \in \{1,3\} \quad &: \quad \mathtt{b(e,i)} = \mathtt{b(e,i-1)}\end{aligned}$$

The second constraint is already in the form of a linear equation, whereas we need to transform the first constraint. Therefor, we need two constraints:

$$1/\texttt{MAX} \cdot (\texttt{p(e,i).x} - \texttt{p(e,i+1).x}) - \texttt{b(e,i)} \leq 0$$
$$1/\texttt{MAX} \cdot (\texttt{p(e+1,i).x} - \texttt{p(e,i).x}) - \texttt{b(e,i)} \leq 0$$

For bend minimisation we use an additional variable named `bend_number` which equals to the sum of all binary variables $\texttt{b(e,i)}$:

$$\texttt{bend\_number} = \sum_{\texttt{e} \in E} \sum_{\texttt{i=0}}^{3} \texttt{b(e,i)}$$

Target Function

We minimise the following target function:

$$\alpha \cdot \texttt{bend\_number} + \gamma \cdot \texttt{total\_edge\_length}$$

Initial Solution

During the test runs we did along the development of the ILP approach we discovered that it often takes a long time until the ILP solver finds an initial solution. In order to skip this step we provide a partial initial solution. In our experiments we gain the best performance boost if we only provide y-coordinates for both, the boxes and the edge points. The initial solution is computed as follows: (i) we compute a layering of the boxes and (ii) assign the y-coordinates to the edge points (see Figure 4.3). The first layer consists of all boxes with in-degree zero. The $(i+1)$-th layer contains all boxes whose predecessors have all been positioned in one of the first i layers. The distance between two layers depends on the number of edges leaving the upper layer plus the number of edges entering the lower layer. Each horizontal edge segment is assigned with one unique grid row in between the two layers.

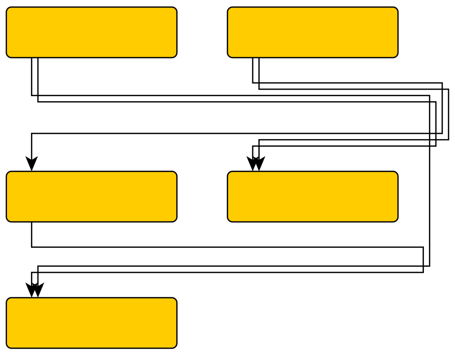


Figure 4.3: The provided initial solution.

Initial Solution for Grouped Instances

For complex grouping structures it is already a tough task to compute an initial solution. We only supply the y-coordinates of the boxes conforming to the same strategy we already described in the foregoing paragraph. However, this approach does not always give a valid initial solution. Nevertheless, a provided infeasible initial solution can be detected quickly by the ILP solver, whereas a feasible initial solution can massively speed up the computation.

4.2 Experiments

As software component we use the gurobi optimiser in version 4.6.0. We did the experiments on a 42 core machine where each core is a AMD Opteron Processor 6172 with 2100 MHz and 512 KB cache running SuSE 11.3. Of course, no end user of the algorithms we present in this work will have such a machine available. However, as already mentioned the goal of this approach is to get good overview over different layouts.

As test data we used the argument maps from the lecture “René Descartes: Meditationen über die Grundlagen der Philosophie” described in Section 2.2. For each map we ran five different configurations: one without grouping and one for each grouping strategy. Each run took 30 minutes and the intermediate layouts have been tracked. We assigned the parameters in the target function with $\alpha = 10000$ and $\gamma = 1$.

Instances without Grouping

For eight of the 34 input argument maps the ILP solver proved that it returned the optimal solution. Furthermore, for six argument maps a valid layout could not be found, because the graph was cyclic. In Figure 4.4a-4.4d we show some of the computed results. For all four instances the ILP solver did not come to an end. However, Figure 4.4a and 4.4b look like layouts that are probably close to the optimal solution. In contrast the layouts in Figure 4.4c and 4.4d show obvious shortcomings. In Figure 4.4c the layout is still close to the initial solution. The boxes with in-degree zero are still positioned at the top of the layout even though the total edge length could be minimised by moving them downward. In Figure 4.4c further shortcomings resulting from the initial solution are depicted: (i) the long vertical edges that need to span over several layers of the initial solution and (ii) the topmost edge crossing could be avoided by moving the box “Signale haben mehrere Quellen” downwards and to the right such that incident edge is routed on the right side of “Außenwelt verursacht Wahrnehmung”.
Summing up, the computed layouts look very good for instances with less than ca. 20 nodes but have obvious shortcomings for larger instances.

Instances using the Grouping Strategies

The most amazing insight concerning the grouping strategy was that they neither give benefit in terms of computation time nor in terms of aesthetics. Although the size of the matrix representing the ILP decreases dramatically due to grouping, the ILP gets harder to solve. In Table 4.1 we give the number of instances for which a layout has been computed within the 30 minutes. Using no grouping outperformed all four grouping strategies. Thus, the grouping approach failed totally. Nevertheless, we present layouts for two instances in Figure 4.5 and 4.6. Note that different strategies often lead to the same grouping and, hence, the layouts do not differ from each other. Only the strategy “lowest degree first” yielded significantly different solutions to the other grouping strategies.

Grouping	Number of solved instances
No grouping	28
Highest betweenness first	25
Highest closeness first	24
Highest degree first	25
Lowest degree first	24

Table 4.1: Number of solved instances broken down by grouping strategies.

4.3 Extending the Approach

As already mentioned this ILP approach does not support all properties and optimisation goals described in Section 3.2 and Section 3.3, respectively. In this section we discuss how the remaining properties and goals can be modelled:

Alignment

Two boxes `b` and `c` can simply be aligned at their bottom by enforcing equality of the corresponding y-coordinates.

Free Sources and Sinks

It is easy to require that above a source and below a sink no other box is positioned. We describe how to enforce free sources. The constraints for free sinks are analog. We use the relative positions between sources and all other boxes. For each pair of a source `s` and another box `b` we add the following constraint: If `b` is above `s`, then it needs to be to either the right side or the left side of `s` as well.

Distinguish between $s_{\text{edge}}^{\text{edge}}$ and $\hat{s}_{\text{edge}}^{\text{edge}}$

Up to now, we do not distinguish between $s_{\text{edge}}^{\text{edge}}$ and $\hat{s}_{\text{edge}}^{\text{edge}}$, i.e. all pairs of edges have the same minimum spacing. It is easy to modify our approach such that this distinction is respected. For those pairs of edges that neither have a common source nor sink we simply replace $s_{\text{edge}}^{\text{edge}}$ by $\hat{s}_{\text{edge}}^{\text{edge}}$ in the constraints presented in the paragraph "Overlapping Of Edges".

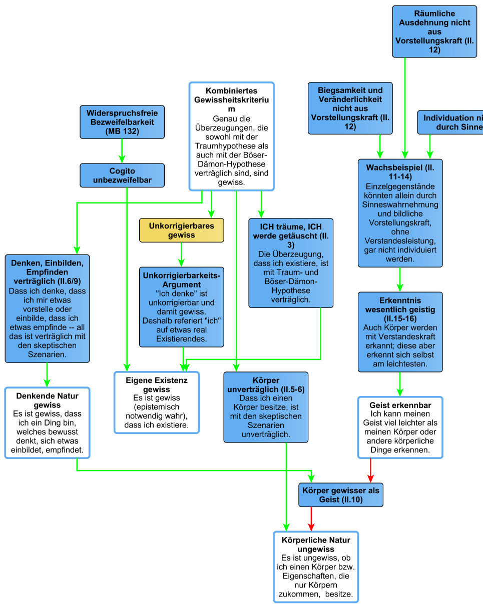
Edge Crossings

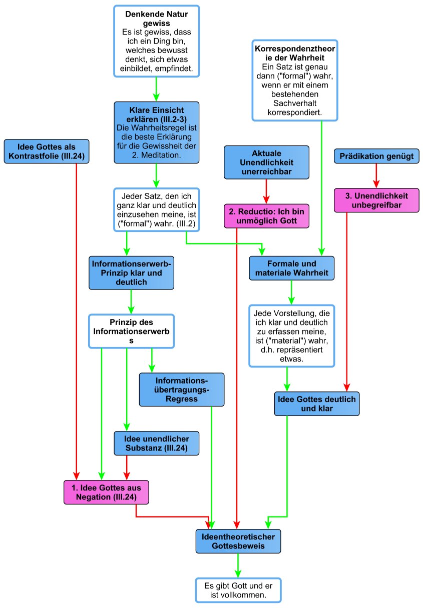
Edge crossings are complex to handle within an ILP. Basically, we need to add a binary variable for each pair of edge segments that could possibly cross. Depending on the relative positions of the incident points we can determine whether the two edge segments cross. Additionally, we introduce a variable that equals to the sum of all those binary variables and add it to the target function.

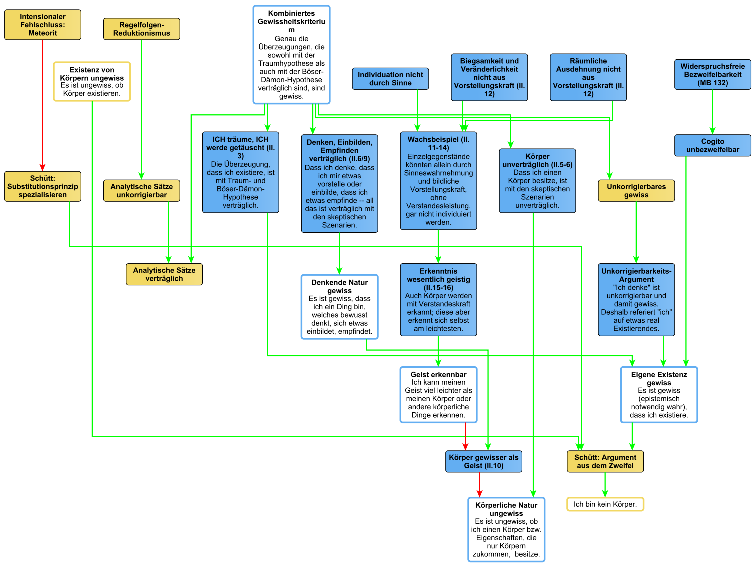
Total Source/Sink Distance Minimisation

The minimisation of the total source/sink distance can be reduced to the minimisation of edge crossings. We add two artificial nodes $\hat{s}$ and $\hat{t}$ and add edges from $\hat{s}$ to all sources and from all sinks to $\hat{t}$. The number of crossings on these edges equals the total source/sink distance. Thus, we count these crossings separately and add their sum to the target function.

As shown the remaining properties and optimisation goals can easily be added to the existing ILP approach. We believe that alignment and the distinction between $s_{\text{edge}}^{\text{edge}}$ and $\hat{s}_{\text{edge}}^{\text{edge}}$ will have only little impact on the runtime, whereas the two optimisation goals will drastically increase the runtime of the ILP solver. Therefore, it is not advisable to further pursue this approach. Instead we turn towards an efficient algorithm for the layout of argument maps that is based on the topology-shape-metrics framework. This algorithm is subject of the subsequent chapter.


Räumliche Ausdehnung nicht aus Vorstellungskraft (II. 12)
Widerspruchsfreie Bezweifelbarkeit (MB 132)
Kombiniertes Gewissheitskriterium Genau die Überzeugungen, die sowohl mit der Traumhypothese als auch mit der Böser-Dämon-Hypothese verträglich sind, sind gewiss.
Biegsamkeit und Veränderlichkeit nicht aus Vorstellungskraft (II. 12)
Individuation nicht durch Sinne
Cogito unbezweifelbar
Wachsbeispiel (II. 11-14) Einzelgegenstände könnten allein durch Sinneswahrnehmung und bildliche Vorstellungskraft, ohne Verstandesleistung, gar nicht individuiert werden.
Unkorrigierbares gewiss
ICH träume, ICH werde getäuscht (II. 3) Die Überzeugung, dass ich existiere, ist mit Traum- und Böser-Dämon-Hypothese verträglich.
Denken, Einbilden, Empfinden verträglich (II.6/9) Dass ich denke, dass ich mir etwas vorstelle oder einbilde, dass ich etwas empfinde – all das ist verträglich mit den skeptischen Szenarien.
Unkorrigierbarkeits-Argument "Ich denke" ist unkorrigierbar und damit gewiss. Deshalb referiert "ich" auf etwas real Existierendes.
Erkenntnis wesentlich geistig (II.15-16) Auch Körper werden mit Verstandeskraft erkannt; diese aber erkennt sich selbst am leichtesten.
Denkende Natur gewiss Es ist gewiss, dass ich ein Ding bin, welches bewusst denkt, sich etwas einbildet, empfindet.
Eigene Existenz gewiss Es ist gewiss (epistemisch notwendig wahr), dass ich existiere.
Körper unverträglich (II.5-6) Dass ich einen Körper besitze, ist mit den skeptischen Szenarien unverträglich.
Geist erkennbar Ich kann meinen Geist viel leichter als meinen Körper oder andere körperliche Dinge erkennen.
Körper gewisser als Geist (II.10)
Körperliche Natur ungewiss Es ist ungewiss, ob ich einen Körper bzw. Eigenschaften, die nur Körpern zukommen, besitze.
(a) Instance D-2.3.


Denkende Natur gewiss Es ist gewiss, dass ich ein Ding bin, welches bewusst denkt, sich etwas einbildet, empfindet.
Korrespondenztheorie der Wahrheit Ein Satz ist genau dann ("formal") wahr, wenn er mit einem bestehenden Sachverhalt korrespondiert.
Klare Einsicht erklären (III.2-3) Die Wahrheitsregel ist die beste Erklärung für die Gewissheit der 2. Meditation.
Idee Gottes als Kontrastfolie (III.24)
Aktuale Unendlichkeit unerreichbar
Prädikation genügt
Jeder Satz, den ich ganz klar und deutlich einzusehen meine, ist ("formal") wahr. (III.2)
2. Reductio: Ich bin unmöglich Gott
3. Unendlichkeit unbegreifbar
Informationserwerb-Prinzip klar und deutlich
Formale und materiale Wahrheit
Prinzip des Informationserwerbs
Jede Vorstellung, die ich klar und deutlich zu erfassen meine, ist ("material") wahr, d.h. repräsentiert etwas.
Informations-übertragungs-Regress
Idee Gottes deutlich und klar
Idee unendlicher Substanz (III.24)
1. Idee Gottes aus Negation (III.24)
Ideentheoretischer Gottesbeweis
Es gibt Gott und er ist vollkommen.
(b) Instance D-3.5.


Intensionaler Fehlschluss: Meteorit
Regelfolgen-Reduktionismus
Kombiniertes Gewissheitskriterium Genau die Überzeugungen, die sowohl mit der Traumhypothese als auch mit der Böser-Dämon-Hypothese verträglich sind, sind gewiss.
Existenz von Körpern ungewiss Es ist ungewiss, ob Körper existieren.
Individuation nicht durch Sinne
Biegsamkeit und Veränderlichkeit nicht aus Vorstellungskraft (II. 12)
Räumliche Ausdehnung nicht aus Vorstellungskraft (II. 12)
Widerspruchsfreie Bezweifelbarkeit (MB 132)
ICH träume, ICH werde getäuscht (II. 3) Die Überzeugung, dass ich existiere, ist mit Traum- und Böser-Dämon-Hypothese verträglich.
Denken, Einbilden, Empfinden verträglich (II.6/9) Dass ich denke, dass ich mir etwas vorstelle oder einbilde, dass ich etwas empfinde – all das ist verträglich mit den skeptischen Szenarien.
Wachsbeispiel (II. 11-14) Einzelgegenstände könnten allein durch Sinneswahrnehmung und bildliche Vorstellungskraft, ohne Verstandesleistung, gar nicht individuiert werden.
Körper unverträglich (II.5-6) Dass ich einen Körper besitze, ist mit den skeptischen Szenarien unverträglich.
Cogito unbezweifelbar
Schütt: Substitutionsprinzip spezialisieren
Analytische Sätze unkorrigierbar
Unkorrigierbares gewiss
Analytische Sätze verträglich
Denkende Natur gewiss Es ist gewiss, dass ich ein Ding bin, welches bewusst denkt, sich etwas einbildet, empfindet.
Erkenntnis wesentlich geistig (II.15-16) Auch Körper werden mit Verstandeskraft erkannt; diese aber erkennt sich selbst am leichtesten.
Unkorrigierbarkeits-Argument "Ich denke" ist unkorrigierbar und damit gewiss. Deshalb referiert "ich" auf etwas real Existierendes.
Geist erkennbar Ich kann meinen Geist viel leichter als meinen Körper oder andere körperliche Dinge erkennen.
Eigene Existenz gewiss Es ist gewiss (epistemisch notwendig wahr), dass ich existiere.
Körper gewisser als Geist (II.10)
Schütt: Argument aus dem Zweifel
Ich bin kein Körper.
Körperliche Natur ungewiss Es ist ungewiss, ob ich einen Körper bzw. Eigenschaften, die nur Körpern zukommen, besitze.
(c) Instance D-2.7.

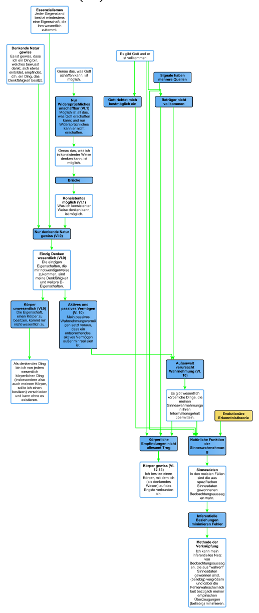
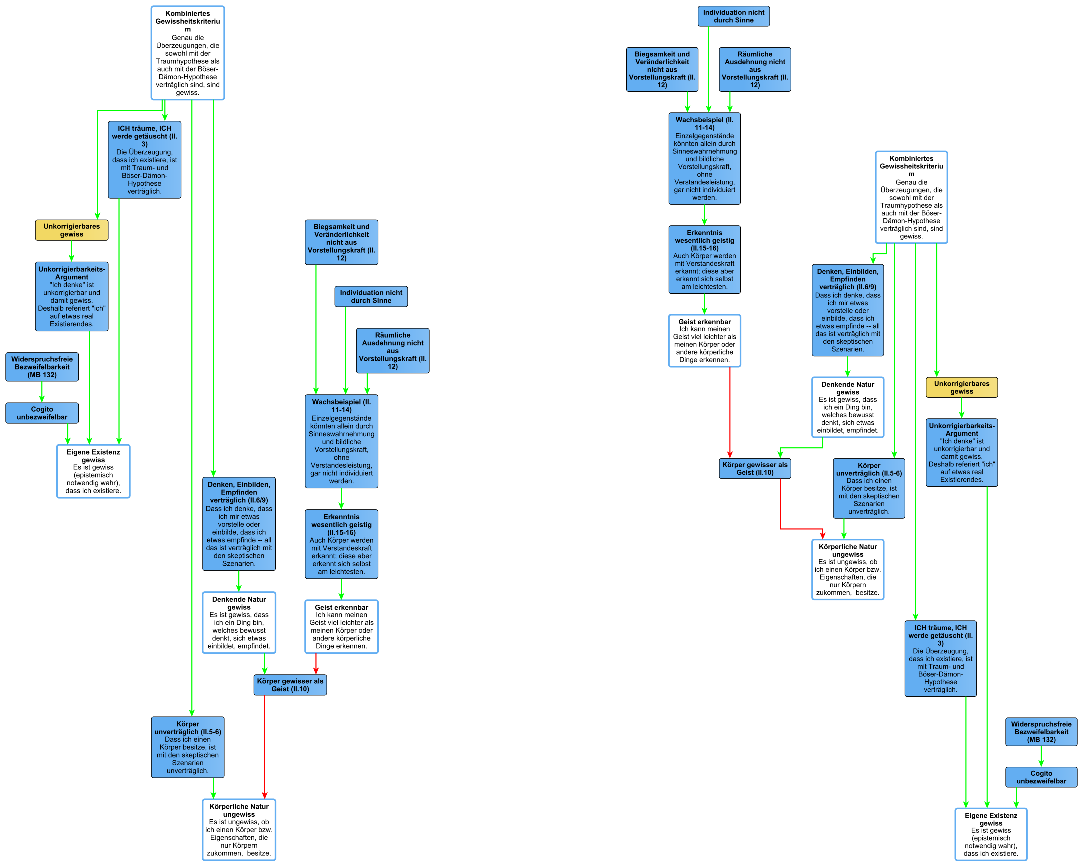

(d) Instance D-6.4.

Figure 4.4: Solutions for instances without grouping.


Kombiniertes Gewissheitskriterium
Genau die Überzeugungen, die sowohl mit der Traumhypothese als auch mit der Böser-Dämon-Hypothese verträglich sind, sind gewiss.
ICH träume, ICH werde getäuscht (II. 3)
Die Überzeugung, dass ich existiere, ist mit Traum- und Böser-Dämon-Hypothese verträglich.
Unkorrigierbares gewiss
Unkorrigierbarkeits-Argument
"Ich denke" ist unkorrigierbar und damit gewiss. Deshalb referiert "ich" auf etwas real Existierendes.
Widerspruchsfreie Bezweifelbarkeit (MB 132)
Cogito unbezweifelbar
Eigene Existenz gewiss
Es ist gewiss (epistemisch notwendig wahr), dass ich existiere.
Biegsamkeit und Veränderlichkeit nicht aus Vorstellungskraft (II. 12)
Individuation nicht durch Sinne
Räumliche Ausdehnung nicht aus Vorstellungskraft (II. 12)
Wachsbeispiel (II. 11-14)
Einzelgegenstände könnten allein durch Sinneswahrnehmung und bildliche Vorstellungskraft, ohne Verstandesleistung, gar nicht individuiert werden.
Denken, Einbilden, Empfinden verträglich (II.6/9)
Dass ich denke, dass ich mir etwas vorstelle oder einbilde, dass ich etwas empfinde – all das ist verträglich mit den skeptischen Szenarien.
Erkenntnis wesentlich geistig (II.15-16)
Auch Körper werden mit Verstandeskraft erkannt; diese aber erkennt sich selbst am leichtesten.
Denkende Natur gewiss
Es ist gewiss, dass ich ein Ding bin, welches bewusst denkt, sich etwas einbildet, empfindet.
Geist erkennbar
Ich kann meinen Geist viel leichter als meinen Körper oder andere körperliche Dinge erkennen.
Körper gewisser als Geist (II.10)
Körper unverträglich (II.5-6)
Dass ich einen Körper besitze, ist mit den skeptischen Szenarien unverträglich.
Körperliche Natur ungewiss
Es ist ungewiss, ob ich einen Körper bzw. Eigenschaften, die nur Körpern zukommen, besitze.

(a) Using highest betweenness first, highest closeness first or highest degree first.

(b) Using lowest degree first.

Figure 4.5: Solutions for instance D-2.5 with grouping.

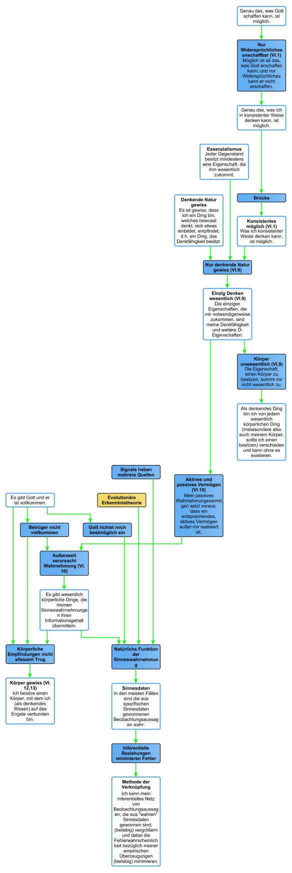

Genau das, was Gott schaffen kann, ist möglich.
Nur Widersprüchliches unschaffbar (VI.1) Möglich ist all das, was Gott erschaffen kann; und nur Widersprüchliches kann er nicht erschaffen.
Genau das, was ich in konsistenter Weise denken kann, ist möglich.
Essenzialismus Jeder Gegenstand besitzt mindestens eine Eigenschaft, die ihm wesentlich zukommt.
Brücke
Denkende Natur gewiss Es ist gewiss, dass ich ein Ding bin, welches bewusst denkt, sich etwas einbildet, empfindet, d.h. ein Ding, das Denkfähigkeit besitzt.
Konsistentes möglich (VI.1) Was ich konsistenter Weise denken kann, ist möglich.
Nur denkende Natur gewiss (VI.9)
Einzig Denken wesentlich (VI.9) Die einzigen Eigenschaften, die mir notwendigerweise zukommen, sind meine Denkfähigkeit und weitere D-Eigenschaften.
Körper unwesentlich (VI.9) Die Eigenschaft, einen Körper zu besitzen, kommt mir nicht wesentlich zu.
Als denkendes Ding bin ich von jedem wesentlich körperlichen Ding (insbesondere also auch meinem Körper, sollte ich einen besitzen) verschieden und kann ohne es existieren.
Signale haben mehrere Quellen
Aktives und passives Vermögen (VI.10) Mein passives Wahrnehmungsvermögen setzt voraus, dass ein entsprechendes, aktives Vermögen außer mir realisiert ist.
Es gibt Gott und er ist vollkommen.
Evolutionäre Erkenntnistheorie
Betrüger nicht vollkommen
Gott richtet mich bestmöglich ein
Außenwelt verursacht Wahrnehmung (VI. 10)
Es gibt wesentlich körperliche Dinge, die meinen Sinneswahrnehmungen ihren Informationsgehalt übermitteln.
Körperliche Empfindungen nicht allesamt Trug
Natürliche Funktion der Sinneswahrnehmung
Körper gewiss (VI. 12,13) Ich besitze einen Körper, mit dem ich (als denkendes Wesen) auf das Engste verbunden bin.
Sinnesdaten In den meisten Fällen sind die aus spezifischen Sinnesdaten gewonnenen Beobachtungsaussagen wahr.
Inferentielle Beziehungen minimieren Fehler
Methode der Verknüpfung Ich kann mein inferentielles Netz von Beobachtungsaussagen, die aus "wahren" Sinnesdaten gewonnen sind, (beliebig) vergrößern und dabei die Fehlerwahrscheinlichkeit bezüglich meiner empirischen Überzeugungen (beliebig) minimieren.

Figure 4.6: Solution for instance D-6.4 using any grouping strategy.

5. Topology-Shape-Metrics

A frequently used framework for drawing graphs is the so called *topology-shape-metrics framework*. It was introduced in 1987 by Tamassia in his seminal paper “On embedding a graph in the grid with the minimum number of bends” [Tam87]. A decade later, the term topology-shape-metrics was introduced by Tollis et al. [TDET98].
The framework consist of the three basic steps *topology*, *shape* and *metrics*. The key idea is that reducing the number of crossings is the most important aesthetic criterion and, therefore, is minimised in the first step without considering any other criteria. Thereby, an embedding of the input graph is computed. For non-planar graphs crossings are replaced by dummy nodes with degree 4. The planarisation together with its embedding is called *topology*. Generally, the topology is fixed throughout the remaining steps of the algorithm.
In the second step, the *shape* of the final layout is optimised with respect to the fixed topology. The shape is the assignment of bends to edges and the angle of these bends. Usually, the goal of this step is to minimise the number of bends.
Finally, in the step *metrics*, the edge lengths and node dimensions are computed.
Since each step can be handled individually, topology-shape-metrics can be seen as an algorithmic framework. It has been intensively studied in the last two decades yielding algorithms for a vast variety of applications including the visualisation of data flow diagrams [BNT86], database schemes [DDPP02], industrial plant schemes [DPP02] and UML diagrams [EKS03]. In this work we apply the topology-shape-metrics approach to argument maps.

Remember that the problem Argument Map is a complex problem. Using the topology-shape-metrics framework we split up Argument Map into smaller subproblems and either solve them optimally or apply heuristics if they are $\mathcal{NP}$-hard. In Table 5.1 we show for each phase of the framework which of the required properties mentioned in Section 3.2 is enforced and which of the optimisation goals described in Section 3.3 is treated. Note that grouping is not listed in Table 5.1, because we do not treat this property within our topology-shape-metrics approach. However, this is an interesting topic to pursue for future work (see Chapter 7).

Our algorithm has two important differences in comparison to the common usage of the topology-shape-metrics-framework:

- In the first phase we not only minimise the number of crossing but also optimise the total source/sink distance.

	Enforced Properties	**Treated Goals**
Topology	Upward Drawing	Crossing Minimisation Total Source/Sink Distance Minimisation
Shape	Orthogonal Edges Port Distribution Free Sources And Sinks	Bend Minimisation
Metrics	Box Dimensions Spacing Constraints Alignment	Total Edge Length Minimisation

Table 5.1: An overview of the topology-shape-metrics approach.

- We do not fix the topology computed in the first step. We found out that during the shape and metrics phase minor changes to the topology can improve the aesthetics of the final layout. Thereby, we assign higher significance to bend minimisation and total edge length minimisation than in usual applications of this framework.

The remaining part of this chapter is structured along the topology-shape-metrics framework. In the next section we state some preliminaries. The subsequent sections are each dedicated to one of the three steps, topology, shape and metrics (Sections 5.2-5.4). Finally, we give a short summary in Section 5.5.

5.1 Preliminaries

In this section, we present some preliminary work on which the algorithms in the following three sections are based. In particular, we make the input graph acyclic and transform it such that there is a unique source and a unique sink.

In the formal problem statement Argument Map we required that the input graph $G = (V, A)$ is a DAG (see Section 3.4). However, as discussed in Chapter 2 there are some real input instances that contain cycles. In this section we treat these instances by reversing the direction of edges in G such that it becomes cycle-free. Such a set of edges is called a *feedback arc set.* We denote the resulting graph by $\widehat{G} = (\widehat{V}, \widehat{A})$. Then we apply the algorithms presented in the following chapters to $\widehat{G}$. Afterwards, we convert the layout of $\widehat{G}$ to G. For edges that are reverted in $\widehat{G}$ we simply toggle the position of the arrowhead. Note that for these edges the upward drawing constraint is not satisfied anymore. Because of this, we aim to compute a small feedback arc set. However, the problem of finding a minimum feedback arc set is $\mathcal{NP}$-complete [Kar72]. In our implementation we use a well-known heuristic suggested by Eades et al. [ELS93], which is implemented in the class GreedyCycleRemoval of the Open Graph Drawing Framework (OGDF)[1]. This algorithm runs in linear time and returns a set of at most $|A|/2 - |V|/6$ edges such that reversing their direction results in an acyclic graph $\widehat{G}$.

Besides the acyclicity of $\widehat{G}$, we further require that it is an s-t-graph. Before describing how $\widehat{G}$ is transformed to an s-t-graph, we define the term s-t-graph:

Definition 5.1 (s-t-Graph, s-T-Graph)**.** *An* s-t-graph *is a graph having a single source and a single sink. An* s-T-graph *has a single source as well but can have multiple sinks.*

We transform $\widehat{G}$ to an s-t-graph by adding a super source $\hat{s}$ and a super sink $\hat{t}$ and connecting them with all previous sources and sinks, respectively. Note that we do this independently of whether $\widehat{G}$ was an s-t-graph before, because the additional edges adjacent

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

to $\hat{s}$ and $\hat{t}$ will play a special role in Section 5.2 in order to weight crossing minimisation and total source/sink distance minimisation. When converting the layout of $\widehat{G}$ to G we simply omit $\hat{s}$ and $\hat{t}$ as well as all incident edges.

With these preparations we can apply the algorithms presented in Sections 5.2-5.4. In the following chapter we describe an algorithm that computes a planarisation of $\widehat{G}$ such that all edges are directed upwards while heuristically minimising the number of crossings and the total source/sink distance.

5.2 Topology

In this section we deal with the first step of the trinomial topology-shape-metrics framework. As we already applied the techniques described in Section 5.1 we assume to have an s-t-graph $\widehat{G}$ as input data. We denote the unique source by $\hat{s}$ and the unique sink by $\hat{t}$. Note that $\widehat{G}$ may be non-planar. Therefore, we cannot directly compute an upward planar embedding of it. Instead, we pursue two goals, namely crossing minimisation and total source/sink distance minimisation. We offer the possibility to weight these two criteria by using two parameters. As upward crossing minimisation by itself is already $\mathcal{NP}$-complete [GT94, GT02], we build on a heuristic approach called "layer-free upward crossing minimization" by Chimani et al. [CGMW10]. The outcome is a so called *upward planar representation*:

Definition 5.2 (Upward Planar Representation)**.** *Given a DAG $G = (V, A)$, an* upward planar representation *$\mathcal{U}$ of G is an upward planar graph $\mathcal{U} = (V_\mathcal{U}, A_\mathcal{U})$ in which edge crossings are replaced by crossing dummies together with an upward planar embedding Γ and a designated external face f_0. The set $V_\mathcal{U}$ is a super set of V that additionally contains the crossing dummies. The arcs $a \in A$ are either elements of $A_\mathcal{U}$ as well or correspond to a path p in $\mathcal{U}$. The inner nodes of such a path p are crossing dummies, whereas the source and target node of p correspond to the source and target of a, respectively.*

The motivation for layer-free upward crossing minimisation originates from the shortcomings of the traditional Sugiyama framework [STT81]. This approach consists of three steps of which each step can be solved individually. Although this framework has been invented in 1981, it is still state-of-the-art in drawing DAGs. The three steps are:

(i) **Layer assignment**: The nodes are assigned to layers such that the target of each edge is in a layer below its source's layer. If an edge spans multiple layers it is split into several edges such that each segment connects neighbouring layers.

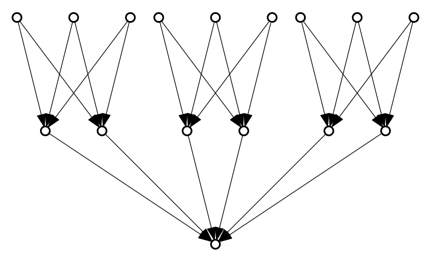
(ii) **Crossing reduction**: The order of the nodes within a layer is changed such that the number of edge crossings is minimised.

(iii) **Coordinate assignment**: Final coordinates are assigned to the nodes as well as to bend-points of the edges.

The shortcoming of this framework is that computing an unfortunate layering of the nodes in step (i) can enforce crossings in step (ii) which would not be necessary if another layering was chosen. We cite an illustrating example by Chimani et al. in Figure 5.1.

In order to circumvent this shortcoming of the Sugiyama framework Chimani et al. developed an upward crossing minimisation method that works without a prescribed layer assignment of the nodes. We use their technique in order to minimise both, the crossings and the total source/sink distance.

The remaining part of this section is structured as follows: In Section 5.2.1 we discuss the complexity of upward crossing minimisation of s-t-graphs. Afterwards, we describe the algorithm by Chimani et al. in Section 5.2.2 and discuss how their approach can be used to compute a layout of an argument map in Section 5.2.3.


(a) Using this layering the nine crossings cannot be avoided.

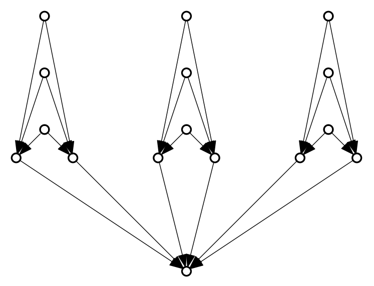

(b) Using another layering the crossing number can be reduced to zero.

Figure 5.1: An unfortunate layering leads to crossings that are unnecessary if another layering is chosen.

5.2.1 Complexity Considerations

While planarity of arbitrary graphs can be tested efficiently, testing upward planarity is $\mathcal{NP}$-complete for general graphs [GT94, GT02]. Therefore, upward crossing minimisation is $\mathcal{NP}$-complete as well. However, for s-T-graphs the test for upward planarity can be done in polynomial time [BDMT98], and thus upward crossing minimisation might be efficiently solvable for s-T-graphs. Remember that the definition of Argument Map includes the four parameters α, β, γ and δ which weight the four optimisation goals number of crossings, number of bends, total edge length and total source/sink distance, respectively. For parameter choices that satisfy $\alpha = \delta$ the approach we will present in this chapter reduces to upward crossing minimisation of an s-t-graph. Therefore, we investigate this problem in the remaining part of this section. We first give a formal definition of this problem and denote it by s-t-Upward Crossing Minimisation:

Instance: An s-t-graph $G = (V, A)$ and an integer $k \geq 0$.
Question: Does there exist an upward drawing of G in the plane with at most k edge crossings?

Garey and Johnson proved that crossing minimisation in general undirected graphs is $\mathcal{NP}$-complete [GJ83]. By slightly modifying their proof, we show that s-t-Upward Crossing Minimisation is $\mathcal{NP}$-complete as well.

In their proof Garey and Johnson reduce Bipartite Crossing Number to the crossing minimisation problem. The problem Bipartite Crossing Number is known under a variety of names. Eades et al. refer to it as Left Optimal Drawing [EMW86] and it is often denoted by 2-Layer Crossing Minimisation:

Instance: A connected bipartite graph $G = (V_1 \cup V_2, E)$ and an integer $k \geq 0$.
Question: Can G be embedded in a unit square such that all vertices of V_1 are on the northern boundary, all vertices in V_2 are on the southern boundary, all edges are within the square and there are at most k crossings?

They transform a given Bipartite Crossing Number instance to an instance of the crossing minimisation by constructing the graph $G' = (V', E \cup E_1 \cup E_2 \cup E_3)$, where

$$\begin{aligned}
V' &= V_1 \cup V_2 \cup \{u_0, w_0\}, \\
E_1 &= \{3k + 1 \text{ copies of } \{u_0, u\} : u \in V_1\}, \\
E_2 &= \{3k + 1 \text{ copies of } \{w_0, w\} : w \in V_2\}, \\
E_3 &= \{3k + 1 \text{ copies of } \{u_0, w_0\}\}.
\end{aligned}$$

Obviously, this reduction can be performed in polynomial time. Then, they show that the Bipartite Crossing Number instance has a solution if and only if G' has a drawing with at most k crossings. One direction of this claim is easy to prove. Having a drawing of the Bipartite Crossing Number instance, we can easily extend it to a drawing of G' by adding the missing edges (see Figure 5.2a). The other direction is harder to prove. Garey and Johnson give a sequence of normalisation steps that transform a valid drawing of G' to a drawing that is topologically equivalent to the drawing depicted in Figure 5.2a. For details on these normalisation steps we refer the reader to the work by Garey and Johnson [GJ83]. Having such a drawing at hand, a solution of the Bipartite Crossing Number instance can be constructed by simply removing the additional edges in E_1, E_2 and E_3.

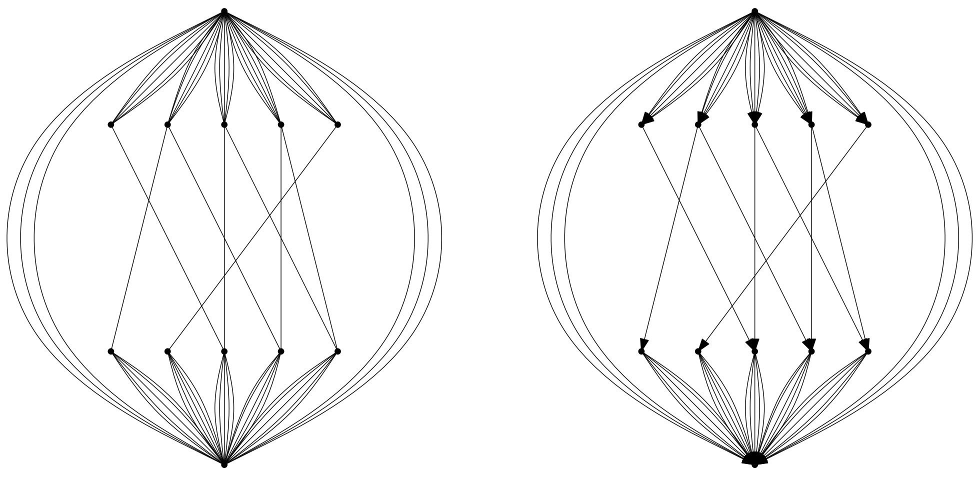

(a) Graph G' from the proof by Garey and Johnson.
(b) The directed version of G' used in our modified proof.

Figure 5.2: Illustration of the $\mathcal{NP}$-completeness proof of s-t-Upward Crossing Minimisation.

We now modify the proof by Garey and Johnson such that we can prove, that s-t-Upward Crossing Minimisation is $\mathcal{NP}$-complete. First, we assign directions to the edges in G' such that it is a DAG. To this end, we direct all edges such that they point upwards in Figure 5.2a. Since G was restricted to be connected by the Bipartite Crossing Number definition, the resulting graph (see Figure 5.2b) is an *s*-*t*-graph. Thus, we transformed the Bipartite Crossing Number instance to an s-t-Upward Crossing Minimisation instance. Again the following equivalence holds: The Bipartite Crossing Number instance has a solution if and only if G' has an upward drawing with at most k crossings. We can directly carry over the proof of this equivalence from the work by Garey and Johnson [GJ83]. Thus, s-t-Upward Crossing Minimisation is $\mathcal{NP}$-complete as well.

5.2.2 Layer-Free Upward Crossing Minimisation

Recently, Chimani et al. presented a heuristical way to compute an upward drawing of a DAG such that the number of crossing is small. They came back to the idea of the planarisation approach, which is a frequently and successfully used heuristic for crossings minimisation in undirected graphs [BTT84, GM04]. This approach can be divided into two steps: (i) a large planar subgraph is computed and embedded and (ii) the remaining edges are reinserted. During the insertion of edges, the created crossings are replaced by nodes of degree four. These nodes are denoted by *dummy crossings*. Thus, the result is a planar representation of the original graph. In this section we describe the details of these two steps in the context of layer-free upward crossing minimisation. For technical reasons Chimani et al. assume that the input graph is an s-T-graph, i.e. a graph with a unique source and multiple sinks. If there are several sources, a super source $\hat{s}$ is added and connected to all sources. During the crossing minimisation the crossings on the edges adjacent to $\hat{s}$ are not counted, i.e. only the number of crossings in the original graph is minimised. The following three paragraphs cite the work by Chimani et al. [CGMW10] – the definitions and lemmas are directly carried over.

A Feasible Upward Planar Subgraph

The goal of the whole method is to compute an upward planar representation $\mathcal{U}$ of a DAG $G = (V, A)$ with few crossing dummies. In this paragraph we focus on the computation of an upward planar subgraph of G and an embedding of it. We denote the upward planar subgraph by $U = (V, A')$ and its embedding by Γ. Note that we cannot choose any upward planar of subgraph of G. There might be subgraphs that do not admit the remaining edges to be reinserted in an upward drawing style (see Figure 5.4a). Subgraphs that allow such a reinsertion of the missing edges are called feasible upward planar subgraphs (see Definition 5.5). Before we can discuss feasibility of U, we need to introduce the terms *upward insertion path* and *upward insertion sequence*:

Definition 5.3 (Upward Insertion Path)**.** *An* upward insertion path p_1 *with respect to some edge* $e_1 = (x_1, y_1) \in A \backslash A'$ *is an ordered list of edges* $a_1, \ldots, a_\kappa \in U$ *such that the graph* U_1 *obtained from* realising p_1 *is upward planar. The realisation works as follows: we split the edges* $a_1, \ldots, a_\kappa$ *obtaining the dummy nodes* $d_1, \ldots, d_\kappa$*, and add the edges* $(x_1, d_1), (d_1, d_2), \ldots, (d_\kappa, y_1)$ *representing* e_1*.*
Let Γ_1 *be an upward planar embedding of* U_1*. We say* Γ_1 induces *an upward planar embedding* Γ *of* U*, which is obtained by reversing the realisation procedure while maintaining the embedding.*

Definition 5.4 (Upward Insertion Sequence)**.** *An* upward insertion sequence *is a sequence of* k *upward insertion paths of the edges that are in* $A \backslash A'$ *. Thereby, the first edge in the sequence is inserted into* U *– introducing dummy nodes – which results in an upward-planar graph* U_1*. The second edge is then inserted into* U_1*, etc. After realising all insertion paths, we hence obtain a final upward planar graph* U_k*, which is a planarised representation of* G*.*

Now, we are ready to define feasibility of an upward planar subgraph U as the possibility to reinsert all remaining edges such that we gain an upward planar representation $\mathcal{U}$ of G:

Definition 5.5 (Feasible Upward Planar Subgraph and Embedding)**.** *An upward planar subgraph* U *of* G *is* feasible *iff there exists an upward insertion sequence. An upward planar embedding* Γ *of a feasible upward planar subgraph* U *is feasible iff there exists an upward insertion sequence such that* Γ_k *induces* Γ*.*

We omit the algorithm for computing a feasible upward planar subgraph by Chimani et al. but present our own approach in Section 5.2.3. However, it is important to note that the feasible upward planar subgraph Chimani et al. use is an s-T-graph and since the insertion of an edge cannot introduce a second source, all U_i are s-T-graphs as well.
As shown in Figure 5.4a not all upward planar subgraphs of G are feasible. Thus, there is a need for a characterisation of feasibility such that an upward planar subgraph can efficiently be tested upon feasibility. Therefore, Chimani et al. introduce the *merge graph* which is acyclic if and only if there exists an upward insertion sequence.
The definition of the merge graph uses the term *sink-switch.* A *sink-switch* of a face f of an upward planar embedding Γ is defined as a node v that is incident to f but has no

outgoing edge incident to f (see Figure 5.3). Since U is an s-T-graph, for each face there is a unique sink-switch that has to be drawn above all other sink-switches. We denote this sink switch by *top sink-switch.* Since the preferred drawing style for argument maps have downward directed edges, the top sink-switch is the bottommost sink-switch in Figure 5.3.

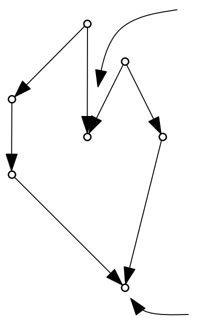


Figure 5.3: A face and its sink-switches.

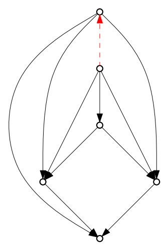
Now, we are able to define the *merge graph* as introduced by Chimani et al.:

Definition 5.6 (Merge Graph)**.** *Let U be an upward planar subgraph of G and Γ an upward planar embedding of U. The* merge graph $\mathcal{M}(\Gamma)$ *of U with respect to Γ is constructed as follows:*

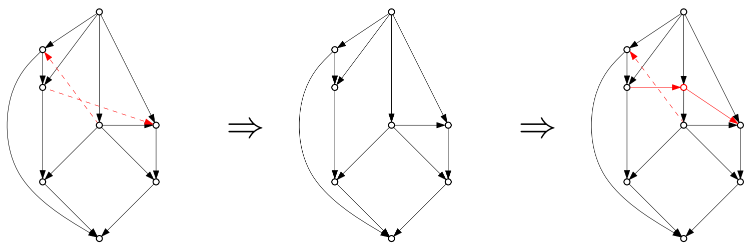
1. *Start with $\mathcal{M}(\Gamma)$ as a copy of G.*
2. *For each internal face f of Γ add an arc from each non-top sink-switch of f to the top sink-switch of f. We call these edges* sink arcs*.*

Lemma 5.1 characterises feasibility of an upward planar graph and its embedding in terms of acyclicity of the corresponding merge graph. This lemma is the crucial point of layer-free upward crossing minimisation and is as well important in the context of edge reinsertion. See Chimani et al. for the proof [CGMW10].

Lemma 5.1 (Feasibility Lemma)**.** *The merge graph $\mathcal{M}(\Gamma)$ is acyclic if and only if there exists an upward insertion sequence such that the resulting graph is upward planar.*



(a) Not all upward planar subgraphs are feasible. The red dashed edge cannot be inserted.



(b) The given graph (left) and a feasible upward planar subgraph (middle) of it. All edges have cost 1. After one edge has been inserted cost minimal (right) the other one cannot be inserted anymore.

Figure 5.4: Problems in the context of edge insertion.

Reinserting Edges

After a feasible upward planar subgraph has been computed, the remaining edges are inserted one by one. For the insertion of a single edge Chimani et al. use a routing network that offers the possibility to assign costs to all edges in the feasible upward planar subgraph and to the already reinserted edges. Crossing an edge costs the amount that is assigned to the crossed edge. The edge that is to be reinserted is routed such that the sum of the costs of the crossed edges is minimised. Chimani et al. assign costs 1 to all edges – except the edges adjacent to $\hat{s}$, which are assigned with cost 0. Thus, only crossings of original edges are counted.
However, as can be seen in Figure 5.4b the cost minimal insertion of an edge e_i can make further edge insertions impossible. According to Chimani et al., one needs to check the merge graph $\mathcal{M}(\Gamma_i)$ corresponding to the intermediate graph U_i and its upward planar embedding Γ_i upon acyclicity. However, we disagree on this point: The merge graph does not respect the upward insertion paths of the already inserted edges. We illustrate this using Figure 5.4b. After the first edge has been reinserted yielding U_1 and Γ_1 (rightmost picture) the merge graph $\mathcal{M}(\Gamma_1)$ equals to the original graph G (leftmost picture). Although this graph is acyclic, there is no possibility to reinsert the second edge into U_1 such that it is directed upwards.
However, we can fix this mistake by modifying the definition of a merge graph:

Definition 5.7 (Merge Graph (Corrected Version))**.** *Let U_i be a planarised upward subgraph of G and Γ_i an upward planar embedding of U_i. The* merge graph *$\mathcal{M}(\Gamma_i)$ of U_i with respect to Γ_i is constructed as follows:*

1. *Start with $\mathcal{M}(\Gamma_i)$ as a copy of U_i.*
2. *Add the missing edges $e_{i+1}, \dots, e_k \in G$ to $\mathcal{M}(\Gamma_i)$.*
3. *For each internal face f of Γ_i add a sink arc from each non-top sink-switch of f to the top sink-switch of f.*

As long as U_i is an upward planar subgraph of G the two definitions Definition 5.6 and 5.7 are equal. However, Definition 5.7 can also be applied in the context of edge reinsertion. All the proofs presented in the work by Chimani et al. [CGMW10] are still correct if using Definition 5.7 instead of Definition 5.6.

We return to the reinsertion of an edge e_i. If after the insertion (our modified version of) the merge graph $\mathcal{M}(\Gamma_i)$ contains a cycle, the edge insertion is undone and the edge is routed by using a simple heuristic. This heuristic, has no guarantee on the optimality, i.e. the number of crossed edges, but yields an intermediate graph U_i and an upward planar embedding Γ_i such that the merge graph $\mathcal{M}(\Gamma_i)$ is acyclic.
Chimani et al. conducted experiments that revealed that the heuristic edge insertion is used in far below 1% of edge insertions. Thus, for a fixed feasible upward planar subgraph and a fixed ordering of the remaining edges the resulting upward planar representation is very close to the cost-minimal solution.

Runtime and Space Analysis

Chimani et al. present an algorithm that computes a feasible upward planar subgraph in $\mathcal{O}(|A|^2)$ time. The algorithm for cost minimal insertion of an edge a_i runs in $\mathcal{O}(|V| + r)$ time, where r is the number of edges to insert after a_i, whereas the heuristic runs in $\mathcal{O}(|V|^2 + r|V|)$ time. Note that these time complexities are not affected by our correction of the definition of the merge graph.
Chimani et al. further prove that the routing network used for the reinsertion of the first

edge has $\mathcal{O}(|V|)$ nodes and edges. For further edge insertions the routing network has $\mathcal{O}(|V| + \#\text{crossing dummies})$ nodes and edges.

Chimani et al. suggest to randomise the algorithm for the computation of the feasible upward planar subgraph and the order in which the remaining edges are reinserted. Among several runs the crossing minimal (not cost minimal!) upward planar representation is chosen as the overall result.

5.2.3 Modifications for Argument Maps

The layer-free upward crossing minimisation approach by Chimani et al. as described in the previous section only serves for crossing minimisation. However, for the computation of argument map layouts we want to minimise both, the number crossings as well as the total source/sink distance. Furthermore, we want to weight these two optimisation goals against each other by using the two parameters α and δ in Argument Map. To this end, we model the total source/sink distance in terms of the number of crossings. When constructing the routing network for edge reinsertion we distinguish between two types of edges and assign the costs α and δ, respectively, in order to weight the two optimisation goals against each other.

In Section 5.1 we already removed cycles in G and added a super source $\hat{s}$ as well as a super sink $\hat{t}$. We denote the resulting graph by $\widehat{G}$. Now, we first compute a feasible upward planar subgraph U of $\widehat{G}$ that contains all edges adjacent to $\hat{s}$ and $\hat{t}$ and an upward planar embedding Γ of U such that $\hat{s}$ and $\hat{t}$ are on the outer face. We compute such a feasible upward planar subgraph as follows:
We start with U containing only the edges adjacent to $\hat{s}$ and $\hat{t}$. The neighbours of $\hat{s}$, $\hat{s}$ itself and $\hat{t}$ are marked as visited. Afterwards, we start directed depth first searches at all neighbours of $\hat{s}$. We add the encountered edges to U if the target has not been marked as visited yet. These depth-first searches can be performed in $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ time. We will use this intermediate subgraph U in order to construct a large feasible upward planar subgraph by trying to reinsert the remaining edges. Beforehand, we prove that U is already a feasible upward planar subgraph in the following lemma:

Lemma 5.2. *The subgraph U of $\widehat{G}$ is a feasible upward planar subgraph.*

Proof. Obviously, the subgraph U is an upward planar s-T-graph, because of its construction via a sequence of depth-first searches. All internal faces have the super sink $\hat{t}$ as sink. Thus, when constructing the merge graph $\mathcal{M}(\Gamma)$ for an embedding Γ of U, we start with G and only add edges that end at the super sink $\hat{t}$. Thus, the edge insertions cannot create a cycle in $\mathcal{M}(\Gamma)$. Using Lemma 5.1 we conclude that U is a feasible upward planar subgraph. □

After computing the intermediate feasible upward planar subgraph U, we try to enlarge it in terms of the number of contained edges. To this end, we treat the edges of $\widehat{A}$ that are not in U one by one. We add them to U and check whether it remains a feasible upward planar subgraph. Otherwise, the edge insertion is undone. After treating all remaining edges we have a large feasible upward planar subgraph U of $\widehat{G}$.
We now discuss the time complexity of the feasible upward planar subgraph computation. We already stated that the depth-first searches run in $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ time. Furthermore, there are $\mathcal{O}(|\widehat{A}|)$ edges to reinsert. For each edge we perform one upward planarity check in $\mathcal{O}(|\widehat{V}|)$ time [BDMT98] and one check upon acyclicity in $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ time. Since $\widehat{G}$ is connected, $\mathcal{O}(|\widehat{V}|) \subseteq \mathcal{O}(|\widehat{A}|)$ holds. Thus, the feasible upward planar subgraph can be computed in $\mathcal{O}(|\widehat{A}|^2)$ and $\mathcal{O}(|\widehat{A}|)$ space.

After computing a feasible upward planar subgraph, we assign costs to all its edges. We distinguish between two types of edges. The edges incident to the super source $\hat{s}$ or the super sink $\hat{t}$ have cost δ. All other edges are assigned cost α. Then, we reinsert the missing edges.

Using this approach, we optimise two summands of the cost function stated in Section 3.4:

$$\alpha \cdot \#\text{edge crossings} + [\ldots +]\ \delta \cdot \text{total source/sink distance}$$

To see that we optimise the summand $\alpha \cdot \#$edge crossings is easy: Each original edge of G has cost α such that crossing it increases the total cost by α. The optimisation of $\delta \cdot$ total source/sink distance is modelled by the edges adjacent to $\hat{s}$ and $\hat{t}$. Since $\hat{s}$ and $\hat{t}$ are on the outer face of the embedding Γ of U these are the imaginary edges we mentioned in Section 3.3. As we assign cost δ to each edge adjacent to $\hat{s}$ or $\hat{t}$, the costs induced by these crossings equals to $\delta \cdot$ total source/sink distance.

As already mentioned in the last section, layer-free upward crossing minimisation is open for randomisation. We randomise it at two points: (i) For the construction of the feasible upward planar subgraph we randomise the order in which we visit the outgoing edges during the depth-first searches. Furthermore, we randomise the order of insertion of edges that are not in the initial feasible upward planar subgraph. (ii) We randomise the order in which the missing edges are routed.

In this section we described how an upward planar representation $\mathcal{U}$ of $\widehat{G}$ can be computed such that the number of crossings and the total source/sink distance is heuristically minimised. In the following section we will present three techniques how to calculate the shape of $\widehat{G}$, i.e. the bends on each edge, if an upward planar representation is given. In Section 5.4 we then compute the length of each edge segment and come to a final layout of an argument map.

5.3 Shape

In this section we explain the second step of the topology-shape-metrics framework, i.e. how to compute the shape of a layout if an upward planar representation $\mathcal{U}$ of $\widehat{G}$ is given. In this context shape denotes the number of bends on each edge and their bend directions. Since we deal with orthogonal edges, we only allow 90° bends to the left or to the right.
We present three algorithms for shape computation. All algorithms have in common that they lead to column-based layouts. This means that the canvas is separated into disjoint columns that are as broad as the boxes. Nodes and edges are later positioned within these columns. Thereby, vertical edge segments run within a column, whereas the horizontal edge segments span over several columns.
An edge with two bends consists of two vertical segments which are necessarily positioned in the source's and the target's column, respectively. Thus, as long as we deal with 2-bend shapes, it is sufficient to assign the nodes to columns. However, edges having four bends consist of three vertical segments and hence we need to determine where to position the middle vertical segment. To this end, we assign the edge, i.e. the middle vertical segment, to a column as well. The three algorithms presented in this section compute such a *column assignments* for the nodes (and edges) of the respective input graph:

Definition 5.8 (Column Assignment). *In the case of 2-bend shapes, a* column assignment *of a DAG $G = (V, A)$ is a mapping* $\text{col} : V \to \mathbb{Z}$ *which assigns each node $v_i \in V$ to a column* $\text{col}(v_i)$. *If dealing with 4-bend shapes* $\text{col} : V \cup A \to \mathbb{Z}$ *assigns nodes and edges of G to columns.*

Note that a column assignment already induces the shape, i.e. which edges contain bends to which direction. In the metrics steps we will realise the computed column assignment, i.e. we compute a layout that respects the column assignment:

Definition 5.9 (Realisation of a Column Assignment)**.** *Let L be a valid layout of a DAG $G = (V, A)$, i.e. L satisfies all constraints listed in the formal problem statement* Argument Map. *Layout L is a* realisation *of a column assignment* col*, if we can divide the canvas into disjoint columns of uniform width and number the columns from left to right such that each node $v_i \in V$ (and edge $e_j \in A$) is positioned within column* $\mathrm{col}(v_i)$ *(and* $\mathrm{col}(e_j)$*, respectively).*

Now, we give a short introduction to the three algorithms for shape computation:

2-Bend Shape

The first algorithm computes layouts of an upward planar representation $\mathcal{U}$. These layouts have at most two bends per edge of $\mathcal{U}$. Thus, if an edge $a \in \widehat{A}$ is in $\mathcal{U}$ represented by a path p of length $\ell(p)$, then there are at most $2 \cdot \ell(p)$ bends on a. This algorithm draws the graph face by face and has linear time complexity. However, as the resulting layouts look confusing, this algorithm is only of theoretical interest.

4-Bend Shape

The second algorithm takes up the idea of Biedl and Kant how to draw a graph with few bends in linear time and adapts it to orthogonal upward drawings [BK94]. Again, we compute a layout for an upward planar representation $\mathcal{U}$. Thus, the upper bound of at most four bends per edge only holds for the edges of $\mathcal{U}$. An arc $a \in \widehat{A}$ that is in $\mathcal{U}$ represented by a path p has at most $4 \cdot \ell(p)$ bends. Although the computed layouts are better than the 2-bend drawings, they are still not good enough for practical use: Edges that are split by crossings dummies have many bends and, thus, they are hard to trace with the eyes. We only present this algorithm, because it forms the base for the third algorithm we developed.

4-Bend Shape with Topology Modifications

The third approach is similar to the second one but directly computes a shape of $\widehat{G}$ instead of the upward planar representation $\mathcal{U}$. Thereby, it does not fully respect the given upward planar representation. It only respects the left to right order of outgoing at each node $v_i \in \widehat{V}$ prescribed by $\mathcal{U}$. Thereby, the given topology is relaxed in some points, i.e. the edge crossings are not fixed anymore but can move to other edges. The resulting layouts have at most four bends per edge of $\widehat{G}$ and can be computed in linear time.

The remainder of this section is structured as follows: In Sections 5.3.1-5.3.3 we present the three shape algorithms. We conclude this section with a comparison of them in Section 5.3.4.

5.3.1 2-Bend Shape

In this section we present an algorithm that computes the shape of an upward planar representation $\mathcal{U}$ face by face from left to right. Initially, we draw the edges on the leftmost path from $\hat{s}$ to $\hat{t}$. Afterwards, we handle the faces of $\mathcal{U}$ one by one from left to right. When treating a face, we only need to draw its right contour, because its left contour is already drawn. The algorithm basically consists of two steps: (i) computation of the face insertion order and (ii) assignment of the nodes to columns. The following two paragraphs are dedicated to these two steps.

The Face Insertion Order

Before describing how to compute the face insertion order, we need to introduce the term *contour*:

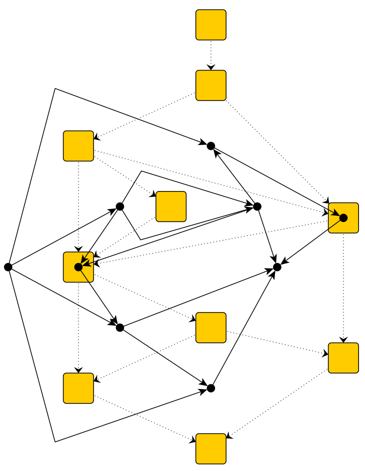
Definition 5.10 (Contour). *Given an upward planar representation $\mathcal{U}$ and a face f of $\mathcal{U}$ the left (right) contour of f is the path from f's source to its sink that needs to be layouted on the left (right) side of the face due to the edge order prescribed by $\mathcal{U}$. Note that the source and the sink of a face are neither part of its left nor of its right contour.*

The algorithm for computing the shape processes the faces of $\mathcal{U}$ from left to right, i.e. a face f_i can only be treated if all faces adjacent via edges on the f_i's left contour have been handled before. Beside this *left-right property* the face order needs to fulfil the *top-down property* as well – the top-down property prescribes, that for each node v the faces having v as sink are handled before the faces of which v is the source. For the computation of such an order, we construct an auxiliary graph $\mathcal{D}$. A topological order of the nodes in $\mathcal{D}$ will imply an order of the faces that satisfies both, the left-right property as well as the top-down property.
The graph $\mathcal{D}$ is the combination of $\mathcal{D}_{lr} = (V_{lr}, E_{lr})$ enforcing the left-right property and $\mathcal{D}_{td} = (V_{td}, E_{td})$ which ensures the top-down property. In the following, we describe the construction of $\mathcal{D}_{lr}$ and $\mathcal{D}_{td}$. Let $\mathcal{D}_{lr} = (V_{lr}, E_{lr})$ be the dual graph of $\mathcal{U}$ and direct all dual edges such that the sources are on the left side of the primal edge and the targets are on the right side, i.e. the edges are directed rightwards. Let v_0 be the dual node of the external face of $\mathcal{U}$ and remove all incoming edges of v_0.
For the construction of $\mathcal{D}_{td}$ we start with $V_{td} = V_{lr}, E_{td} = \emptyset$ and add a node $c(v)$ for each node v of $\mathcal{U}$ that is source of at least one face and sink of at least one face. We build up E_{td} by inserting edges from $c(v)$ to the dual nodes of all faces of which v is the source. Analogously, we add edges from the dual nodes of faces of which v is the sink to $c(v)$.
Obviously, $\mathcal{D}_{lr}$ and $\mathcal{D}_{td}$ enforce the left-right property and the top-down property, respectively. The auxiliary graph $\mathcal{D}$ is the union of $\mathcal{D}_{lr}$ and $\mathcal{D}_{td}$ graphs and hence enforces both properties. In Figure 5.5a we depict $\mathcal{D}$ for a simple upward planar representation $\mathcal{U}$.
After the construction of $\mathcal{D}$ we compute a topological order of its nodes. Such an order exists if and only if $\mathcal{D}$ is acyclic. As shown in Lemma 5.3 graph $\mathcal{D}$ is acyclic and, thus, such an order can be computed. We then transform the order of the nodes of $\mathcal{D}$ to a face order of $\mathcal{U}$ which fulfils both, the left-right property and the top-down property. For pseudocode see Algorithm 1. Note that the construction of $\mathcal{D}$ as well as the computation of the topological order, can be implemented to run in $\mathcal{O}(|V_{\mathcal{U}}|)$ time as we are dealing with planar graphs.

Lemma 5.3. *The graph $\mathcal{D}$ is acyclic.*

Proof. In this proof, we assume that $\mathcal{D}$ contains a cycle C and then construct a directed cut C' containing an edge of C. Thereby, we deduce a contradiction, because an edge of a cycle cannot be part of a directed cut.
We denote the cycle in $\mathcal{D}$ by C. Since the two graphs $\mathcal{D}_{lr}$ and $\mathcal{D}_{td}$ are acyclic, C contains edges from both sets, E_{lr} and E_{td}. Then there exists a subpath $(x, c(v_i), y)$ of C with $x, y \in V_{lr}, V_{td}$ and $c(v_i) \in V_{td}$. We now construct the directed cut C' in $\mathcal{D}$ which contains edge $(x, c(v_i))$. For the construction of C' we first compute a borderline path p in $\mathcal{U}$ which is depicted in Figure 5.5b. Starting at $c(v_i)$ we traverse $\mathcal{U}$ to the top always taking the rightmost incoming edge. Analogously, we traverse $\mathcal{U}$ to the bottom by always choosing the leftmost outgoing edge. Let C' contain all edges in $\mathcal{D}$ that are dual to edges on p. Furthermore, p can contain nodes v_j that have a correspondence $c(v_j)$ in $\mathcal{D}_{td}$. Let C' contain the incoming edges of these $c(v_j)$ as well. It is easy to see that the sources of all

edges in C' are on the same side, i.e. C' is a directed cut. Thus, no edge in C' can belong to a cycle in $\mathcal{D}$ which contradicts to $(x, c(v_i)) \in C$. □


(a) Graph $\mathcal{D}$ (round nodes and solid edges) used for the computation of the face order of $\mathcal{U}$ (rectangular nodes and dotted edges).

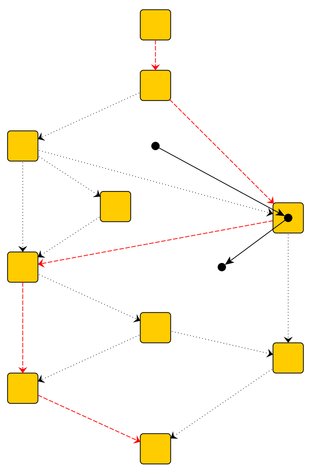

(b) The borderline path p (red dashed) for the subpath $(x, c(v_i), y)$ (black solid) of cycle C.

Figure 5.5: The graph $\mathcal{D}$ and a borderline path in $\mathcal{U}$.

Algorithm 1: Face Order

```
Input   : Upward planar representation $\mathcal{U}$
Output: Left to right order of the internal faces $f_1, \dots, f_k$
1  $\mathcal{D} \leftarrow$ dual graph of $\mathcal{U}$
2  Direct all edges in $\mathcal{D}$ rightwards
3  $v_0 \leftarrow$ the dual node of the outer face of $\mathcal{U}$
4  Remove all incoming edges of $v_0$
5  For $v$ node of $\mathcal{U}$, $v$ is source of at least one face and sink of at least one face do
6      Add new node $c(v)$ to $\mathcal{D}$
7      For $f$ face of $\mathcal{U}$ such that $v$ is source of $f$ do
8          Add edge from $c(v)$ to the dual node of $f$
9      For $f$ face of $\mathcal{U}$ such that $v$ is sink of $f$ do
10         Add edge from the dual node of $f$ to $c(v)$
11 $v_0, \dots, v_l \leftarrow$ topological order of the nodes of $\mathcal{D}$
12 $k \leftarrow 1$
13 For $i$ from 1 to $l$ do
14     If $v_i$ is the dual node of a face in $\mathcal{U}$
15         $f_k \leftarrow$ primal face of $v_i$
16         $k \leftarrow k + 1$
```

Column Assignment of the Nodes

Having computed the face insertion order, we start to draw the upward planar representation $\mathcal{U}$. Initially, we assign $\hat{s}$, $\hat{t}$ and all nodes on the left contour of the external face to column 0. Then, we treat the faces according to the computed order. When treating a face f_i we only need to draw the right contour. Due to the left-right property, the left contour has already been drawn in an earlier iteration. We now distinguish two cases depending on the number of nodes on the right contour of f_i: In Case (a) there is at least

one node on the right contour of f_i. In this case, we assign all nodes on the right contour to column i.
However, consider Case (b), i.e. that the right contour of f_i is a single edge. Furthermore, assume, that the nodes on the left contour, the source and the sink of f_i are all assigned to the same column. The remaining edge cannot be inserted with two bends as it needs to start and end in the same column but must circumvent the nodes in between, i.e. it would require four bends. Thus, we move the sink of f_i to column i, which has been empty up to now.
It is not obvious, that the movement of a sink in Case (b) does not introduce overlapping of edges or of edges and boxes to the final layout. In the following paragraph we show that the column assignments computed by Algorithm 2 can be realised in terms of Definition 5.9.
After all faces have been processed, the algorithm terminates and returns the column assignment of the nodes. We present the pseudocode of this method as Algorithm 2. Since the number of faces and edges of a planar graph is linear in the number of its nodes, Algorithm 2 runs in $\mathcal{O}(|V_{\mathcal{U}}|)$. As already discussed Algorithm 1 has the same time complexity. Thus, the 2-bend shape as a whole can be computed in linear time.

Algorithm 2: 2-Bend Column Assignment

```
Input : Upward planar representation $\mathcal{U}$,
        Face order $f_1, \dots, f_k$ computed by Algorithm 1
Output: Column assignment for each node of $\mathcal{U}$
// Initialisation
1  col($\hat{s}$) ← 0
2  col($\hat{t}$) ← 0
3  $f_0$ ← external face of $\mathcal{U}$
4  For $v$ on the left contour of $f_0$ do
5      col($v$) ← 0
// Process faces and assign the nodes to columns
6  For $i$ from 1 to $k$ do
7      If right contour of $f_i$ contains at least one node        // Case (a)
8          For $v$ on the right contour of $f_i$ do
9              col($v$) ← $i$
10     Else                                                          // Case (b)
11         $s$ ← sink of $f_i$
12         col($s$) ← $i$
```

Correctness

In this paragraph we show that column assignments computed by Algorithm 2 can be used in order to compute a final layout of $\mathcal{U}$, i.e. the column assignment can be realised.

As long as all faces of $\mathcal{U}$ have at least one node on the right contour, i.e. Case (b) in Algorithm 2 does never occur, the column assignment can easily be realised by using an s-t-ordering of the nodes in $\mathcal{U}$ in order to compute the y-coordinates of the boxes (see Lemma 5.4 for details).
However, if Case (b) occurs during the execution of Algorithm 2, i.e. we move the sink of a face f_i to the right, it is not obvious anymore that the computed column assignment can be realised. In Lemma 5.4 we show that this is possible as long as the y-coordinates of the nodes of $\mathcal{U}$ correspond to a right-first s-t-order. Such an order is computed by Algorithm 3 which traverses $\mathcal{U}$ using a right-first depth-first search, i.e. the outgoing edges of each node

are treated from right to left. Since each node is visited as often as there are incoming edges and as $\mathcal{U}$ is planar the runtime of this algorithm is linear in $|V_{\mathcal{U}}|$.

Algorithm 3: Vertical Order

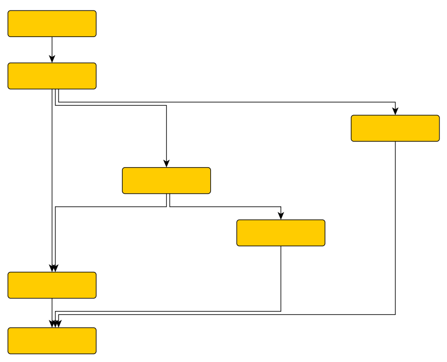
```
Input   : Upward planar representation $\mathcal{U}$
Output: Vertical order $v_0, \dots, v_{n-1}$ of the nodes of $\mathcal{U}$
$\mathcal{S} \leftarrow$ stack containing source of $\mathcal{U}$
$i \leftarrow 0$
while $\mathcal{S}$ not empty do
    $n \leftarrow \mathcal{S}$.top()
    If n is not marked
        If all predecessors of v are marked
            Mark $v$
            $v_i \leftarrow n$
            $i \leftarrow i + 1$
        Else
            $\mathcal{S}$.pop()
            Continue
    If there is a next right most successor $n'$ of $v$
        $\mathcal{S}$.push($n'$)
    Else
        $\mathcal{S}$.pop()
```

Lemma 5.4. *The column assignments computed by Algorithm 2 can be realised.*

Proof. Since the x-coordinates of the boxes are already induced by the column assignment, we only need to discuss the computation of the y-coordinates in this proof.

First, assume that all faces $f_i, i = 0, \dots, k$ have at least one node on the right contour. Thus, Case (b) in Algorithm 2 does never occur. Then, the vertical arrangement of the nodes necessarily corresponds to an s-t-ordering of the nodes in $\mathcal{U}$ in order to preserve the upward drawing constraint. The edges can easily be routed such that they do not overlap each other or cut through boxes by drawing the horizontal segment of the first (last) edge on the right contour as high (low) as possible. All other edges can be drawn as straight lines. Figure 5.6a shows such a layout.

If in Algorithm 2 Case (b) occurs, we need to show, that after the movement of the sink the column assignment can still be completed to a valid layout where no edges overlap each other or cut through boxes. Firstly, consider the outgoing edges of the sink. Because of the top-down property at most one outgoing edge of the sink can already be processed at this time. This edge can easily be routed by drawing its horizontal segment as high as possible.
Now, consider the incoming edges of the moved sink. Note that for each of these edges the source is assigned to a column with an index less than i, i.e. all incoming edges have a horizontal segment that is directed rightwards. Consider a single incoming edge e. Let $f_l(e)$ be the face to the left side of e. We denote the set of nodes on the left contour of $f_l(e)$ by $N_l(e)$. Analogously, we define $f_r(e)$ and $N_r(e)$. In order to be able to draw the incoming edges for each incoming edge e the following property needs to hold: The nodes in $N_l(e)$ must be positioned below the nodes in $N_r(e)$ (see Figure 5.6b for illustration). A right-first s-t-order satisfies this property.
We conclude that the column assignments computed by Algorithm 2 can be realised. □


(a) Case (b) does never occur.

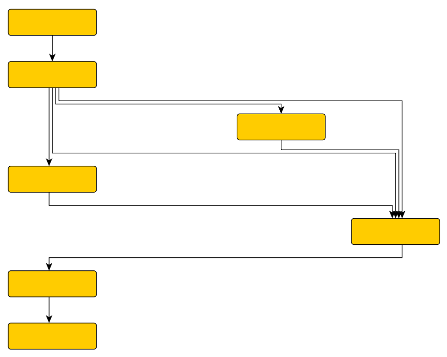
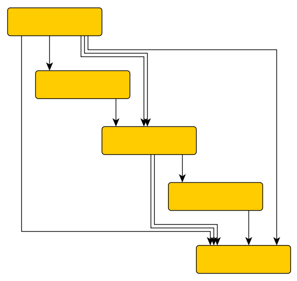

(b) Case (b) occur es two times.

Figure 5.6: Realisations of the 2-bend shape column assignment.

The realisations of the column assignments computed by Algorithm 2 do not fulfil our expectations concerning the aesthetics of layout. Figure 5.6 shows that the successors of a node are not evenly distributed to its left and right side, but most outgoing edges bend to the right. Therefore, we do not use this algorithm for the computation of argument map layouts but present it because of its bound of two bends per edge in $\mathcal{U}$.

2-Bend Drawings are not Bend Minimal

As seen in the foregoing section for each upward planar representation $\mathcal{U}$ there exists a layout with at most two bends per edge in the upward planar representation. In this section we show that these layouts are not bend minimal in the global sense, i.e. the total number of bends can be decreased by allowing more than two bends per edge. In Figure 5.7a we show a layout with at most two bends per edge. Among the layouts with at most two bends per edge this one is bend minimal. However, there exists a layout with less bends in total but this layout requires four bends on two edges. This layout is depicted in Figure 5.7b. We conclude that restricting to two bends per edges conflicts with global bend minimisation.


(a) At most two bends per edge.

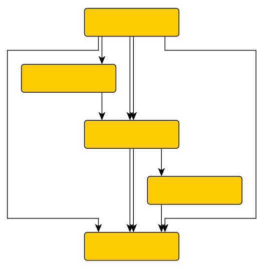

(b) Bend minimal layout.

Figure 5.7: Layouts having only two bends per edge are not bend minimal.

5.3.2 4-Bend Shape

In this section we present an algorithm that computes a shape of an upward planar representation $\mathcal{U}$ such that there are at most four bends per edge. Since we allow four bends per edge, the resulting column assignment assigns both, nodes and edges, to columns.
We take up the idea by Biedl and Kant how to draw a graph with few bends [BK94] and

adapt it to orthogonal upward planar drawings of DAGs. This algorithm runs in linear time as well but is fundamentally different to the 2-bend shape algorithm presented in the previous section. Instead of treating the faces of $\mathcal{U}$ one by one, we now handle the nodes of $\mathcal{U}$ according to an s-t-order. In contrast to Section 5.3.2 this time the s-t-order has no significance for the final layout – we only treat the nodes according to this order. Thus, we start with the super source $\hat{s}$ and assign it to column 0. Afterwards, we assign the outgoing edges of $\hat{s}$ according to the left to right order prescribed by $\mathcal{U}$ such that they are evenly distributed to the left and right of column 0.
The invariant of this algorithm is that all incoming edges of v_i are already assigned to columns, when treating v_i itself. Then we assign v_i to column m – the column of the median incoming edge – and assign the outgoing edges of v_i to columns. However, the columns to the left and right of m are possibly already occupied by other edges. Therefore, we shift all columns left (right) of m to the left (right) such that there are out-degree(v_i) empty columns which can be assigned with the outgoing edges of v_i according to the left to right order prescribed by the upward planar representation $\mathcal{U}$. Thus, in contrast to the 2-bend shape, we evenly distribute the outgoing edges of v_i to the left and right of the column assigned to v_i. After all nodes have been treated the algorithm terminates and returns the column assignment for nodes and edges. For pseudocode see Algorithm 4.

Since we assign nodes and edges to columns, each edge can contain at most two horizontal edge segments – one spanning from the source's column to the edge's column and one spanning from the edge's column to the target's column. Thus, there are at most four bends per edge.

Lemma 5.5. *The column assignment computed by Algorithm 4 induces at most four bends per edge of the upward planar representation $\mathcal{U}$.*

Furthermore, Algorithm 4 positions the nodes and edges such that incoming and outgoing edges of each node v_i are symmetrically distributed to the columns left and right of the column assigned to v_i. Thus, the resulting layouts stand out due to a high degree of symmetry. We denote this property by *local symmetry.*

Lemma 5.6. *In a column assignment* col *computed by Algorithm 4 for each node $v_i \in \widehat{V}$ the column* $\mathrm{col}(v_i)$ *is the median of the columns assigned to the incoming edges of v_i as well as the median of the columns assigned to its outgoing edges.*

Again, we prove that the computed column assignments are realisable:

Lemma 5.7. *The column assignments computed by Algorithm 4 can be realised.*

Proof. A layout of $\mathcal{U}$ can be constructed inductively during the execution of Algorithm 4. Since the x-coordinates are fixed by the column assignment, we only need to cope with the y-coordinates.
Initially, we set the y-coordinate of $\hat{s}$ to 0 and draw the first vertical and horizontal segments belonging to the outgoing edges of $\hat{s}$ as high as possible.
When treating v_i we draw the middle vertical segments of all incoming edges of v_i such that they reach farther down than any other edge or box in the intermediate layout. Then we append the second horizontal segments of these edges and position the box v_i. Afterwards, we draw the first vertical and horizontal segment of the outgoing edges of v_i as high as possible. We present such a realisation in Figure 5.8. □

Furthermore, we prove that a realisations of a column assignments of an upward planar representation $\mathcal{U}$ that is computed by Algorithm 4 is upward planar:

Lemma 5.8. *The realisations of the column assignments of an upward planar representation $\mathcal{U}$ that are computed by Algorithm 4 are upward planar layouts.*

Proof. Again, we present an inductive proof. When positioning v_i and drawing the first segments of the outgoing edges, we cannot add crossings to the layout, because we shifted all other boxes and edges to the left and right, respectively. Thus, we draw the outgoing edges in an empty region.
Note that the left-to-right order of the incoming edges of a node v_i that is induced by the column assignment coincides with the left to right order prescribed by $\mathcal{U}$. Thus, when treating v_i we can draw the remaining segments of the incoming edges without introducing crossings. Since this holds for all v_i the realisation of the column assignments computed by Algorithm 4 are upward planar layouts. □

Algorithm 4: 4-Bend Column Assignment

```
Input  : Upward planar representation $\mathcal{U}$,
         $s$-$t$-order $v_0, \dots, v_{n-1}$ of the nodes in $\mathcal{U}$
Output: Column assignment for each node and each edge of $\mathcal{U}$
1  For $i$ from $0$ to $n-1$ do
2      $\mathcal{I} \leftarrow$ column indices assigned to incoming edges of $v_i$
3      $m \leftarrow$ median of $\mathcal{I}$
4      $\mathrm{col}(v_i) \leftarrow m$
5      Shift Left$(\mathcal{U}, m, \lfloor \text{out-degree}(v_i)/2 \rfloor)$
6      Shift Right$(\mathcal{U}, m, \lfloor (\text{out-degree}(v_i) - 1)/2 \rfloor)$
7      $j \leftarrow m - \lfloor \text{out-degree}(v_i)/2 \rfloor$
8      For outgoing edge $e$ of $v_i$ from left to right do
9          $\mathrm{col}(e) \leftarrow j$
10         $j \leftarrow j + 1$
```

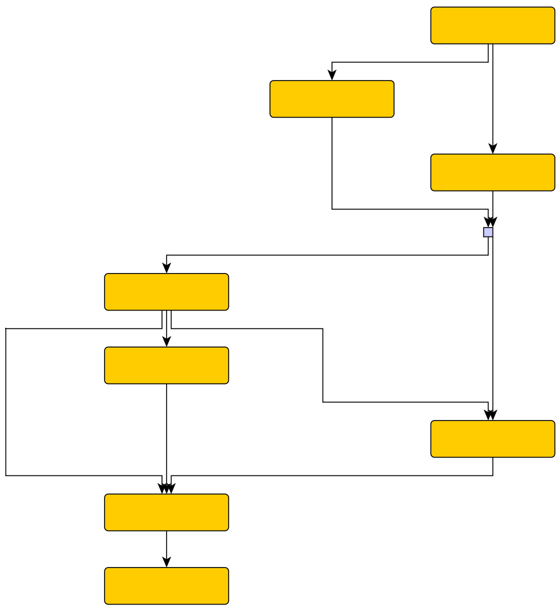


Figure 5.8: A layouted that is based on the 4-bend shape algorithms (the small square node is a crossing dummy).

Runtime Analysis

Algorithm 4 requires an s-t-order of the nodes in $\mathcal{U}$ as input. Such an order can be computed in linear time [ET76, ET77, Bra02]. The time complexity of Algorithm 4 itself depends on the implementation of the two methods Shift Left and Shift Right, which are called $|V_{\mathcal{U}}|$ times.
In a greedy implementation each box and each edge would be assigned with a column index. When calling Shift Left or Shift Right each box and edge needs to be checked and possibly modified. Thus, the two methods would run in $\mathcal{O}(|V_{\mathcal{U}}| + |A_{\mathcal{U}}|) = \mathcal{O}(|V_{\mathcal{U}}|)$ yielding an overall runtime of $\mathcal{O}(|V_{\mathcal{U}}|^2)$ of Algorithm 4.
Biedl and Kant explain how the time complexity of their approach can easily be reduced to $\mathcal{O}(|V_{\mathcal{U}}|)$ [BK94]. They represent the columns by a doubly linked list. Each box and edge has a pointer to the column to which it is assigned. Since the position at which the new columns shall be inserted are known, Shift Left and Shift Right can then be implemented in constant time.
Nevertheless, we need to be careful about the computation of the median of the incoming edges in Line 3. The column indices that are assigned to the incoming edges of v_i induce an order of these edges. The upward planar representation $\mathcal{U}$ prescribes an order of these edges as well. Since $\mathcal{U}$ is planar, these two orders are equal. Thus, we can select the median m without taking the assigned columns into considerations. The computation only considers the edge order prescribed by $\mathcal{U}$ and, therefore, can be done in constant time.
Thus, Algorithm 4 can be implemented as an $\mathcal{O}(|V_{\mathcal{U}}| + |A_{\mathcal{U}}|)$-algorithm, which equals to $\mathcal{O}(|V_{\mathcal{U}}|)$, because $\mathcal{U}$ is planar.

5.3.3 4-Bend Shape with Topology Modifications

In the last section we described how to compute a column assignment of nodes and edges of an upward planar representation $\mathcal{U}$. However, for an edge $e \in \widehat{A}$ and p its representing path in $\mathcal{U}$, the number of bends on e is bounded by $4 \cdot \ell(p)$, where $\ell(p)$ is the number of edges on p. This can lead to inelegant layouts, because edges with many bends are hard to trace with the eyes. To this end, we now present an algorithm that computes a column assignment for $\widehat{G}$ such that all edges in $\widehat{A}$ have at most four bends, i.e. the bound on the number of bends is independent of the topology computed in Section 5.2. Basically, we apply the same technique as in Section 5.3.2. However, $\widehat{G}$ contains no information about the left to right order of the outgoing edges of a node. Instead, this order is taken from an upward planar representation $\mathcal{U}$ of $\widehat{G}$. For edges $e_i \in \widehat{A}$ that have a direct correspondence $e'_i \in A_{\mathcal{U}}$, we use the ordering index of e'_i that is prescribed by $\mathcal{U}$. However, there are edges e_i of $\widehat{G}$ that have no direct correspondence in $\mathcal{U}$, because they are split into several edges due to crossing dummies. Then e_i is represented by a path $p \in \mathcal{U}$. In this case we consider the ordering index of the first edge on p. For pseudocode see Algorithm 5.

Lemmas 5.5-5.7 – stated in the context of Algorithm 4 – hold for Algorithm 5 with the same argumentation as in the previous section. Putting them together, we can prove the following theorem:

Theorem 5. *The column assignments computed by Algorithm 5 are locally symmetric and can be realised in layouts that have at most four bends per edge of $\widehat{G}$.*

Note, that for an upward planar representation $\mathcal{U}$ containing no crossing dummies, the two algorithms presented in this and in the previous section are the same. Since the realisations of the column assignments computed by 4 are upward planar (see Lemma 5.8), the following lemma holds:

Lemma 5.9. *If the upward planar representation $\mathcal{U}$ contains no crossing dummies, the realisations of the column assignments computed by Algorithm 5 are upward planar as well.*

Since we do not respect the whole upward planar representation in Algorithm 5 anymore, we relax the topology which we computed in the first step. We exemplify this in Figure 5.9. It depicts a realisation of the column assignment computed by Algorithm 5. It is the same input instance as in Figure 5.8. However, the edge crossings in Figure 5.9 do not correspond to the crossing dummies in Figure 5.8. In fact there is one crossing more than there have been crossing dummies. Nevertheless, Figure 5.9 looks clearer and more well-arranged than Figure 5.8.
In particular, the column assignments computed by Algorithm 5 can be realised in layouts that have different topologies. We illustrate this in Figure 5.10. In Figure 5.10a we present an upward planar representation $\mathcal{U}$ of a graph. We compute a shape for this upward planar representation using Algorithm 5. The resulting column assignment can be realised in layouts that differ in their topology (see Figure 5.10b).
Due to the same reason a source whose edge to $\hat{s}$ contains no crossing dummy in $\mathcal{U}$ can move to an internal face again. Figure 5.11 illustrates such a situation. In Figure 5.11a we depict the upward planar representation of a graph. Note that "Source" is on the external face of the final layout. However, in Figure 5.11b, which shows a realisation of a column assignment computed by Algorithm 5, "Source" is positioned in an internal face.
Because of the relaxation of the topology, the final layout can have a higher number of crossings or total source/sink distance than the topology computed in Section 5.2. In Chapter 6 we analyse this increment empirically. In contrast, we can exploit the relaxation when minimising the vertical edge length in the metrics step in order to gain more compact layouts. Thus, by relaxing the topology, we increase the significance of total edge length minimisation in comparison to crossing number and total source/sink distance minimisation.

Algorithm 5: Pseudo 4-Bend Column Assignment

```
Input  : Graph $\widehat{G}$, Upward planar representation $\mathcal{U}$,
         $s$-$t$-order $v_0, \dots, v_{n-1}$ of the nodes of $\widehat{G}$
Output: Column assignment for each node and each edge of $\widehat{G}$
1  For $i$ from $0$ to $n-1$ do
2      $\mathcal{I} \leftarrow$ column indices assigned to incoming edges of $v_i$
3      $m \leftarrow$ median of $\mathcal{I}$
4      $\text{col}(v_i) \leftarrow m$
5      Shift Left($\widehat{G}, m, \lfloor \text{out-degree}(v_i)/2 \rfloor$)
6      Shift Right($\widehat{G}, m, \lfloor (\text{out-degree}(v_i) - 1)/2 \rfloor$)
7      $j \leftarrow m - \lfloor \text{out-degree}(v_i)/2 \rfloor$
8      For outgoing edge $e$ of $v_i$ from left to right according to $\mathcal{U}$ do
9          $\text{col}(e) \leftarrow j$
10         $j \leftarrow j + 1$
```

Runtime and Space Analysis

Although Algorithm 5 and Algorithm 4 are closely related, we cannot directly carry over the runtime analysis. The crucial difference lies in the computation of the median incoming edge in Line 3. In the context of Algorithm 4 we computed the shape of an upward planar representation, i.e. of a planar graph. Then, the order of the incoming edges of a node v_i that is induced by the column indices assigned to these edges coincides with the order that is prescribed by the upward planar representation.

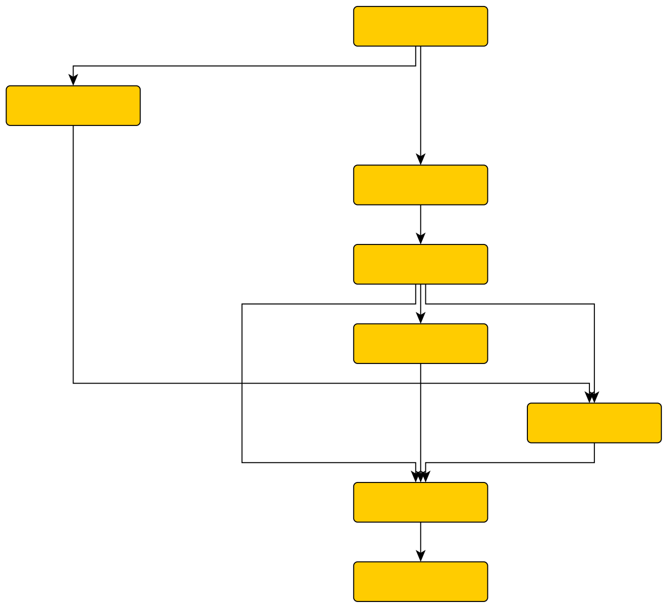
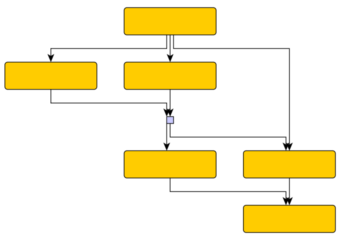
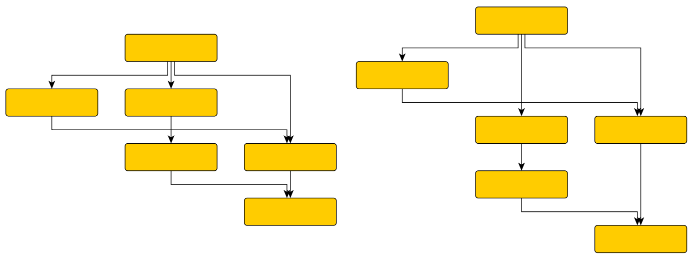


Figure 5.9: A layout that is based on the pseudo 4-bend shape algorithms.



(a) The topology computed in Section 5.2.



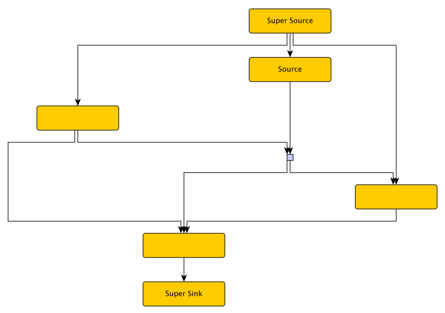
(b) The shape computed by Algorithm 5 relaxes this topology. The edge crossing represented by the crossing dummy is not fixed anymore.

Figure 5.10: The shape computed by Algorithm 5 relaxes the topology computed in Section 5.2.

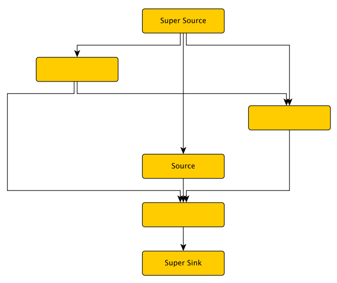
However, Algorithm 5 does not compute the shape of a planar graph. In Figure 5.8 and Figure 5.9 we exemplify that these two orders are different. We denote the unique node having in-degree two by v. In Figure 5.8, which shows the upward planar representation the edge that is split by the crossing dummy is the rightmost incoming edge of v. In a realisation of the column assignment computed by Algorithm 5 (see Figure 5.9) this edge is the leftmost incoming edge of v.
Thus, we need to consider the columns that are assigned to the incoming edges of v_i in order to compute their median m in Line 3. The median m can simply be computed using $\mathcal{O}(\text{in-degree}(v_i))$ operations [BFP+73]. We sum up over all iterations:

$$\sum_{i=0}^{n-1} \text{in-degree}(v_i) = |\widehat{A}|$$

Thus, Algorithm 5 runs in $\mathcal{O}(|\widehat{V}| + |\widehat{A}|) = \mathcal{O}(|\widehat{A}|)$ time.
Of course, we need $\mathcal{O}(|\widehat{V}|+|\widehat{A}|)$ space in order to store the assigned columns. Furthermore, we need $\mathcal{O}(b)$ space in order to store the doubly linked listed that represents the columns. Thus, in total Algorithm 5 needs $\mathcal{O}(|\widehat{V}| + |\widehat{A}| + b) = \mathcal{O}(|\widehat{A}|)$ space.



(a) The topology computed in Section 5.2. “Sink” is on the external face.



(b) Using the pseudo 4-bend shape algorithm “Sink” can move to an internal face again.

Figure 5.11: The pseudo 4-bend shape algorithm can move sources and sinks to internal faces.

5.3.4 Comparison

In the foregoing sections we described three different approaches how to compute the shape for a given upward planar representation $\mathcal{U}$ in either $\mathcal{O}(|V_{\mathcal{U}}|)$ or $\mathcal{O}(|\widehat{A}|)$ time. The actual goal we should optimise is bend minimisation (see Section 3.3). Note that none of the three algorithms minimises the number of bends to the optimum but all have an upper bound for the number of bends that are created. In Table 5.2 we oppose these bounds to each other. The 2-bend shape algorithm as well as the 4-bend shape algorithm operate on the upward planar representation $\mathcal{U}$ itself. Thus, the bounds in Table 5.2 are the product of the maximum number of bends per edge times the number of edges in $\mathcal{U}$.
In contrast the 4-bend shape with topology modifications have at most $4 \cdot |\widehat{A}|$ bends, i.e. the bound is independent of the computed upward planar representation $\mathcal{U}$. Because of this and because of the promising clearness and aesthetic of the 4-bend shapes, we decided to use 4-bend shapes with topology modifications for the shape phase of the topology-shape-metrics framework.

Shape Algorithm	**Runtime**	**Number Of Bends**
2-Bend Drawings	$\mathcal{O}(\|V_{\mathcal{U}}\|)$	$\leq 2 \cdot \|A_{\mathcal{U}}\|$
4-Bend Drawings	$\mathcal{O}(\|V_{\mathcal{U}}\|)$	$\leq 4 \cdot \|A_{\mathcal{U}}\|$
4-Bend Drawings With Topology Modifications	$\mathcal{O}(\|\widehat{A}\|)$	$\leq 4 \cdot \|\widehat{A}\|$

Table 5.2: Comparison of the three shape algorithms.

5.4 Metrics

In the last phase of the topology-shape-metrics approach, we compute the final coordinates of boxes and edges and, thereby, minimise the total edge length. The topology that we originally computed in Section 5.2 has no influence on this step. Instead, we only rely on the column assignment computed by Algorithm 5 in Section 5.3.3.
First we minimise the vertical edge length, i.e. the length of the vertical edge segments. We offer two approaches for this in Section 5.4.1. Afterwards, we minimise the horizontal edge length in Section 5.4.2. The horizontal compaction consists of two steps: (i) the treatment of so-called *bows* and (ii) the actual compaction of the width.

Note that during this process we do not drop the columns introduced while computing the shape. All operations are performed with respect to these columns such that the final layout is column based.

5.4.1 Vertical Edge Length Minimisation

In this section we minimise the vertical edge length, i.e. the sum of the lengths of all vertical edge segments. We denote this problem by VERTICAL EDGE LENGTH MINIMISATION and define it as follows:

Instance: An s-t-graph $\widehat{G} = (\widehat{V}, \widehat{A})$, a column assignment for each box $n \in \widehat{V}$ and each edge $e \in \widehat{A}$, a set of spacing constraints and an integer $k \geq 0$.
Question: Is it possible to assign y-coordinates to the boxes and edges of $\widehat{G}$ such that the resulting layout is valid and the vertical edge length is at most k?

Note that the resulting column assignment of Algorithm 5 is an instance of VERTICAL EDGE LENGTH MINIMISATION. In this section, we prove that VERTICAL EDGE LENGTH MINIMISATION is $\mathcal{NP}$-complete. Afterwards, we give an overview about two heuristic approaches that minimise vertical edge length for a given shape. The first one is mainly of theoretical interest, because it is complicated to implement. Therefore, we suggest another one – a greedy approach – as well. We use the greedy approach in our implementation in order to minimise vertical edge length.

- **Network Flow:** In the original version of the topology-shape-metrics approach and in many papers basing upon Tamassia's work the final coordinate assignment is inspired by a network flow [Tam87]. We adapt this idea and show how the lengths of the vertical edges can be minimised while the constraints on box dimensions, spacing constraints and alignment of predecessors are enforced. However, this approach has a significant drawback, namely, for the construction of the network we need a fixed topology. Algorithm 5 relaxes the topology computed in Section 5.2 and does not fix a new one. Therefore, we need to select one of the possible topologies before the construction of the network. Choosing the wrong one can require unnecessary edge length. However, we have no strategy to select a good one.

- **Greedy approach:** The second approach we present is a simple greedy heuristic. In a first step we compute the groups of boxes that need to be aligned at the bottom. Afterwards, we assign y-coordinates to each group. This approach cannot guarantee any optimality for the computed layouts. However, the open decisions concerning the topology are decided one by one along the edge length compaction. Using this approach, we compute y-coordinates that lead to good looking final layouts.

The remainder of this section is structured as follows. In Section 5.4.1.1 we present the $\mathcal{NP}$-completeness proof of VERTICAL EDGE LENGTH MINIMISATION. Afterwards, we discuss the network flow approach and the greedy approach in Section 5.4.1.2 and 5.4.1.3.

5.4.1.1 Complexity Considerations

We prove $\mathcal{NP}$-completeness of VERTICAL EDGE LENGTH MINIMISATION by reduction from 3-PARTITION (see Section 3.5.1 for its definition):

Theorem 6. VERTICAL EDGE LENGTH MINIMISATION *is $\mathcal{NP}$-complete.*

The reduction is similar to the reduction from 3-PARTITION to ARGUMENT MAP we described in Section 3.5.1. Again, we use the 3-PARTITION instance $A = \{2, 2, 2, 2, 3, 3\}$ and $B = 7$ as example.

For the Vertical Edge Length Minimisation instance we use the following minimum spacing constraints: $s^{\text{box}}_{\text{box}} = 40$, $s^{\text{edge}}_{\text{box}} = 20$, $\hat{s}^{\text{edge}}_{\text{edge}} = 20$, $s^{\text{edge}}_{\text{edge}} = 5$ and $s^{\text{port}}_{\text{corner}} = 20$. Furthermore, we require a uniform box width $w = 135$. For transforming a 3-Partition instance to a Vertical Edge Length Minimisation instance we basically construct gadgets of two types: (i) a frame with height of ca. mB and width of ca. $|A|$ and (ii) one number gadget for each $a \in A$. The frame is horizontally separated into m cells of height B. Again, these cells represent the sets S_i in the solution of the 3-Partition instance. The number gadgets are connected to the top and the bottom of the frame such that moving them up and down is a cost neutral operation. Some boxes of each number gadget are assigned to a single column. The cumulative height of these boxes represent the value of the corresponding element $a \in A$. Since a part of each number gadget is in the shared column, they need to be vertically ordered and cannot overlap each other. Furthermore, we force the number into the cells of the frame. Otherwise, i.e. in the case that an edge separating two cells crosses a number gadget, the total vertical edge length would increase due to the internal construction of number gadgets and due to the spacing constraints. If a solution of the Argument Map instance is found, we can simply reconstruct the 3-Partition solution by checking which number gadget lies within which cell. In the following we describe the construction of the frame and the number gadgets in detail.

The Frame

In Figure 5.12 we depict the frame for the instance $A = \{2, 2, 2, 2, 3, 3\}, B = 7$. The frame consists of $|A| + 3$ columns which we denote by c_i for $i = 1, \dots, |A| + 3$. Edges going from the leftmost to the rightmost column separate the whole frame into cells which correspond to the sets S_i in the solution of the 3-Partition instance. Therefore, there are m boxes of the height $H = 20 \cdot B + 100 \cdot (B - 3) + 160$ as well as two boxes of height 40 in column c_1 and $c_{|A|+3}$. There are edges among them such that their vertical order is fixed to: a small box, m high boxes, a small box. Furthermore, there is an edge from the i-th box in column c_1 to the $(i + 1)$-th box in column $c_{|A|+3}$. The horizontal edge segments spanning over the columns $c_2, \dots, c_{|A|+2}$ separate the frame into the cells which correspond to the sets S_i. Therefore, we denote them by *separating edges.*
Column c_2 is not occupied by the frame. Parts of the number gadgets will later be assigned to this column. Columns $c_3, \dots, c_{|A|+2}$ contain two boxes of height 40. At first, we assume that in each column one of the two boxes is above the highest separating edge and the other one is below the lowest separating edge. The number gadgets will be connected to these boxes. Therefore, we call them *anchors.*

The Number Gadgets

For each $a_i \in A$ we add a number gadget to the Vertical Edge Length Minimisation instance. We depict the number gadget together with two anchors in Figure 5.13. A number gadget consists of $a_i + (a_i - 1)$ boxes and is assigned to two columns. Each of these boxes has height 20. There are a_i boxes in column c_2 and $a_i - 1$ boxes in column c_{i+2}. The first and the last box of the number gadget in column c_2 are connected to the anchors in column c_{i+2}.
Note that the drawings of the number gadgets in Figure 5.13 are minimal with respect to the vertical edge length. In a valid solution of the Vertical Edge Length Minimisation instance the number gadgets need to be drawn in this way.
Since we assume that the anchors in column c_{i+2} are fixed, moving the number gadget upwards or downwards is cost neutral, because the sum of the vertical edge lengths on the two edges incident to the anchors remains unchanged.
Note that there are no edges between the number gadgets, i.e. their vertical ordering is not constrained. However, they cannot overlap, because all number gadgets occupy space

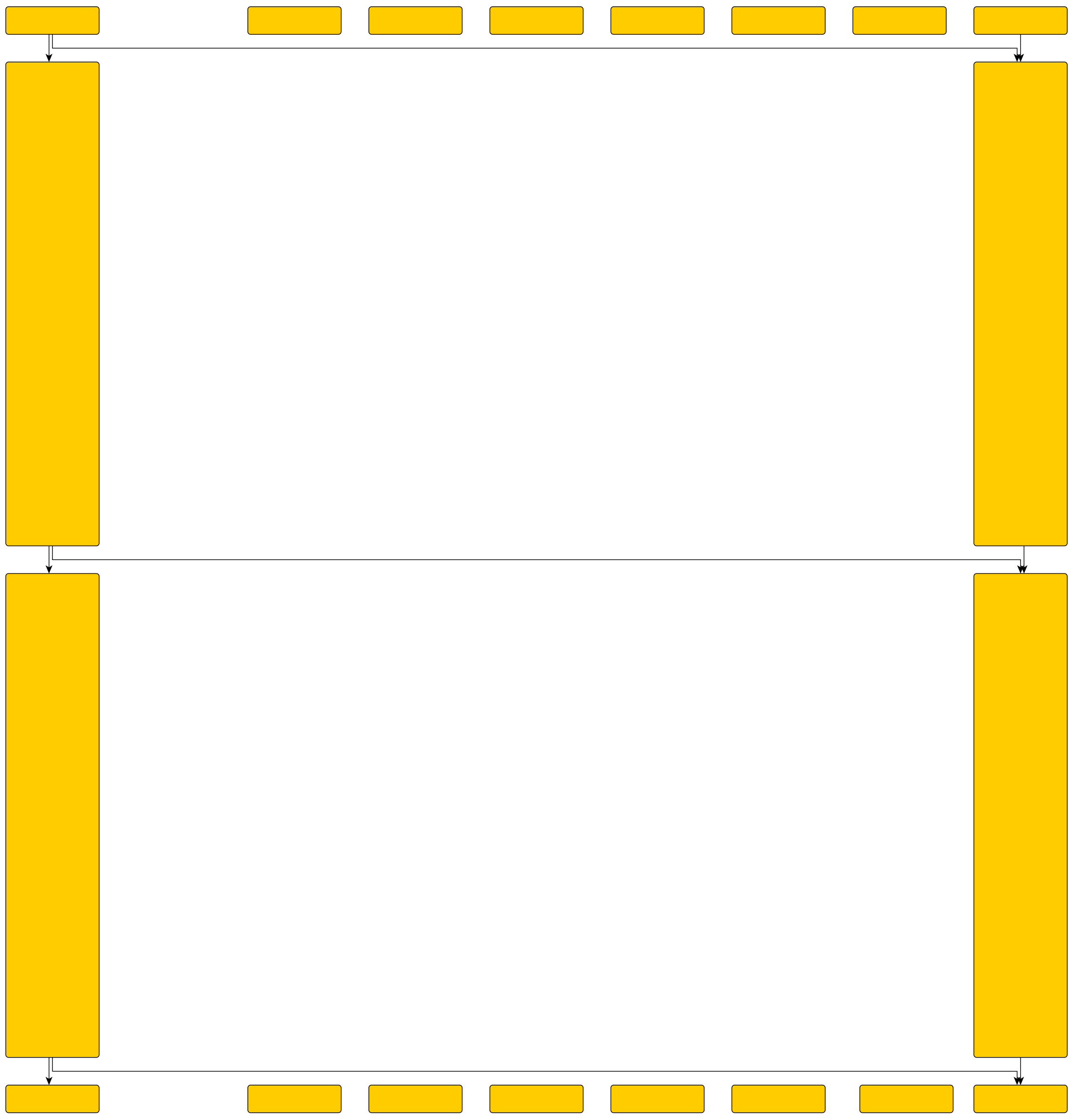
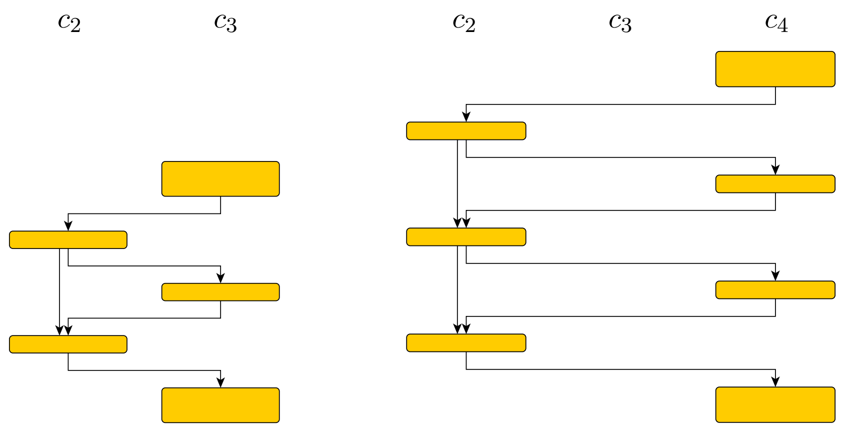


Figure 5.12: The frame of the VERTICAL EDGE LENGTH MINIMISATION instance for the 3-PARTITION instance $A = \{2, 2, 2, 2, 3, 3\}, B = 7$.

in column c_2. Number gadget a_i occupies $20 \cdot a_i + 100 \cdot (a_i - 1) + 40$ vertical units of column c_2. Summing up over the height of three number gadgets whose corresponding numbers sum up to B and taking the spacing between them into consideration, we can explain the height H of the boxes in column c_1 and $c_{|A|+3}$.
Now consider a horizontal edge segment that separates the frame into cells. If this edge segment lies above or below the number gadget, it crosses only one vertical edge. However, if it cuts through the number gadget, it crosses two edges.



(a) The number gadget for $a = 2$ and the anchors are one column apart.

(b) The number gadget for $a = 3$ and the anchors are two columns apart.

Figure 5.13: Two number gadgets for $a = 2$ and $a = 3$.

We are almost done with the transformation to the Vertical Edge Length Minimisation instance. Three things are missing: (i) the instance is not an s-t-graph yet, (ii) the assignment of edges to columns and (iii) we did not compute k. For (i) we simply add a super source and a super sink to column c_1 and connect them with all other sources and sinks, respectively. The resulting instance is shown in Figure 5.14. If this exceeds the maximum in- or out-degree of a box, we need intermediate super sources and sinks that are then connected to the real super source and sink, respectively. For (ii) we assign all edges to the column of its source. For (iii) we set k to the sum of the minimum vertical edge length (due to the spacing constraints) in the frame, in the number gadgets, on the edges incident to the anchors and on the edges incident to the super source and super sink.

Correctness

We now show, that the 3-Partition instance has a solution if and only if the Vertical Edge Length Minimisation instance is solvable. Therefore, we firstly assume that the anchors are positioned outside the cells as shown in Figure 5.14. Obviously, if the 3-Partition instance has a solution, then it can be transformed to a solution of the Vertical Edge Length Minimisation instance by vertically ordering the number gadgets. If the Vertical Edge Length Minimisation instance is solvable, we need to show, that the 3-Partition instance has a solution as well. Therefore, we show that in a solution of the Vertical Edge Length Minimisation instance each number gadget lies within exactly one cell of the frame, i.e. (i) no number gadget is outside the frame and (ii) no number gadget is crossed by a separating edge.
(i) Obviously, moving a number gadget outside the frame increases the sum of the vertical edge length on the edges connecting the number gadgets with the anchors. As this movement cannot decrease the vertical edge length somewhere else, the total vertical edge length would exceed k. Thus, this configuration would not solve the Vertical Edge Length Minimisation instance.

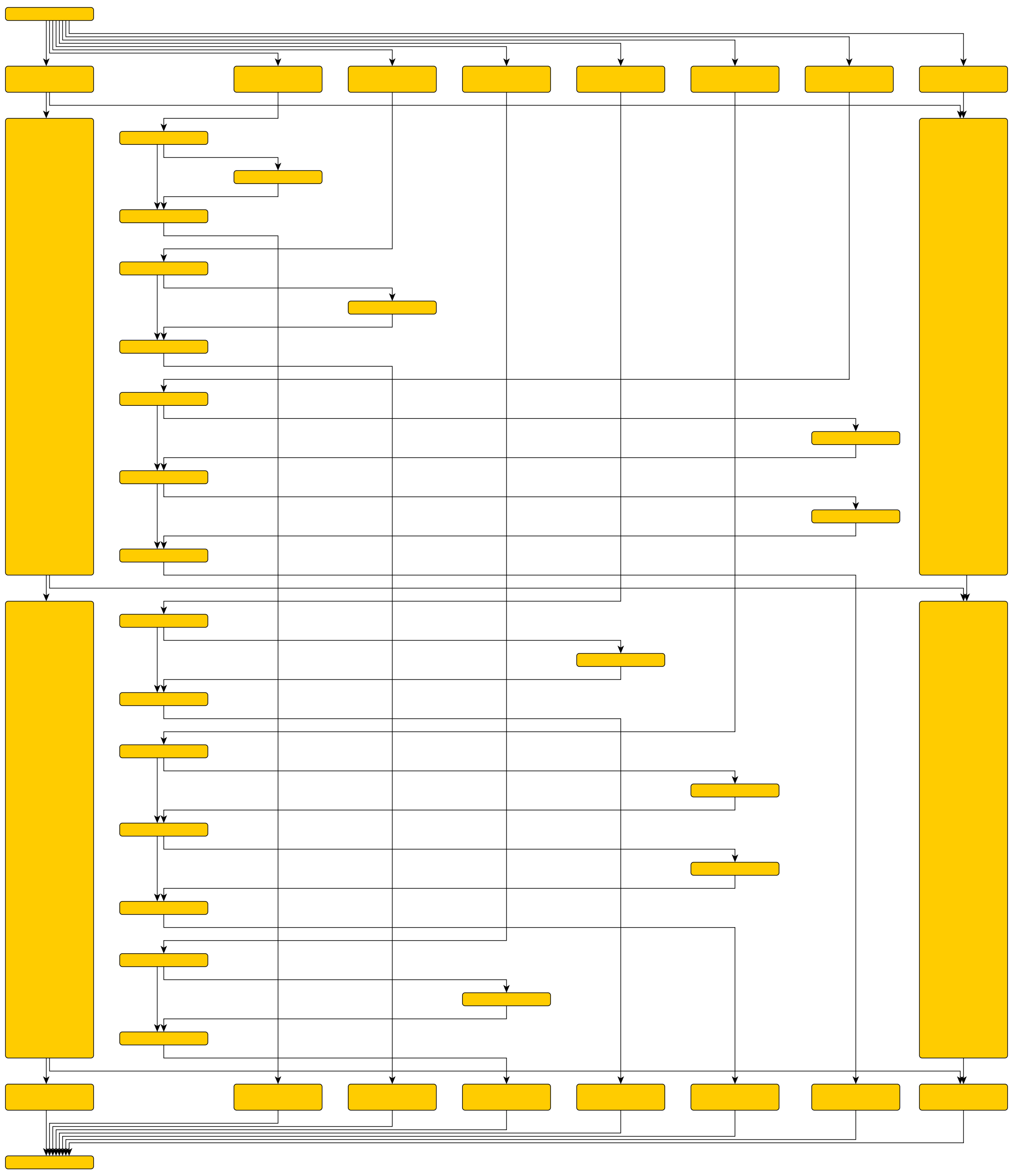


Figure 5.14: The complete VERTICAL EDGE LENGTH MINIMISATION instance for the 3-PARTITION instance $A = \{2, 2, 2, 2, 3, 3\}, B = 7$.

(ii) We already discussed, that if a number gadget is crossed by a horizontal edge segment, then two edges are crossed by this segment. Due to the spacing constraints the y-range of the crossed number gadget would increase by 20 units. Thus, the vertical edge length of the number gadget would increase by 40 units. In contrast only 20 units of vertical edge length can be reduced on the edges connecting the number gadget with its anchors. Thus, if a number gadget is crossed by a separating edge, the total vertical edge length exceeds k by at least 20 units.
Thus, in a solution of the Vertical Edge Length Minimisation instance all number gadgets are assigned to a cell of the frame. This assignment corresponds to the assignment of the values $a_i \in A$ to the sets S_i.

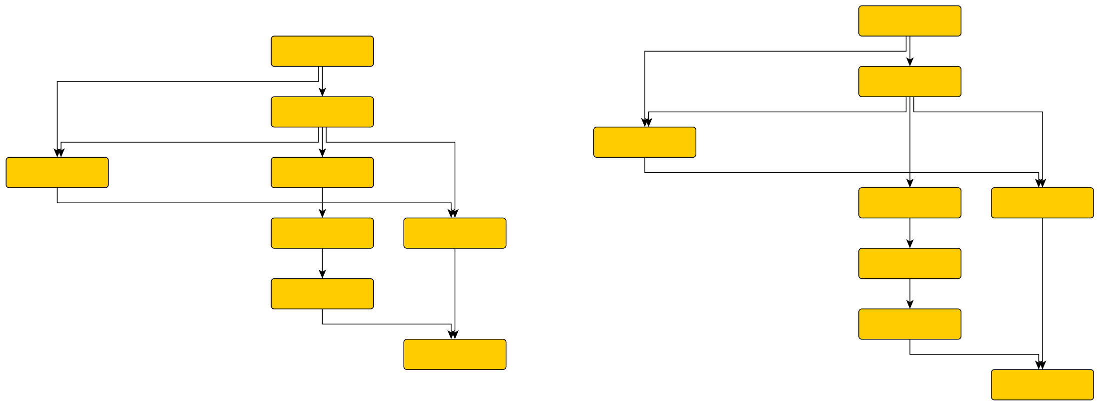
Now, we drop the assumption that in each column one anchor is above the highest separating edge and the other one is below the lowest separating edge. Thus, the anchors can move into the cells now. However, moving an anchor into a cell cannot reduce the total vertical edge length. The sum of the vertical edge length on the edge connecting the anchor with the super source or sink, respectively, and on the edge connecting the anchor with the number gadget remains unchanged. Thus, we can drop this assumption without further modification and conclude, that Vertical Edge Length Minimisation is $\mathcal{NP}$-hard. Since it is easy to see that Vertical Edge Length Minimisation $\in \mathcal{NP}$, Vertical Edge Length Minimisation is $\mathcal{NP}$-complete. In the following two sections we suggest two approaches how to cope with the problem of vertical edge length minimisation.

5.4.1.2 Network Flow Compaction

In this section we describe how a network-flow approach can be used to minimise the vertical edge length for a given column assignment. Before we describe the construction of the network we refer to one shortcoming of the network flow approach, i.e. a fixed topology is required. However, Algorithm 5 in Section 5.3 introduces some flexibility to the topology of $\widehat{G}$. In Figure 5.15 we depict two possible topologies of the same graph for the shape computed by Algorithm 5. Having these topologies fixed in both layouts the total vertical edge length is minimal. Nevertheless, in Figure 5.15a the vertical edge length is less than in Figure 5.15b. Thus, the optimum concerning the vertical edge length depends on the chosen topology.
Unfortunately, we have no efficient approach to choose the topology that leads to a layout that has minimal vertical edge length over all possible topologies. Thus, we suggest to choose one arbitrarily. Furthermore, we need to decide for each edge crossing which segment of the one edge is crossed by which segment of the other one.

Having made these decisions, we are ready to construct the flow network. We will first describe the general construction of the network and, afterwards, explain how the edge-box and edge-edge spacing constraint and alignment can be enforced. We conclude with a discussion of how edge crossings can be handled.

The general idea behind the network flow approach is that for each vertical edge segment and each vertical box boundary there exists an edge in the flow network. The length of a vertical edge segment is modelled as the volume of the flow on the corresponding edge. Furthermore, we add some edges to the flow network in order to model minimum spacing constraints. We enforce them by adding upper or lower bounds for the flow on a single edge. Furthermore, we assign costs to the edges of the flow network. Edges that correspond to vertical edge segments have cost 1, whereas all other edges are assigned with zero costs. In this network we then search for a cost-minimal flow using standard techniques. The resulting flow corresponds to a layout with minimal vertical edge length subject to the fixed topology.



(a) Vertical edge length minimal layout over all possible embeddings.

(b) Vertical edge length minimal layout for a specific embedding.

Figure 5.15: The vertical edge length minimal layout depends on the chosen embedding.

For the construction of the flow network we create one node per box and one node per face except the external face. For the external face we create two nodes of which one will serve as the source and one as the sink of the flow network. Furthermore, we add an edge for each vertical line in the layout, i.e. there is one edge for each left or right boundary of a box as well as one edge for each vertical edge segment. Edges that correspond to a box boundary connect the node corresponding to the box and the node corresponding to the face to the left or the right of the box, respectively. Edges that belong to a vertical edge segment connect the nodes that belong to the two incident faces. All edges are directed such that their source is on the left side of the corresponding vertical line and their target is on the right side. In Figure 5.16 we depict the flow network. We label the edges of the flow network using the notation “lower bound/upper bound/cost per unit”.
The boxes have a fixed height, i.e. the flow on the corresponding edges needs to be fixed as well. Thus, we set the lower and upper bound of these edges to the height of the corresponding boxes. The costs of these edges are zero as our goal function shall be independent of the box heights.
The edges that correspond to a vertical edge segment are labelled with $l/\infty/1$ where l depends on the number of edge segments and whether this one is the first, second or third segment. In general an edge of $\widehat{G}$ consists of three edge segments. However, if the column of the source or the target is the same as the column of the edge, segments can coincide. For the choice of l we distinguish the following three cases:

- **One vertical segment:** For a unique vertical segment we set l to $s_{\text{box}}^{\text{box}}$.
- **Two vertical segments:** For both vertical segments we require a lower bound of $l = s_{\text{box}}^{\text{edge}}$ for the flow on the corresponding edge.
- **Three vertical segments:** For the first and the last segment we set l to $s_{\text{box}}^{\text{edge}}$. The flow on the middle segment is not bounded as we do not enforce any spacing constraint on two segments belonging to the same edge, i.e. $l = 0$.

Using the already described flow network we only enforce the minimum spacing constraints $s_{\text{box}}^{\text{box}}$ and $s_{\text{box}}^{\text{edge}}$ as long as the box is source or target of the edge. However, we do not treat the edge-box spacing constraint, if the box is neither source nor target of the edge, and the edge-edge spacing constraint yet. Enforcing these constraints is subject of the two following paragraphs.

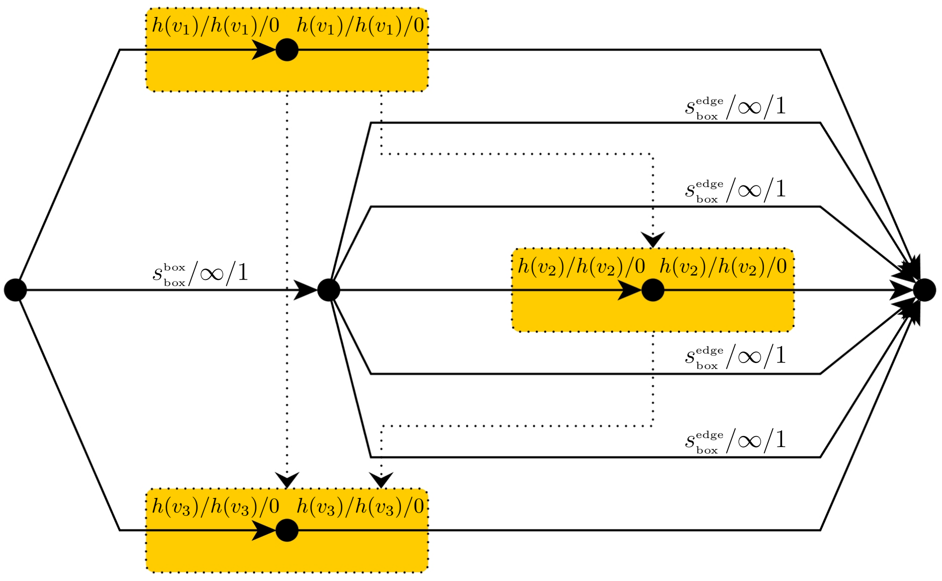

$h(v_1)/h(v_1)/0$
$h(v_1)/h(v_1)/0$
$s_{\text{box}}^{\text{edge}}/\infty/1$
$s_{\text{box}}^{\text{edge}}/\infty/1$
$s_{\text{box}}^{\text{box}}/\infty/1$
$h(v_2)/h(v_2)/0$
$h(v_2)/h(v_2)/0$
$s_{\text{box}}^{\text{edge}}/\infty/1$
$s_{\text{box}}^{\text{edge}}/\infty/1$
$h(v_3)/h(v_3)/0$
$h(v_3)/h(v_3)/0$

Figure 5.16: The general structure of the flow network for vertical edge length minimisation.

Minimum Edge-Box Spacing

We now enforce the minimum spacing constraint $s_{\text{box}}^{\text{edge}}$ between a box and an edge such that the box is neither the source nor the target of the edge. We depict such a scenario in Figure 5.17. We need to enforce a minimum distance of $s_{\text{box}}^{\text{edge}}$ between v_2 and the first as well as the second horizontal segment of edge (v_1, v_3). However, there is no vertical line in the final layout that represents these two spacing constraints (marked red in Figure 5.17). These vertical lines separate the face surrounded by v_1, v_2 and v_3 into three regions. One of them contains the node we originally added for this face. For the other two regions we create a new node in the flow network and relink the network flow edges corresponding to the vertical edge segments bordering the region to the newly added node. Furthermore, we add two edges from the original node of the face surrounded by v_1, v_2 and v_3 to the newly added nodes. These edges have $s_{\text{box}}^{\text{edge}}$ as lower bound and are assigned with zero cost, because the do not correspond to a vertical edge segment, and hence enforce the minimum spacing constraint $s_{\text{box}}^{\text{edge}}$.

Minimum Edge-Edge Spacing

In the context of vertical edge length minimisation we only need to cope with the vertical distance between two horizontal edge segments. We discuss two cases we need to treat:
The first one occurs if two edges starting at the same box bend to the same side (see Figure 5.18a). In this case we add an additional node to the face that is bounded by these two edges. Some of the edges adjacent to the old node are relinked to the new one. Furthermore, we create an edge connecting the old node with the new one. This edge has $s_{\text{edge}}^{\text{edge}}$ as lower bound. As it does not cross any vertical line in the layout the costs for the flow on this edge are zero.
The second case is that two edges belonging to the same face have overlapping x-coordinates due to the column assignment. There are several possibilities how this case can be constructed. In Figure 5.18b we depict the possibility that one edge originates in the same column as another ends. Thus, the horizontal segments overlap in one point. We can enforce the minimum edge-edge spacing using basically the same approach as in the first

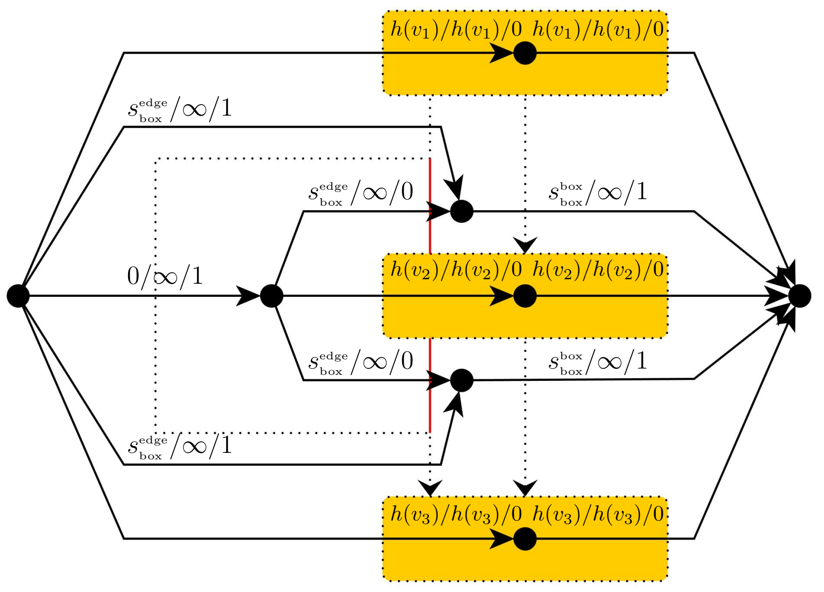

$h(v_1)/h(v_1)/0$
$h(v_1)/h(v_1)/0$
$s^{\text{edge}}_{\text{box}}/\infty/1$
$s^{\text{edge}}_{\text{box}}/\infty/0$
$s^{\text{box}}_{\text{box}}/\infty/1$
$0/\infty/1$
$h(v_2)/h(v_2)/0$
$h(v_2)/h(v_2)/0$
$s^{\text{edge}}_{\text{box}}/\infty/0$
$s^{\text{box}}_{\text{box}}/\infty/1$
$s^{\text{edge}}_{\text{box}}/\infty/1$
$h(v_3)/h(v_3)/0$
$h(v_3)/h(v_3)/0$

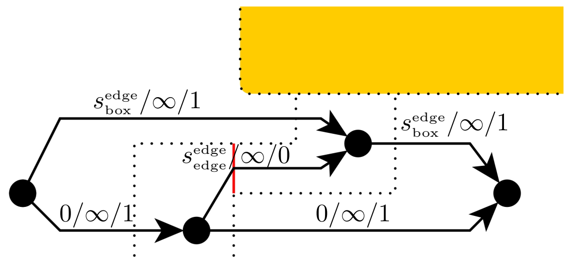
Figure 5.17: Enforcing the minimum spacing constraint $s^{\text{edge}}_{\text{box}}$. The red vertical lines represent the minimum spacing constraint.

case. However, the lower bound for the flow on the edge is $\hat{s}^{\text{edge}}_{\text{edge}}$ as the two edges have neither a common source nor a common target (see Section 3.4).

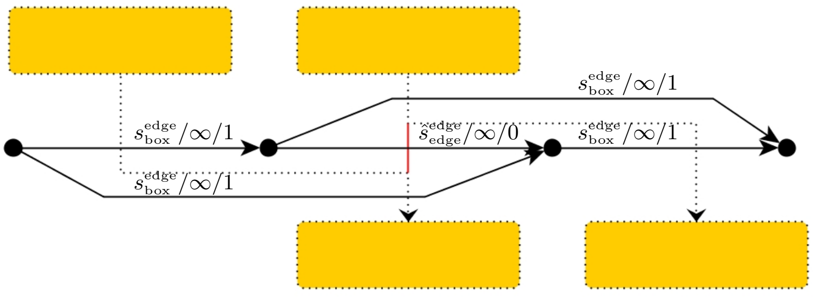
Alignment

We now turn to the alignment of predecessors, i.e. we require that the predecessors of a node are aligned at the bottom of their boxes. As already discussed in Section 3.2 this is not in all cases possible, because there can exist a directed path between two predecessors. Aligning them would conflict with the upward drawing constraint. In such a situation we are allowed to arbitrarily choose a set of predecessors such that their alignment is feasible. In the context of the network flow approach we require that there exists no directed path from an aligned predecessor to an unaligned one, whereas paths from unaligned predecessors to aligned ones are allowed. We illustrate this using the example depicted in Figure 5.19. There exist two paths between the predecessors: (v_1, v_2) and (v_1, v_3). Thus, we can align v_2 and v_3 but leave v_1 unaligned.
In terms of flow this means that the flow that passes between v_2 and v_4 has the same volume as the flow that passes between v_3 and v_4. However, there is v_1 and the edge (v_1, v_4) in between on which no alignment constraints shall be enforced. In this scenario we insert an additional node to each of the two faces and relink the edges passing between v_2 and v_4 and between v_3 and v_4, respectively, to these nodes. Furthermore, we add an edge connecting these two nodes and label it with $0/\infty/1$.

The scenario described above is easy to handle as (v_1, v_4) consists of a single edge segment. Now, assume that it has three edge segments. There are three options how the alignment of the two other boxes can be performed. We depict these options in Figure 5.20a-5.20c. We denote the red dashed line representing the alignment of the two outer nodes by *alignment line*. The point in question is which of the three segment of the edge in the middle is cut by the alignment line. We decided to use Option (b) as this is the most symmetric option but other decisions would be admissible as well.
If the edge consists of two vertical segments we need to distinguish two cases. If there is another predecessor that is assigned to the same column as the edge's source, the alignment


$s_{\text{box}}^{\text{edge}}/\infty/1$
$s_{\text{edge}}^{\text{edge}}/\infty/0$
$s_{\text{box}}^{\text{edge}}/\infty/1$
$0/\infty/1$
$0/\infty/1$

(a) Enforcing the minimum spacing constraint $s_{\text{edge}}^{\text{edge}}$.


$s_{\text{box}}^{\text{edge}}/\infty/1$
$s_{\text{box}}^{\text{edge}}/\infty/1$
$\hat{s}_{\text{edge}}^{\text{edge}}/\infty/0$
$s_{\text{box}}^{\text{edge}}/\infty/1$
$s_{\text{box}}^{\text{edge}}/\infty/1$

(b) Enforcing the minimum spacing constraint $\hat{s}_{\text{edge}}^{\text{edge}}$.

Figure 5.18: Enforcing the minimum distance between two edges. The red vertical lines represent the minimum spacing constraints.

line cuts the segment that is closer to the target. However, if all other predecessors are assigned to other columns than the edge's source, the alignment line cuts the vertical segment that is closer to the source.
In Figure 5.21a-5.21d we give an overview how to treat edges between two aligned boxes depending on the number of their vertical edge segments. Note that the scenario illustrated in Figure 5.19 corresponds to Figure 5.21a.

However, it is not clear, that this modelling does not conflict with vertical edge length minimisation. In Figure 5.21a there are two edges crossing the same vertical edge segment. The lower one is needed for the alignment, whereas the upper one serves the remaining part of the segment. In total at least $s_{\text{box}}^{\text{box}}$ flow units shall cross the edge segment in order to enforce the minimum spacing constraint between the two incident boxes. Now imagine, that the lower edge would already transport $s_{\text{box}}^{\text{box}}$ flow units. Thus, the spacing constraint between the two boxes is already fulfilled. However, the lower bound of the upper edge requires additional $s_{\text{box}}^{\text{box}}$ flow units to cross the vertical edge segment. If this lower bound is a bottleneck in the flow network, this could lead to vertical edge segments that are longer than actually necessary. The same problem arises in Figure 5.21b-5.21d.
In Lemma 5.10 we show that this is not possible. We prove that the network structure and the bounds depicted in Figure 5.21a-5.21d lead to a layout with minimal vertical edge length as long as the spacing constraints fulfil a certain inequality.

Lemma 5.10. *The constraints on the network flow depicted in Figure 5.21a-5.21d do not conflict with vertical edge length minimisation as long as the inequality* $2 \cdot s_{\text{box}}^{\text{edge}} \geq s_{\text{box}}^{\text{box}}$ *holds.*

Proof. In this proof we distinguish the three cases depicted in Figure 5.21a-5.21d and treat them one by one. We denote the node whose predecessors are aligned by t and the edge between two aligned predecessors by (s, t).

- One vertical segment: As s is not aligned with the other predecessors of t there exists a directed path p from s to another predecessor of t, which we denote by v. The

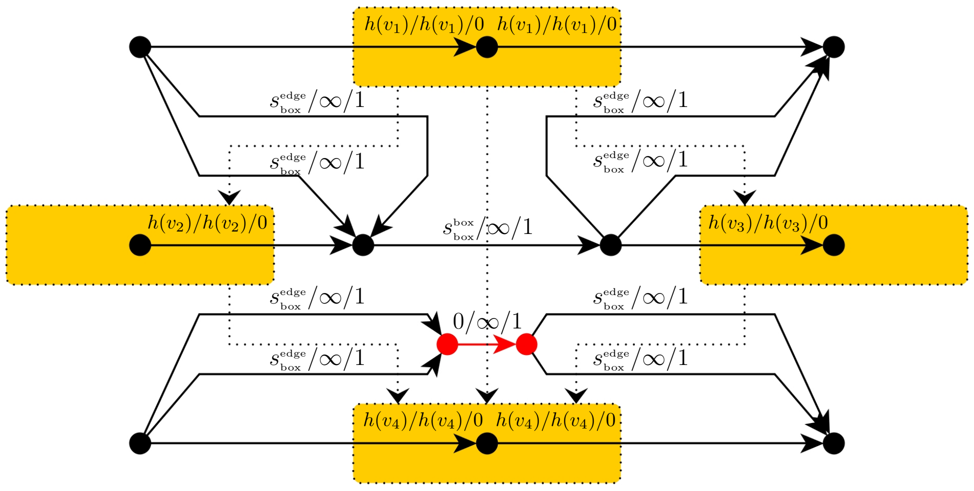
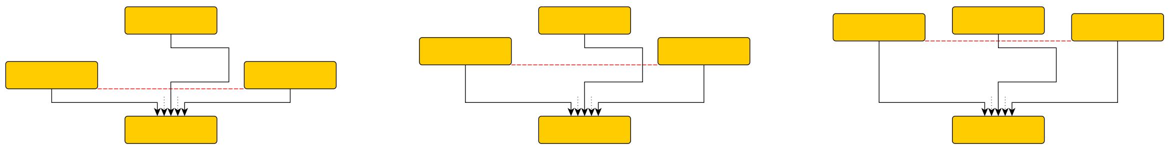


Figure 5.19: Enforcing the alignment of the predecessors. The new nodes and the new edge are marked red.

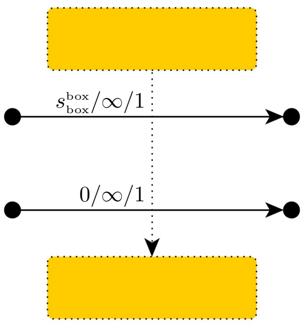


(a) Option (a). (b) Option (b). (c) Option (c).

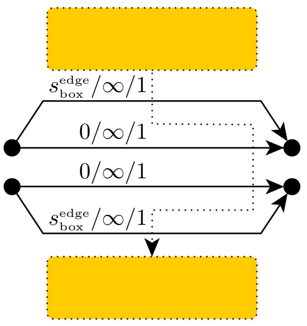
Figure 5.20: The possibilities how to align two predecessors if there is another one in-between.

path p together with (v,t) and (s,t) form a face of $\widehat{G}$. Since s and v are assigned to different columns, there is at least one edge e on p that consists of at least two vertical segments. Thus, e has at least vertical edge length $2 \cdot s_{\text{box}}^{\text{edge}}$, i.e. at least $2 \cdot s_{\text{box}}^{\text{edge}}$ flow units cross the edge segments belonging to p. As p, (v,t) and (s,t) form a face these flow units need to cross (s,t) as well. Due to the network structure these units flow on the upper edge depicted in Figure 5.21a. As $2 \cdot s_{\text{box}}^{\text{edge}} \geq s_{\text{box}}^{\text{box}}$ the lower bound $s_{\text{box}}^{\text{box}}$ on this edge is fulfilled.

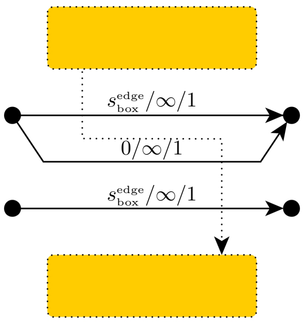
- Two vertical segments: In the case that (s,t) consists of two vertical segments we distinguish which of these segments is cut by the alignment line. By definition the alignment line cuts the segment that is closer to the target, if there is another predecessor that is assigned to the same column as s. Otherwise, the alignment line cuts the vertical segment that is closer to the source.
  - In the first case we show that the lower bound on the lowest edge crossing (s,t) in Figure 5.21c needs to be fulfilled. There are at least two predecessors of t that are aligned. Thus, at least one of them is assigned to another column than t. We denote this one by v. Since v and t are assigned to different columns, the edge (v,t) consists of at least two vertical segment. Thus, it is crossed by at least $2 \cdot s_{\text{box}}^{\text{edge}}$ flow units. These units flow on the lowest edge in Figure 5.21c as well. Hence, the lower bound $s_{\text{box}}^{\text{edge}}$ of this edge is fulfilled.
  - In the second case the argument is similar to the single vertical segment case. Assign v and p as described there. As s and v are assigned to different columns,



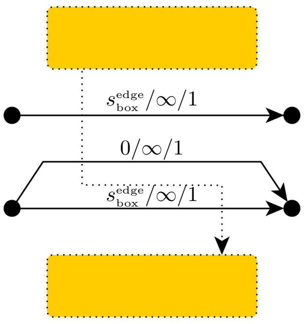
(a) The edge in-between has one vertical segment.


$s^{\text{edge}}_{\text{box}}/\infty/1$
$0/\infty/1$
$0/\infty/1$
$s^{\text{edge}}_{\text{box}}/\infty/1$

(b) The edge in-between has three vertical segments.


$s^{\text{edge}}_{\text{box}}/\infty/1$
$0/\infty/1$
$s^{\text{edge}}_{\text{box}}/\infty/1$

(c) The edge in-between has two vertical segments and the alignment line cuts the segment that is closer to the target.


$s^{\text{edge}}_{\text{box}}/\infty/1$
$0/\infty/1$
$s^{\text{edge}}_{\text{box}}/\infty/1$

(d) The edge in-between has two vertical segments and the alignment line cuts the segment that is closer to the source.

Figure 5.21: The flow network for the alignment of two predecessors if there is an edge between them. The two upper round black nodes belong to the face to the left or the right of the edge. The two lower round black nodes are inserted in order to enforce alignment.

there is at least one edge e on p that consists of at least two vertical segment. Thus, e has at least vertical edge length $2 \cdot s^{\text{edge}}_{\text{box}}$, i.e. at least $2 \cdot s^{\text{edge}}_{\text{box}}$ flow units cross the edge segments belonging to p. As p, (v, t) and (s, t) form a face these flow units need to cross (s, t) as well. Due to the network structure these units flow on the topmost edge depicted in Figure 5.21d. Hence, the lower bound $s^{\text{edge}}_{\text{box}}$ of this edge is fulfilled.

- Three vertical segments: If (s, t) consists of three vertical segments the difference introduced by alignment is that the middle segment is crossed by two network edges. As there are no constraints on these edges, i.e. both have 0 as lower bound and ∞ as upper bound, this cannot conflict with vertical edge length minimisation.

$\square$

In Figure 5.22 we demonstrate how the alignment and the minimum edge-edge spacing can be enforced at the same time. Thereby, we add multiple nodes for some faces, e.g. in the face surrounded by v_2, v_3, v_5 there are three nodes: the original one, one for the alignment and one for the edge-edge spacing constraint.

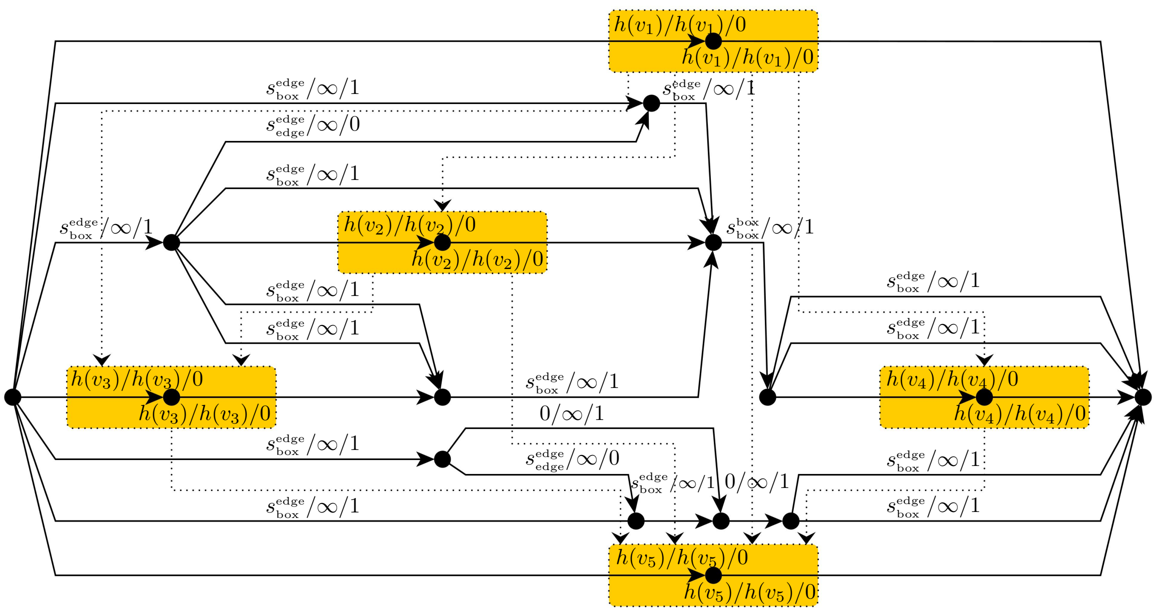


Figure 5.22: Combining the alignment of the predecessors and the edge-edge spacing constraint.

Edge Crossings

Up to now, we described the flow network construction for planar graphs. In this paragraph we discuss how to handle edge crossings. As described in the introduction of this section the topology needs to determine which edges cross. Furthermore, we need to decide which segment of the one edge is crossed by which segment of the other one. Afterwards, we need to enforce the minimum edge-edge spacing or minimum edge-box spacing. Therefore, we have one edge in the flow network above the crossing and one below it (see Figure 5.23). Depending on whether there is a bend or a box we have different lower bounds on these edges.

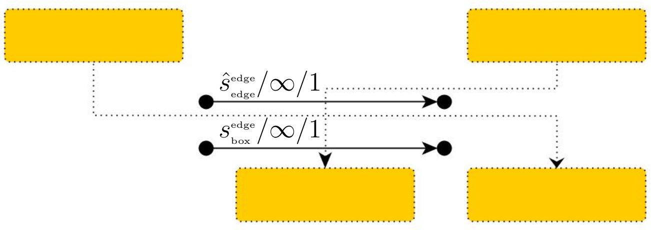

$\hat{s}_{\text{edge}}^{\text{edge}}/\infty/1$
$s_{\text{box}}^{\text{edge}}/\infty/1$

Figure 5.23: How to enforce the spacing constraints around an edge crossing.

Summary

In this short summary of the network flow approach we list the open decisions during the flow network construction. Firstly, we need to fix the topology such that it is determined which edges cross. If two edges cross, we need to decide which edge segments cross each other. Furthermore, we need to choose one of the possibilities depicted in Figure 5.20a-5.20c, i.e. how to treat an edge that is between two aligned predecessors. Having decided on these points the network flow approach computes a layout with minimal vertical edge length as long as $2 \cdot s_{\text{box}}^{\text{edge}} \geq s_{\text{box}}^{\text{box}}$.

5.4.1.3 Greedy Compaction

The network flow approach to edge length minimisation described in the foregoing section has two shortcomings. On the one hand it is quite complicated to implement and on the other hand it only computes an optimal solution if restricting to a specific topology and under certain constraints (see Section 5.4.1.2). Therefore, we decided to follow another approach that is easy to implement and returns promising results although it does not guarantee any optimality.
The idea of this approach is to find groups of nodes whose corresponding boxes need to be aligned at their bottom. These groups are treated one by one according to a reverse topological order of the groups. When positioning a group, we assign the highest possible y-coordinate to the lower side of the boxes in the group. Thereby, we assign coordinates to the source port and the first two bends of the outgoing edges and to the last two bends and the target port of the incoming edges.
In the sequel we describe the single steps of this approach in detail.

Groups of Boxes

In the first step of the greedy approach we find groups of nodes whose corresponding boxes need to be aligned at the bottom. In Figure 5.24 we illustrate that the grouping may not be unique. In this example there are two ways how two group the three nodes $1, 2$ and 3: $\{1, 2\}$ and $\{3\}$ or $\{1\}$ and $\{2, 3\}$. They cannot form one single group as there is a directed path from 3 to 1. Aligning $1, 2$ and 3 would contradict with the upward drawing constraint (see Section 3.2).

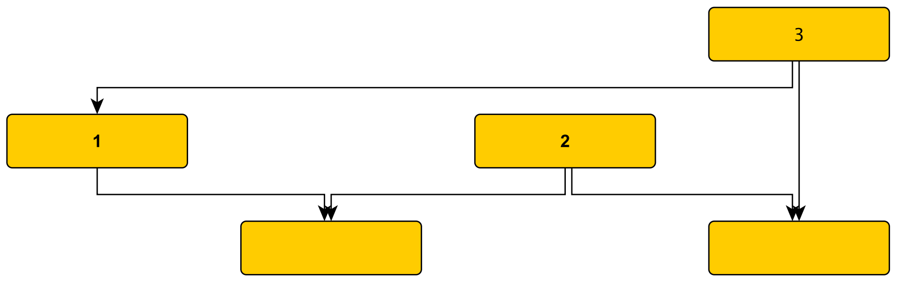


Figure 5.24: The groups are not well-defined. Either we use the groups $\{1, 2\}$ and $\{3\}$ or $\{1\}$ and $\{2, 3\}$.

We compute the grouping of the nodes using Algorithm 6. We iteratively select a node v that is not yet grouped and create a new group g_i containing v. We initialise nextSuccs with the set of successors of v. The group g_i is iteratively extended until the set nextSuccs is empty. In each iteration we check whether the elements of nextSuccs are feasible successors. A successor s is feasible if all its immediate predecessors can be added to g_i:

Definition 5.11 (Feasible Successor)**.** *Let s be the successor of a node in group g. The node s is a* feasible successor *with respect to group g if there is no directed path between an immediate predecessor p of s and a node $n \in g$.*

If a node $s \in$ nextSuccs is a feasible successor we add all its immediate predecessors to g_i. Otherwise, we simply skip s. After treating all nodes in nextSuccs we update nextSuccs such that it contains the successors of the nodes we added to g_i in the current iteration. If one of these successors has been treated in the current or an earlier iteration we do not add it again.

We now turn towards the runtime analysis of Algorithm 6. At first we need to compute which of the predecessors of a node v is an immediate predecessor and between which

Algorithm 6: Grouping

```
Input  : Graph $\widehat{G} = (\widehat{V}, \widehat{A})$
Output: Groups $g_1 \dot\cup g_2 \dot\cup \ldots \dot\cup g_k = \widehat{V}$ of the nodes in $\widehat{G}$
1  $i \leftarrow 1$
2  while $\exists n \in \widehat{V} : n \notin g_j$ for $1 \leq j < i$ do
3      Insert $v$ into $g_i$
4      nextSuccs $\leftarrow$ successors of $v$
5      while nextSuccs $\neq \emptyset$ do
6          For $s \in$ nextSuccs such that $s$ is feasible do
7              For $p$ immediate predecessor of $s$ not in a group $g_j$ with $1 \leq j \leq i$ do
8                  Insert $p$ into $g_i$
9          nextSuccs $\leftarrow$ successors of nodes added to $g_i$ in this iteration that have not
           been treated yet
10     $i \leftarrow i + 1$
```

pair of nodes a directed path exists. This can easily be done in $\mathcal{O}(|\widehat{V}| \cdot (|\widehat{V}| + |\widehat{A}|))$ time: We start depth-first-searches at each $n \in \widehat{V}$. If we reach a successor s of v only once, then v is an immediate predecessor of s. Furthermore, we thereby compute all pairs of node between which a directed path exists.

Now, we analyse the runtime of Algorithm 6 itself assuming that the information about immediate predecessors and directed paths is given. We amortise the cost of the feasibility checks in Line 6 over all iterations and then analyse the runtime of the remaining algorithm: To check whether a successor s is feasible we need to check whether there is a directed path from an immediate predecessor of s to a node in g_i or vice versa. Let $\#p(s)$ be the number of immediate predecessors of s. Checking s upon feasibility needs $2 \cdot |g_i| \cdot \#p(s)$ look-ups whether there exists a directed path. For each i a node s is checked at most once but it can be checked with respect to different groups g_i. In order to amortise the costs of Line 6 we need to consider the following sum:

$$\begin{aligned}\sum_{i=1}^{k} \sum_{s \in \widehat{V}} 2 \cdot |g_i| \cdot \#p(s) &= 2 \cdot \sum_{i=1}^{k} |g_i| \cdot \sum_{s \in \widehat{V}} \#p(s) \\ &= 2 \cdot |\widehat{V}| \cdot \sum_{s \in \widehat{V}} \#p(s) \\ &\leq 2 \cdot |\widehat{V}| \cdot |\widehat{A}|\end{aligned}$$

Thus, the amortised costs of Line 6 are $\mathcal{O}(|\widehat{V}| \cdot |\widehat{A}|)$.

The remaining part of the algorithm runs in $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ time. The summand $|\widehat{V}|$ stems from the insertion of the nodes in $\widehat{V}$ into a group g_i in Line 3 and 8, whereas the treatment of predecessors or successors (Line 4, 7 and 9) is assigned to the arcs leading to the summand $|\widehat{A}|$.

In total Algorithm 6 runs in $\mathcal{O}(|\widehat{V}| \cdot (|\widehat{V}| + |\widehat{A}|) + |\widehat{V}| \cdot |\widehat{A}| + |\widehat{V}| + |\widehat{A}|) = \mathcal{O}(|\widehat{V}| \cdot (|\widehat{V}| + |\widehat{A}|))$ time. In order to store the information about the directed paths, we need $\mathcal{O}(|\widehat{V}|^2)$ space.

Order of Groups

In the second step we compute an order of the groups. We will later position the groups one by one according to the computed order. Since we draw the layout bottom-up in the third step, when positioning a group g_i all successors of nodes in g_i need to be positioned already. Thus, the order we compute needs to be a reverse topological order. This step can be done in $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ time and space.

Positioning the Groups

In the last step of the greedy approach the actual coordinate assignment takes place. We treat the groups according to the order computed in the second step. The y-coordinates of the ports and the bends of an edge are computed in two parts. When the source of an edge is positioned, we compute the y-coordinates of the source port and the first two bends. The first bend is in the source's column, whereas the other one is in the column assigned to the edge. The two last bends and the target port are computed, when the target of it is positioned.

At first, we compute the highest possible y-coordinate $\hat{y}$ for the lower boundary of the nodes in g_i. We compute the highest possible y-coordinate $\hat{y}(v_j)$ for each node $v_j \in g_i$ separately and then take the minimum of these values as $\hat{y}$.
When computing $\hat{y}(v_j)$ we need to consider that we have not routed the second part of the edges whose sources are in g_i yet. For each of these edges we compute the highest possible y-coordinate by going over all columns between the edge's column and the source's column. For each column we consider the highest already drawn edge or node and add the appropriate minimum spacing. We take the minimum of all these values which is the highest possible y-coordinate for the horizontal edge segment. Basing upon this value we can compute an upper bound for the y-coordinate of v_j.
After computing the bounds induced by the horizontal segments of the incident outgoing edges of v_j, we need to check the box-box spacing to another box in the same column as v_j.
After computing $\hat{v_j}$ for each $v_j \in \widehat{V}$, we set $\hat{y}$ to the minimum of these values. Then, we position the boxes in g_i such that their lower boundary is aligned at $\hat{y}$ and draw the second part of the outgoing edges. Thereby, we draw the horizontal segments as high as possible. Afterwards, we draw the incoming edges as low as possible. Therefore, we process the nodes $v_j \in g_i$ in an arbitrary ordering and draw their incoming edges such that all spacing constraints hold.

The runtime of this step is $\mathcal{O}(|\widehat{V}| + |\widehat{A}| \cdot b)$ where b is the width of the shape measured in the number of columns. Each node in $\widehat{V}$ is considered only once, i.e. when it is positioned. The edges are treated twice: Once when the target is positioned and its first part is drawn and a second time when its source is positioned, i.e. the second part of the edge is drawn. Both times the highest possible y-coordinate needs to be determined which requires as much operations as the drawn horizontal segment spans columns. We simply bound this number of columns by b.

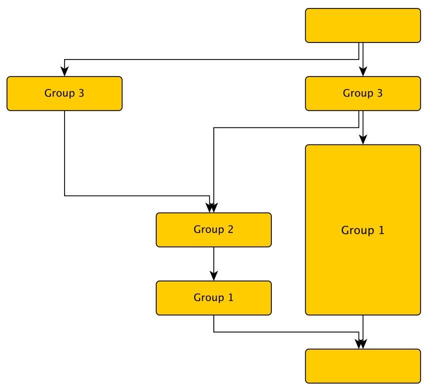
Runtime and Space Analysis

The computation of the groups in step one takes $\mathcal{O}(|\widehat{V}|\cdot(|\widehat{V}|+|\widehat{A}|))$ time, whereas the groups can be order in $\mathcal{O}(|\widehat{V}|+|\widehat{A}|)$. For the final y-coordinate assignment we need $\mathcal{O}(|\widehat{V}|+|\widehat{A}|\cdot b)$ time. Thus, the greedy approach runs in $\mathcal{O}((|\widehat{V}| + b) \cdot (|\widehat{V}| + |\widehat{A}|))$ time in total. As $\widehat{G}$ is connected and, therefore, $\mathcal{O}(|\widehat{V}|) \subseteq \mathcal{O}(|\widehat{A}|)$, we can simplify the runtime expression to $\mathcal{O}((|\widehat{V}| + b) \cdot |\widehat{A}|)$.
For the computation of the groups $\mathcal{O}(|\widehat{V}|^2)$ space is required. Furthermore, we need $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ space for storing the y-coordinates assigned to the boxes and the edge ports and bends. Additionally, we store for each of the b columns the highest object that is already drawn. Thus, in total the greedy compaction needs $\mathcal{O}(|\widehat{V}|^2 + |\widehat{A}| + b) = \mathcal{O}(|\widehat{V}|^2)$ space.

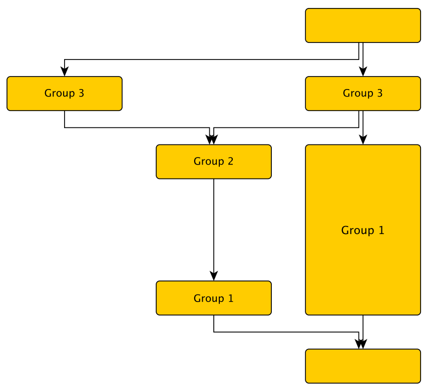
Comparison to the Network Flow Compaction

A shortcoming of the greedy approach in comparison to the network flow approach is that it actually only tries to minimise the height of the layout instead of the vertical edge

length itself. We illustrate this in Figure 5.25. The high box in Group 1 forces Group 3 to be positioned quite high. Thus, Group 2 can be positioned in several ways. The greedy approach positions Group 2 as low as possible. However, Group 2 has two incoming edges and one outgoing edge, i.e. by moving it upwards the vertical edge length can be minimised. Of course, this could be optimised by a post-processing step. However, for the sake of simplicity we pass on without post-processing.
Nevertheless, there is a big advantage over the network flow approach: We do not fix the topology beforehand. The open decisions concerning the edge crossings are made one by one when the edges are drawn. The information gathered by drawing a partial layout can help to reduce the vertical edge length in comparison to fixing an arbitrary topology before constructing the flow network.



(a) The layout resulting from the greedy approach.



(b) The optimal layout with respect to vertical edge length minimisation.

Figure 5.25: A shortcoming of the greedy approach.

5.4.2 Horizontal Edge Length Minimisation

We follow two approaches in order minimise horizontal edge length. On the one hand we try to avoid that an edge contains a horizontal segment that is directed to the right as well as one that is directed to the left in Section 5.4.2.1. On the other hand we minimise the width of the final layout in Section 5.4.2.2.

5.4.2.1 Bow Reduction

The column assignment computed by Algorithm 5 can contain so called *bows*. By bow we denote an edge that contains a horizontal segment that is directed rightwards as well as one that is directed leftwards. Thus, the column assigned to the bow is not between its source's and its target's column. In Figure 5.26 we depict a layout that has two bows. These are the edges $(2, 4)$ and $(1, 6)$. We distinguish *necessary* and *unnecessary* bows. A bow is *unnecessary* iff the enclosed rectangle contains no box (see Figure 5.26). Thus, $(2, 4)$ is an unnecessary bow, whereas $(1, 6)$ is necessary. An unnecessary bow can be removed by assigning the source's or target's column to the edge itself. Due to the removal of a bow, another bow can become compactable. In Figure 5.26 the removal of the bow $(2, 4)$, makes the bow $(1, 6)$ compactable by one column. In this section we only discuss the removal of unnecessary bows. For the compaction of bows we can apply the techniques presented in the subsequent section.

In a first step we iterate over all edges of $\widehat{G}$ and check which of them is a bow. As this condition can be checked in constant time, we can detect the bows in $\mathcal{O}(|\widehat{A}|)$ time. Afterwards, we treat them one by one and check whether they can be removed. A bow is anchored in three different columns, i.e. the source's column, the target's column and

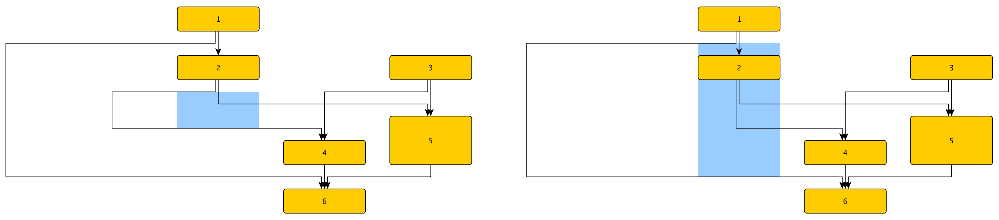


Figure 5.26: The bow $(2, 4)$ can be removed, whereas the bow $(1, 6)$ is necessary, because the blue region (right side) contains Box 2. However, the removal of bow $(2, 4)$ makes bow $(1, 6)$ compactable by one column.

the column assigned to the edge itself. By *middle column* we refer to the column in the middle. Note that due to the definition of a bow, the middle column is either the source's or the target's column. When removing a bow, we move the middle vertical segment from the edge's column to the middle column. Thus, there may be no other box in this column that would be intersected by the moved vertical segment. This can be checked in $\mathcal{O}(|\widehat{V}|)$ time. Summing up over all treated bows, we need $\mathcal{O}(\#\text{bows} \cdot |\widehat{V}|)$ operations in order to check which bows are unnecessary and to remove them. Thus, bow reduction in total needs $\mathcal{O}(|\widehat{A}| + \#\text{bows} \cdot |\widehat{V}|)$ time and $\mathcal{O}(1)$ space.

5.4.2.2 Width Compaction

After compacting the height in Section 5.4.1 we discuss width compaction in this section. Due to the shifting in each iteration of Algorithm 5 we introduce new columns to the shape. Some of them are necessary while others will be partly or even completely empty in the final layout. Thus, the final layout is wider than actually necessary. In Figure 5.27a we depict a layout that can be compacted by two columns. Our approach to width compaction is column based, i.e. the resulting layout again consists of disjoint columns.
In this section we first discuss the theoretic preliminaries of width compaction before we come to an algorithm which performs the width compaction.

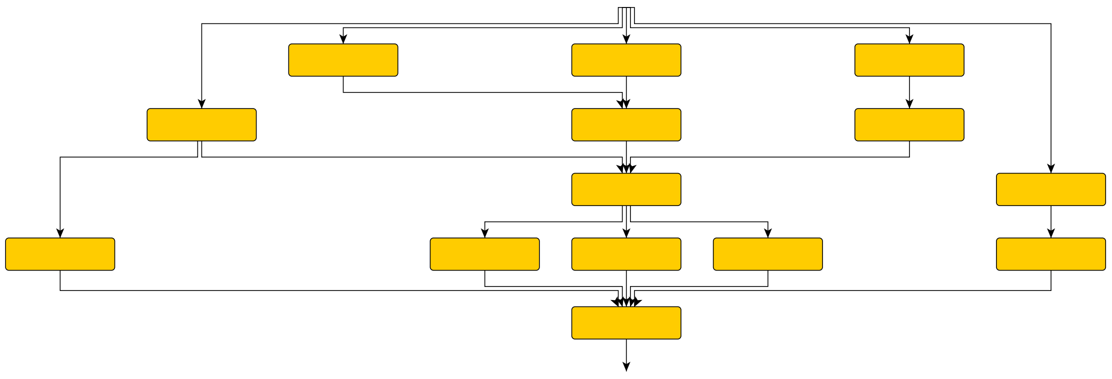
Preliminaries

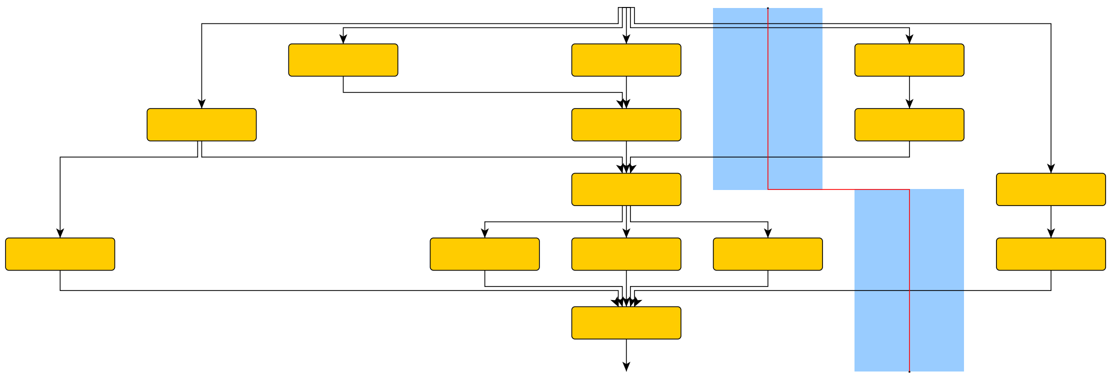
We perform width compaction along so-called *compaction paths*. Each compaction along a *valid* compaction path will reduce the number of columns by one. We firstly define the term compaction path and describe how to compact along such a path. Afterwards, we can discuss validity of a compaction path.

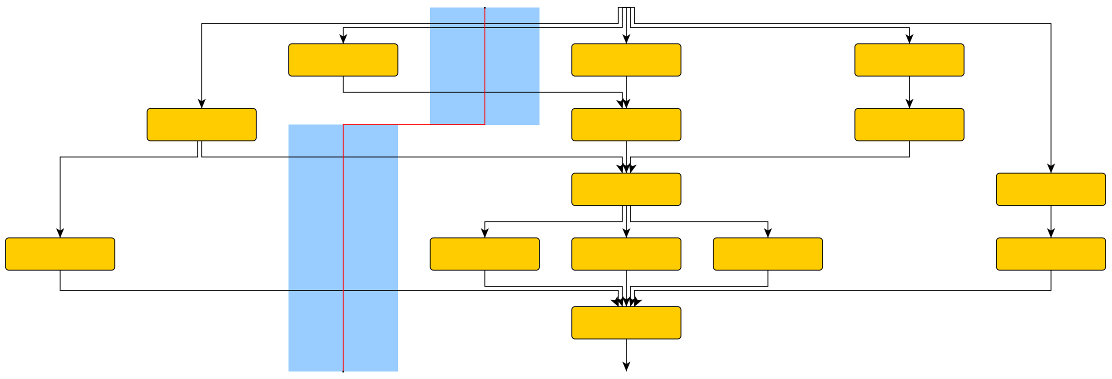
Definition 5.12 (Compaction Path)**.** *A* compaction path *is a axis-parallel y-monotone path passing through a layout from the top to the bottom. It may only cut horizontal edge segments. Furthermore, its vertical segments need to run within columns such that no vertical edge segment is allowed to run through the same part of these columns.*

In Figure 5.27b-5.27d we depict three compaction paths for the layout shown in Figure 5.27a. Of course, many more compaction paths are possible.

In Figure 5.28 we illustrate how compaction along a compaction path is performed. Firstly, we split the path into its vertical segments. Then we delete the y-range of a column that contains a vertical path segment. Thus, in total we gain one column of free space. We move


(a) A layout that can be compacted by two columns.


(b) Compaction Path 1.


(c) Compaction Path 2.

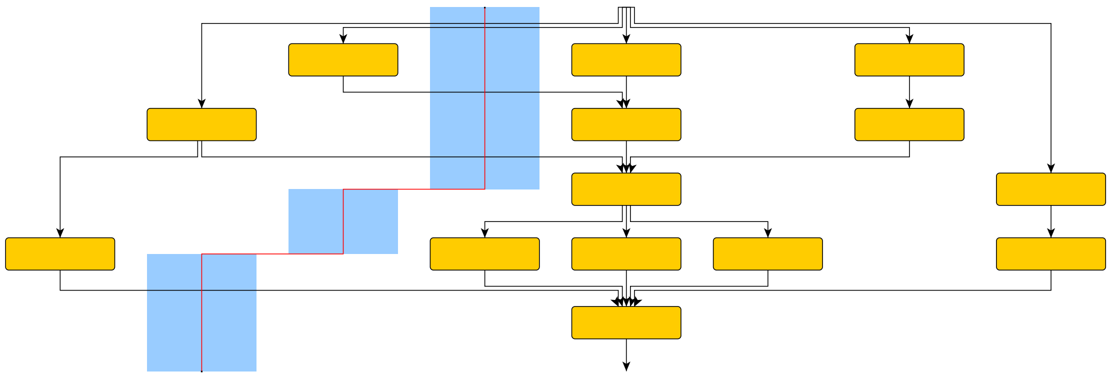
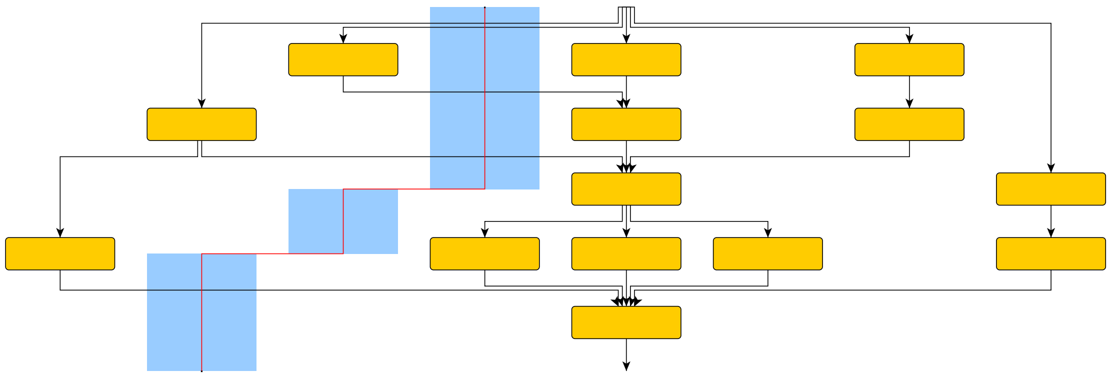
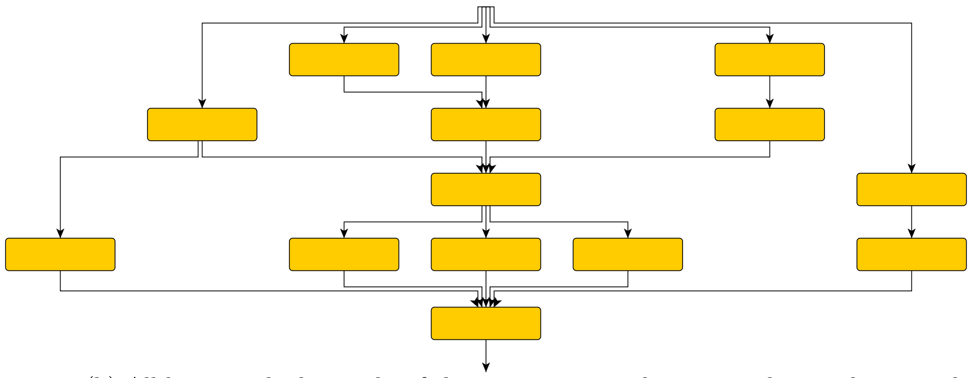

(d) Compaction Path 3.

Figure 5.27: An uncompacted layout and some valid compaction paths.



(a) The compaction path along which we compact.

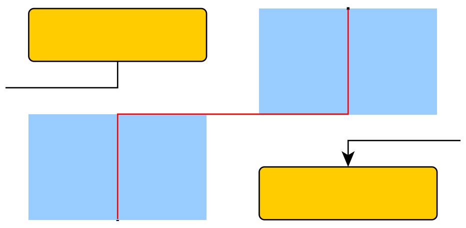


(b) All boxes and edges right of the compaction path are moved one column to the left.

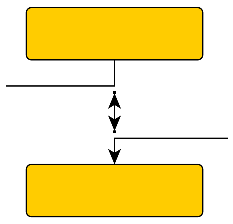
Figure 5.28: How to perform compaction along a compaction path.

everything that has been right to compaction path one column to the left and, thereby, fill the free space again. Thus, in total the number of columns decreased by one.

Note that compacting along a compaction path can lead to an infeasible layout, because spacing constraints could be violated (see Figure 5.29). Therefore, we distinguish between *valid* and *invalid* compaction paths.



(a) The compaction path along which we compact.



(b) After the compaction the spacing constraint between the two edges can be violated.

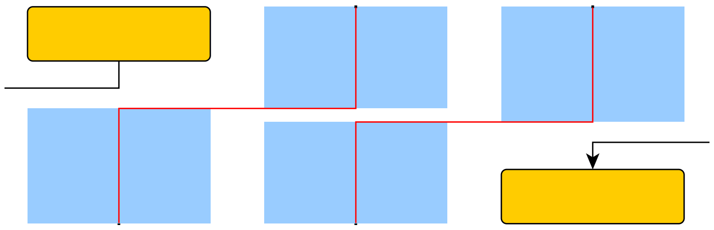
Figure 5.29: An invalid compaction path.

Definition 5.13 (Valid Compaction Path)**.** *A* valid *compaction path p is a compaction path such that compacting the layout along p results in a valid layout.*

The compaction paths depicted in Figure 5.27b-5.27d are all valid. However, we cannot compact the layout along all of them together. Therefore, we expand the term validity from a single compaction path to a set of compaction paths:

Definition 5.14 (Valid Set Of Compaction Paths)**.** *A* valid *set of compaction paths is a set of compaction paths such that the contained paths do not overlap each other. However, they are allowed to cross each other. Furthermore, compacting the layout along these paths needs to result in a valid layout.*

In the example of Figure 5.27 two possible valid sets of compaction paths are {Compaction Path 1, Compaction Path 2} and {Compaction Path 1, Compaction Path 3}. However, Compaction Path 2 and Compaction Path 3 together cannot be in a valid set of compaction paths as they overlap each other.
Note that a set of valid compaction paths is not necessarily a valid set of compaction paths. In Figure 5.30a we depict two valid compaction paths. However, compacting along both paths can violate the spacing constraint between the two edges (see Figure 5.30b).


(a) The set of compaction paths along which we compact. Both compaction paths are valid.

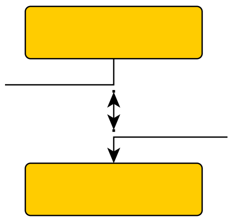

(b) After the compaction the spacing constraint between the two edges can be violated.

Figure 5.30: An invalid set of compaction paths.

The algorithm we present in the following paragraph finds the compaction paths using a right-first search. Therefore, we introduce the terms *rightmost* valid compaction path and *rightmost* valid set of compaction paths. Afterwards, we argue that a cardinality-maximal valid set of compaction paths can be transformed to a rightmost valid set of compaction paths of the same cardinality.

Definition 5.15 (Rightmost Valid Compaction Path)**.** *A* rightmost *valid compaction path is a valid compaction path $p = (p_1, \dots, p_k)$ fulfilling the following property: No subpath $(p_i, \dots, p_j)$ of p can be replaced by a path $(q_1, \dots, q_l)$ that is right of p such that $(p_1, \dots, p_{i-1}, q_1, \dots, q_l, p_{j+1}, \dots, p_k)$ is a valid compaction path.*

Definition 5.16 (Rightmost Valid Set Of Compaction Paths)**.** *A* rightmost *valid set of compaction paths $\mathcal{P} = \{p_1, \dots, p_k\}$ is a valid set of compaction paths such that no pair of paths p_i, p_j for $i \neq j$ cross each other and p_1 is a rightmost valid compaction path. Furthermore, we require for $i = 1, \dots, k-1$ that p_{i+1} is left of p_i and no subpath of p_{i+1} can be replaced by a path in-between p_i and p_{i+1} such that p_{i+1} stays a valid compaction path.*

In Figure 5.31 we depict a rightmost valid set of compaction paths for the example of Figure 5.27. It remains to show that we can transform any cardinality-maximal valid set of compaction paths $\mathcal{P} = \{p_1, \dots, p_k\}$ to a rightmost valid set of compaction paths of the same cardinality. We do this by the following normalisation steps:

- **Remove crossings**: Firstly, we remove all crossings of paths. Let p_i and p_j for $i \neq j$ cross each other. We simply swap the suffixes of the paths that start at the crossing. Afterwards, p_i and p_j touch each other but do not cross anymore. Furthermore, we cannot introduce a new crossing by this approach. Thus, the overall number of crossings decreases by one in each step.

- **Order the paths**: Since the paths $p_1, \dots, p_k$ do not cross, we order them from right to left.
- **Move rightwards**: After removing all crossings we treat the paths p_i in the increasing order of their indices. As long as we can perform a replacement of a subpath of p_i as forbidden in Definition 5.15 and 5.16, we perform this replacement.

After these normalisation steps we gain a rightmost valid set of compaction paths. Note that we do not change the number of paths during the normalisation. Thus, the right-first approach we present in the following paragraph is feasible.

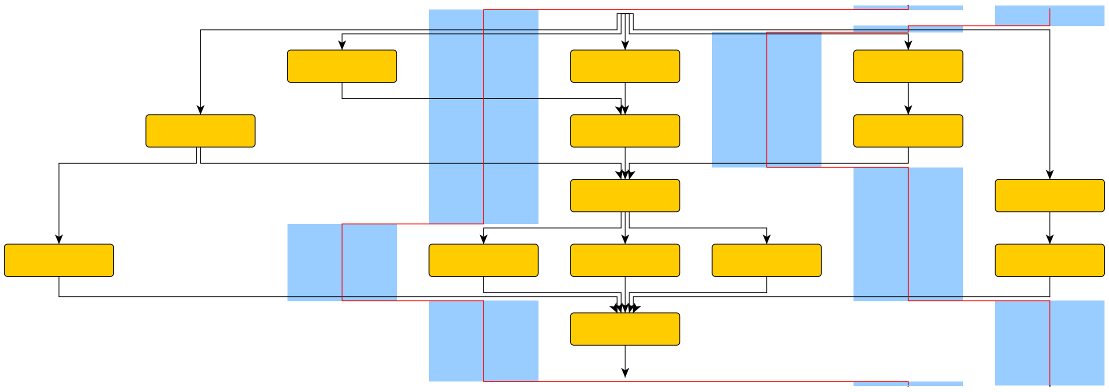


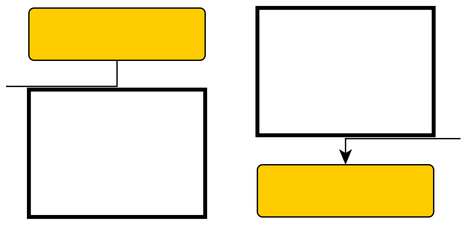
Figure 5.31: A rightmost valid set of compaction paths.

The Algorithm

In this paragraph we present the algorithm that performs the width compaction. As already mentioned in the foregoing paragraph, this algorithm finds compaction paths by using a right-first search. Initially, we build up the compaction network in which we search for compaction paths using the right-first search. After we found a compaction path, we need to slightly modify the compaction network in order to ensure that the computed set of compaction paths is valid.

At first, we describe the initialisation of the compaction network $\mathcal{D}$. The compaction network depends on empty regions in the columns, i.e. we need to find the empty rectangles in each column. Therefore, we create a list per column which will contain the boxes and edge segments that are contained by the corresponding column. Afterwards, we iterate over all boxes and edges. We add the boxes to the list of the column to which they are assigned. The edges are treated more fine-grained. All three vertical segments are added to the list of the corresponding column. Furthermore, we add the horizontal segments to all columns between the source's column and the edge's column and the edge's column and the target's column, respectively. Thus, in total we have $\mathcal{O}(|\widehat{V}| + |\widehat{A}| \cdot b)$ entries in the lists. Since $\widehat{G}$ is connected and, therefore, $\mathcal{O}(|\widehat{V}|) \subseteq \mathcal{O}(|\widehat{A}|)$, we can simplify the number of entries to $\mathcal{O}(|\widehat{A}| \cdot b)$. Each of these entries is associated with two y-coordinates that determine the y-range that is occupied by the box or edge segment, respectively. We sort these entries according to their y-coordinates in $\mathcal{O}(|\widehat{A}| \cdot b \cdot \log(|\widehat{A}| \cdot b))$ time.
Afterwards, we detect the free rectangles in the columns, i.e. the gap between two entries in the list whose y-ranges do not touch. Furthermore, we assume that on the top and on the bottom of each column is a large free rectangle. For each free rectangle we add a node to $\mathcal{D}$. Since there is a bijective mapping between nodes and free rectangles, we do not distinguish between them.
Since we have at most $\mathcal{O}(|\widehat{A}| \cdot b)$ entries that bound the free rectangles, there can be at most $\mathcal{O}(|\widehat{A}| \cdot b)$ rectangles, i.e. at most $\mathcal{O}(|\widehat{A}| \cdot b)$ nodes in $\mathcal{D}$.

We create the edges of $\mathcal{D}$ in two ways: For two rectangles in the same column we add an edge between them, if they are only separated by a single horizontal edge segment.
Furthermore, we create edges between rectangles in neighbouring columns if their y-ranges overlap. Imagine two overlapping rectangles of neighbouring columns as depicted in Figure 5.32a. If a compaction path p runs through these rectangles as shown in Figure 5.29a, compacting along p would move the right box under the left one. Then the distance between the two edges needs to be at lest $s_{\text{edge}}^{\text{edge}}$ or $\hat{s}_{\text{edge}}^{\text{edge}}$, respectively. Note that this distance after the compaction equals to the overlapping of the y-ranges of the two rectangles. Therefore, we only add an edge from the right to the left rectangle, if the overlapping of the y-ranges of the two rectangles is at least the minimum spacing $s_{\text{edge}}^{\text{edge}}$ or $\hat{s}_{\text{edge}}^{\text{edge}}$, respectively. However, we always add an edge from the left to the right rectangle. If compacting along a path that runs from left to right, the two edges do not move to the same column and we cannot violate a spacing constraint. Analogously, we add edges in the mirrored case (see Figure 5.32b).
As final step of the construction of $\mathcal{D}$, we add two nodes to $\mathcal{D}$ which we denote by s and t. We connect s to all rectangles on the top of the columns and connect all rectangles on the bottom of the columns with t.
Obviously, $\mathcal{D}$ is a planar graph, i.e. it has at most $\mathcal{O}(|\widehat{A}| \cdot b)$ edges. However, we need to check more pairs of rectangles than we add edges. Using Algorithm 7 the number of checks that do not lead to an edge is in $\mathcal{O}(|\widehat{A}| \cdot b)$ as well. Thus, the construction of $\mathcal{D}$ can be performed in $\mathcal{O}(|\widehat{A}| \cdot b)$ time. In Figure 5.33a we depict a clipping of a layout and the corresponding compaction network $\mathcal{D}$.


(a) Case (a).

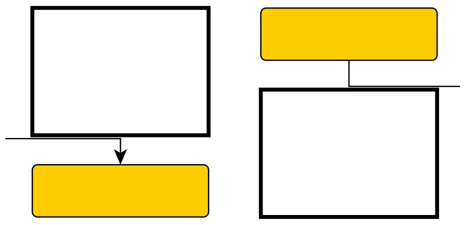

(b) Case (b).

Figure 5.32: Edges between overlapping boxes of neighbouring columns.

Algorithm 7: Edges Between Columns

```
   Input   : Graph $\mathcal{D}$, free rectangles $r_{i,j}$ in columns $c_i$
1  For $i$ from 1 to $b-1$ do
2      $j_1 \leftarrow 0$
3      repeat
4          $j_2 \leftarrow 0$
5          repeat
6              If $r_{i,j_1}$ and $r_{i+1,j_2}$ overlap
7                  Add edges to $\mathcal{D}$
8              Else If $r_{i+1,j_2}$ is below $r_{i,j_1}$
9                  break
10             $j_2 \leftarrow j_2 + 1$
11         until all rectangles in column $c_{i+1}$ treated
12         $j_1 \leftarrow j_1 + 1$
13     until all rectangles in column $c_i$ treated
```

We use the compaction network $\mathcal{D}$ in order to find compaction paths using a right-first depth-first search starting at s. Thereby, we need to ensure that the computed compaction path is y-monotone. Therefore, we always keep the current y-coordinate in mind. We are only allowed to move to a neighbouring rectangle if this requires no upward movement of the compaction path. If the search reaches t, we found a compaction path.

After a compaction path has been found, we need to slightly modify the compaction network. The part of the compaction network that is right of the compaction path cannot be part of a further compaction path, because we use a right-first search. Therefore, we do not need to consider this part. However, we need to treat the part through which the compaction path cuts (see Figure 5.33): We split rectangles that contain (a part of) a vertical segment of the compaction path such that there is one rectangle that is cut from top to bottom and one or two rectangles that are not cut by the compaction path. Rectangles that are crossed horizontally are split into two rectangles as well.
For all rectangles that are cut from top to bottom by a vertical path segment we delete the corresponding nodes in $\mathcal{D}$. Afterwards, we need to create edges between the newly created rectangles on the left side of the compaction path and their surrounding rectangles. In Figure 5.33a-5.33c we illustrate the modification of $\mathcal{D}$ after the compaction along a compaction path. After the modification of $\mathcal{D}$ we continue the right-first search in order to find more compaction paths. The set of compaction paths we compute is (i) rightmost, because we use a right-first search and (ii) valid due to the construction of $\mathcal{D}$ and its modifications.

Runtime and Space Analysis

We now turn to the runtime and space analysis of finding the compaction paths. We already argued that $\mathcal{D}$ has at most $\mathcal{O}(|\widehat{A}| \cdot b)$ nodes and it is planar. However, the size of $\mathcal{D}$ can increase due to the modifications after the compaction along a path. Recall that we are dealing with rightmost compaction paths. Thus, each horizontal segment of a compaction path coincides with the upper or lower boundary of a rectangle. Otherwise, we can replace a subpath of the compaction path with a path that is right of it, i.e. the compaction path was not rightmost yet. We now estimate the number of rectangles we need to create at most.
Assume we would split all rectangles at the y-coordinates of the upper and lower boundary of a single rectangle. Thereby, we can create at most two new rectangles per column, i.e. $2 \cdot b$ rectangles in total. Doing this with all $\mathcal{O}(|\widehat{A}| \cdot b)$ rectangles, we have $\mathcal{O}(|\widehat{A}| \cdot b^2)$ rectangles. Thus, the number of nodes in $\mathcal{D}$ cannot exceed $\mathcal{O}(|\widehat{A}| \cdot b^2)$. As $\mathcal{D}$ is still planar, the number of edges has the same complexity. In order to analyse the time complexity of the right-first depth-first search that operates on a changing graph, we assume that it is directly executed on the final $\mathcal{D}$. Thus, it runs in $\mathcal{O}(|\widehat{A}| \cdot b^2)$.
When all compaction paths are found, we need to compact along them. Compacting along one path can easily be performed in $\mathcal{O}(|\widehat{V}| + |\widehat{A}| \cdot b) = \mathcal{O}(|\widehat{A}| \cdot b)$. Thus, compacting along all paths needs $\mathcal{O}(|\widehat{A}| \cdot b \cdot (b - \hat{b}))$ time, where $\hat{b}$ is the number of columns in the final layout.

Summing up the runtime of the initial construction of $\mathcal{D}$, the computation of the compaction paths and the compaction itself, we get $\mathcal{O}(|\widehat{A}| \cdot b \cdot (\log(|\widehat{A}| \cdot b) + b + b - \hat{b})) = \mathcal{O}(|\widehat{A}| \cdot b \cdot (\log(|\widehat{A}|) + b))$ as total runtime. As argument maps are simple graphs, $|\widehat{A}| \leq |\widehat{V}|^2$ holds. Thus, we can further simplify the time complexity to $\mathcal{O}(|\widehat{A}| \cdot b \cdot (\log(|\widehat{V}|) + b))$.
The required space is in $\mathcal{O}(|\widehat{A}| \cdot b^2)$, because the number of nodes and edges in $\mathcal{D}$ can grow to this number. Furthermore, we need $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ space to store the column assignment. Thus, the size of $\mathcal{D}$ is the greatest summand and the space complexity of the width compaction is $\mathcal{O}(|\widehat{A}| \cdot b^2)$.

In our implementation we use a slightly modified version of the algorithm we just described. In order to avoid the modification of the compaction network after the compaction along a compaction path, we simply recompute the compaction network from scratch. Note that this is a greater effort than modifying the compaction network. Nevertheless, it has only little impact on the empiric runtime of width compaction.

5.5 Summary

Before evaluating the layouts computed by the topology-shape-metrics approach we presented in the foregoing sections, we give a short summary of it in this section. Furthermore, we present the time and space complexity of the single steps in Table 5.3.

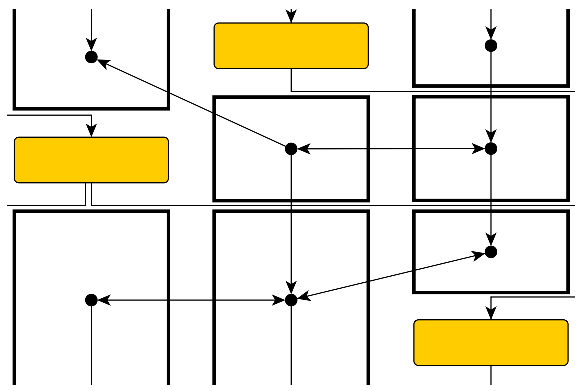
In Section 5.1 we showed how to transform the input graph G into an acyclic graph $\widehat{G}$ with a single source and single sink.
Afterwards, we compute an upward planar representation by using layer-free upward crossing minimisation technique by Chimani et al. [CGMW10] as described in Section 5.2. To this end, we first compute a feasible upward planar subgraph and route the remaining edges through this subgraph. In doing so, we try to insert the edges cost minimal. If a cost minimal insertion would forbid the insertion of remaining edges, we use a heuristic approach to insert the edge instead. For the computation of the upward planar representation we use randomisation when computing the feasible upward planar subgraph as well as for the reinsertion of the remaining edges.
In the subsequent step we assign the nodes and edges to columns (see Section 5.3) such that each edge is bent at most four times. Thereby, we relax some decisions that have been made during the computation of the upward planar representation, i.e. the computed column assignment can lead to other crossings than the upward planar representation prescribes.
In Section 5.4 we compute the final coordinates of boxes and edges. Firstly, we compact the height using a greedy approach that positions groups of nodes that need to be aligned one by one. Afterwards, we remove unnecessary bows and compact the width.
In the last step we reverse the edges that were reversed in Section 5.1 in order to remove cycles.

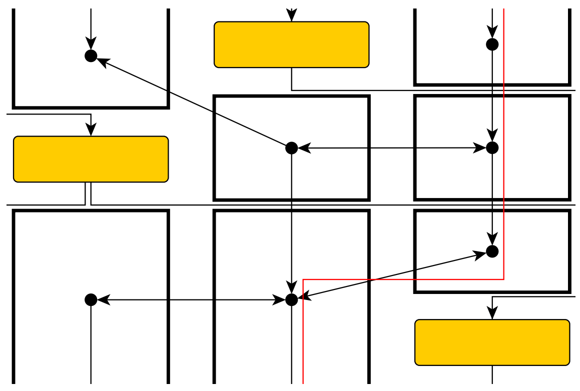
Topology	Time Complexity	Space Complexity
Feasible upward planar subgraph	$\mathcal{O}(\lvert\widehat{A}\rvert^2)$	$\mathcal{O}(\lvert\widehat{A}\rvert)$
Optimal edge insertion	$\mathcal{O}(\lvert\widehat{V}\rvert + r)$	$\mathcal{O}(\lvert\widehat{V}\rvert + \#\text{crossings})$
Heuristic edge insertion	$\mathcal{O}(\lvert\widehat{V}\rvert^2 + r\lvert\widehat{V}\rvert)$	$\mathcal{O}(\lvert\widehat{V}\rvert + \#\text{crossings})$
$\sum$ (only optimal edge insertions)	$\mathcal{O}(\lvert\widehat{A}\rvert^2)$	$\mathcal{O}(\lvert\widehat{A}\rvert + \#\text{crossings})$
$\sum$ (with heuristic edge insertions)	$\mathcal{O}(\lvert\widehat{A}\rvert^2 \cdot \lvert\widehat{V}\rvert)$	$\mathcal{O}(\lvert\widehat{A}\rvert + \#\text{crossings})$
Shape		
4-bend shape with top. mod.	$\mathcal{O}(\lvert\widehat{A}\rvert)$	$\mathcal{O}(\lvert\widehat{A}\rvert)$
Metrics		
Height compaction	$\mathcal{O}((\lvert\widehat{V}\rvert + b) \cdot \lvert\widehat{A}\rvert)$	$\mathcal{O}(\lvert\widehat{V}\rvert^2)$
Bow reduction	$\mathcal{O}(\lvert\widehat{A}\rvert + \#\text{bows} \cdot \lvert\widehat{V}\rvert)$	$\mathcal{O}(1)$
Width compaction	$\mathcal{O}(\lvert\widehat{A}\rvert \cdot b \cdot (\log(\lvert\widehat{V}\rvert) + b))$	$\mathcal{O}(\lvert\widehat{A}\rvert \cdot b^2)$
$\sum$	$\mathcal{O}(\lvert\widehat{A}\rvert \cdot b \cdot (\lvert\widehat{V}\rvert + b))$	$\mathcal{O}(\lvert\widehat{A}\rvert \cdot b^2)$
Total		
Only optimal edge insertion	$\mathcal{O}(\lvert\widehat{A}\rvert^2 \cdot b)$	$\mathcal{O}(\lvert\widehat{V}\rvert^2 + \lvert\widehat{A}\rvert \cdot b^2 + \#\text{crossings})$
With heuristic edge insertion	$\mathcal{O}(\lvert\widehat{A}\rvert^2 \cdot (\lvert\widehat{V}\rvert + b))$	$\mathcal{O}(\lvert\widehat{V}\rvert^2 + \lvert\widehat{A}\rvert \cdot b^2 + \#\text{crossings})$

Table 5.3: An overview over the runtime of the single steps.

In Table 5.3 we give an overview of the running times of the single steps we just described. The variable r stands for the number of edges that we need to insert after the current

edge. By b we denote the number of columns in the column assignment of the shape. Note that $r, b, |\widehat{V}| \leq |\widehat{A}|$ holds.
Furthermore, we present two overall running times of the whole topology-shape-metrics approach. One time we assume, that during the computation of the topology all edges are inserted optimally, and get the overall runtime $\mathcal{O}(|\widehat{A}|^2 \cdot b)$. The other time we assume, that there are heuristic edge insertions as well, which leads to an overall running time of $\mathcal{O}(|\widehat{A}|^2 \cdot (|\widehat{V}| + b))$. The space complexity is $\mathcal{O}(|\widehat{V}|^2 + |\widehat{A}| \cdot b^2 + \#\text{crossings})$ in both cases.


(a) The compaction network $\mathcal{D}$.


(b) A compaction path p.

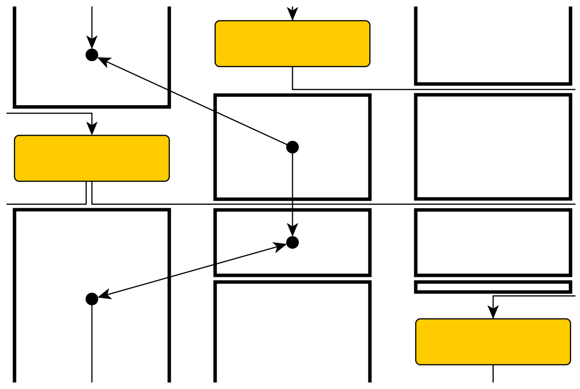

(c) The modified network after the compaction along p.

Figure 5.33: The compaction network $\mathcal{D}$ and how it is modified.

6. Evaluation

In this chapter we evaluate the layouts computed by our topology-shape-metrics approach which we described in the foregoing chapter. For the computation of the topology we set the parameters α and δ to one, i.e. we equally weight crossing minimisation and total source/sink distance. Furthermore, we require the following minimum spacings: $s_{\text{box}}^{\text{box}} = 40$, $s_{\text{box}}^{\text{edge}} = 20$, $s_{\text{edge}}^{\text{edge}} = 5$ and $\hat{s}_{\text{edge}}^{\text{edge}} = 20$. As already mentioned in Section 5.2 we randomise the computation of the topology. We randomise it at two points:
(i) For the construction of the feasible upward planar subgraph we randomise the order in which we visit the outgoing edges during the depth-first searches. Furthermore, we randomise the order of insertion of edges that are not in the initial feasible upward planar subgraph. Each time we compute a feasible upward planar subgraph, we use ten randomised runs and choose the subgraph having at most edges as result.
(ii) Afterwards, we reinsert the missing edges in a random order.
These two steps are performed ten times, i.e. we execute the computation of a feasible upward planar subgraph $10 \cdot 10 = 100$ times in total. Over all runs we take the crossing minimal upward planar representation as result.

In Section 6.1 we discuss the aesthetic qualities of these layouts, whereas we present the hard facts, e.g. the number of crossings, in Section 6.2.

6.1 Case Studies

In this section we present six layouts that are created by the algorithms described in Chapter 5. Using these layouts as example we discuss the benefits and the disadvantages of these algorithms. Figure 6.1 and Figure 6.2 show two layouts for which we have no suggestions for improvement. Both layouts are free of crossings, their numbers of bends are close to the minimum, they are compact and locally symmetric. Beside the hard facts they are well structured and give a good impression of the flow of argumentation. The other four layouts – though they are good looking in general – have minor drawbacks which we discuss in the following.

In Figure 6.3 and Figure 6.4 there is a set of boxes on the right side of the layout that are positioned quite high. It seems as one could move these boxes downwards. However, these boxes are aligned with boxes that are some columns apart, i.e. they need to be positioned that high. One could reconsider the drawing style of argument maps such that alignment of boxes is only required if the number of columns between the boxes is small. Note that

we can easily integrate such a modification by adapting the computation of the groups in Algorithm 6 in Section 5.4.1.3.
However, Figure 6.3 gives a good example of the edge bundling enabled by the different minimum edge-edge spacings $s_{\text{edge}}^{\text{edge}}$ and $\hat{s}_{\text{edge}}^{\text{edge}}$. At the rightmost sink there are two edges having only $s_{\text{edge}}^{\text{edge}}$ distance, because they have a common sink. These edges are bundled. However, above the second rightmost sink we can see two edges that are not bundled. Since the two edges have neither a common source nor sink, the distance between their horizontal segments is at least $\hat{s}_{\text{edge}}^{\text{edge}}$.

In Figure 6.5 on the left side there is a constellation that is similar to a bow which we treated in Section 5.4.2.1. However, this time we have a bow containing a box. We could expand the concept described in Section 5.4.2.1 to paths in $\widehat{G}$ that contain only nodes whose in-degree and out-degree equals to one. However, the scenario depicted in Figure 6.5 only occurs 5 times in the 51 input instances. Therefore, we omit its treatment.

The layout shown in Figure 6.6 consists of many columns but is quite flat. However, the number of columns is bounded by the number of sources and the number of sinks of G as we require free sources and sinks (see Section 3.2). Again, one could reconsider the drawing style of argument maps, because especially for large trees layouts get badly proportioned.

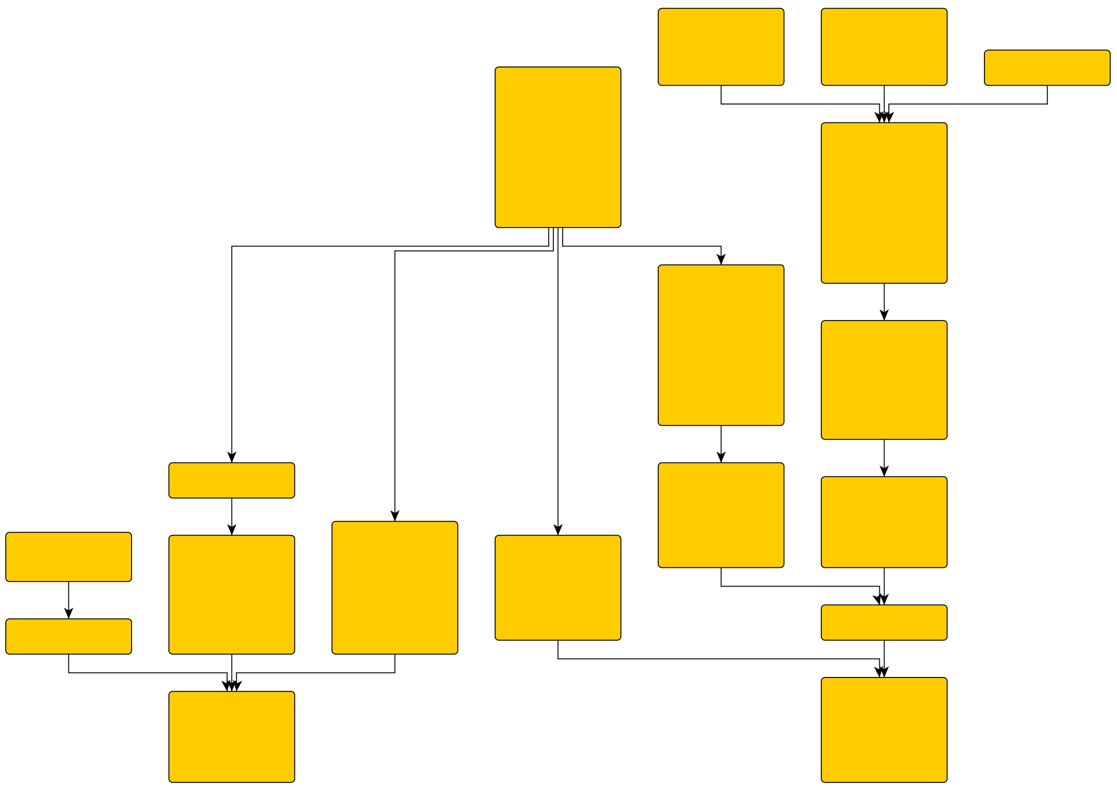


Figure 6.1: Instance D-2.5.

6.2 Statistics

After giving an overview on the aesthetic qualities of the layouts computed by our topology-shape-metrics approach, we discuss the hard facts, e.g. the number of crossings, the number of bends and the runtime, in this section. Our analysis is based on the 51 argument maps supplied by the three sources presented in Section 2.2.

We start analysing the number of crossings in the final layout as well as in the upward planar representation $\mathcal{U}$ (see Figure 6.7). Although it is theoretically possible that the

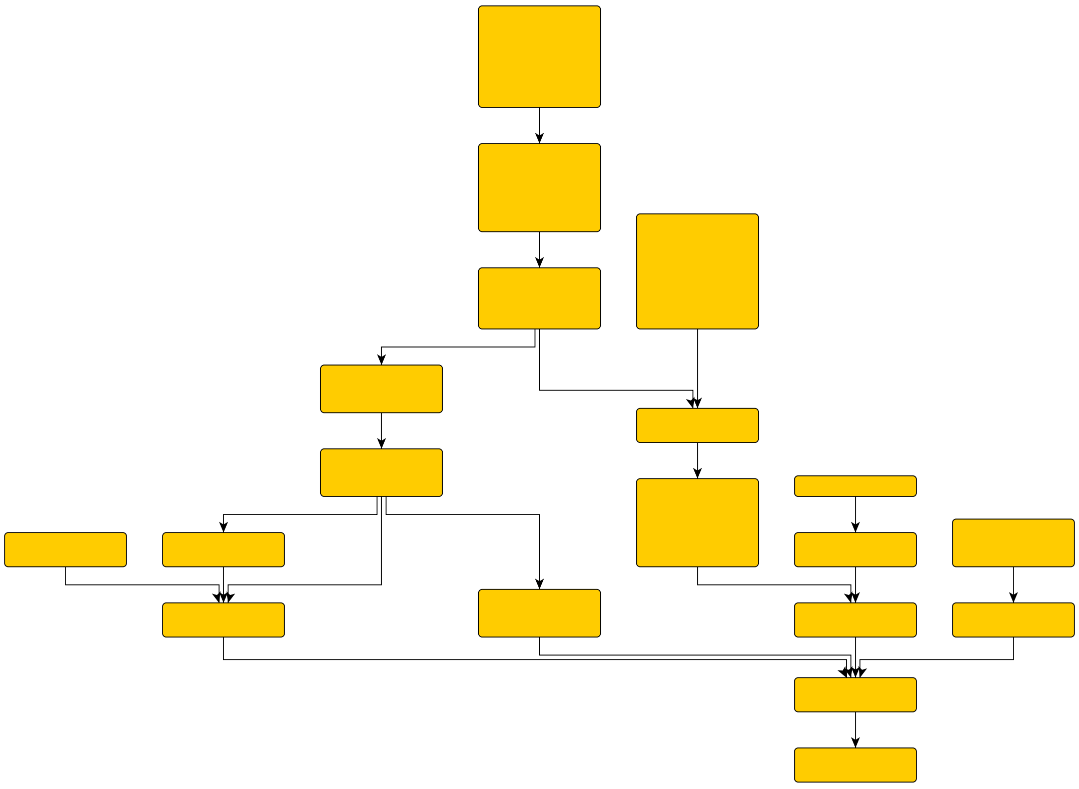


Figure 6.2: Instance D-3.5.

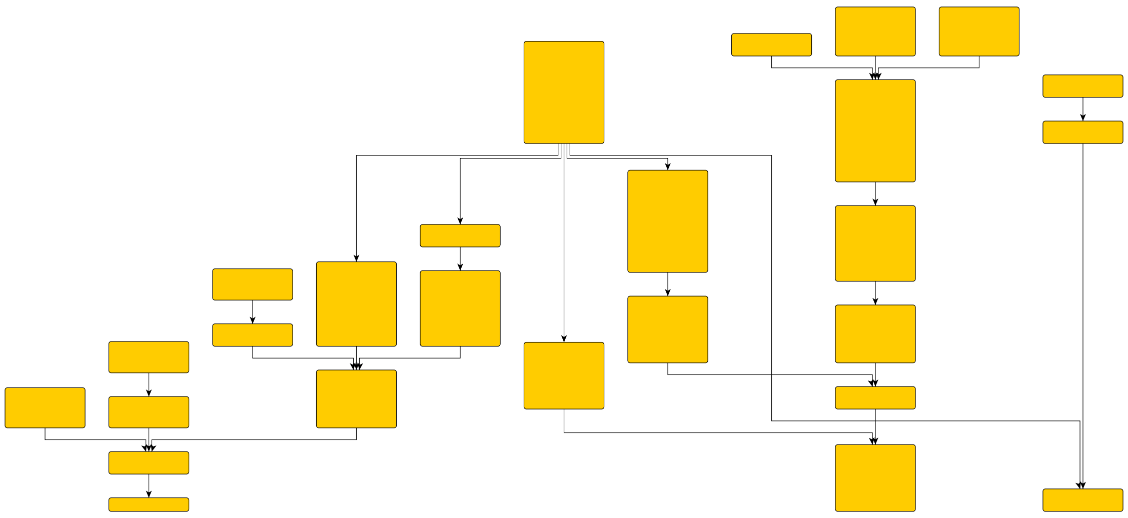


Figure 6.3: Instance D-2.7.

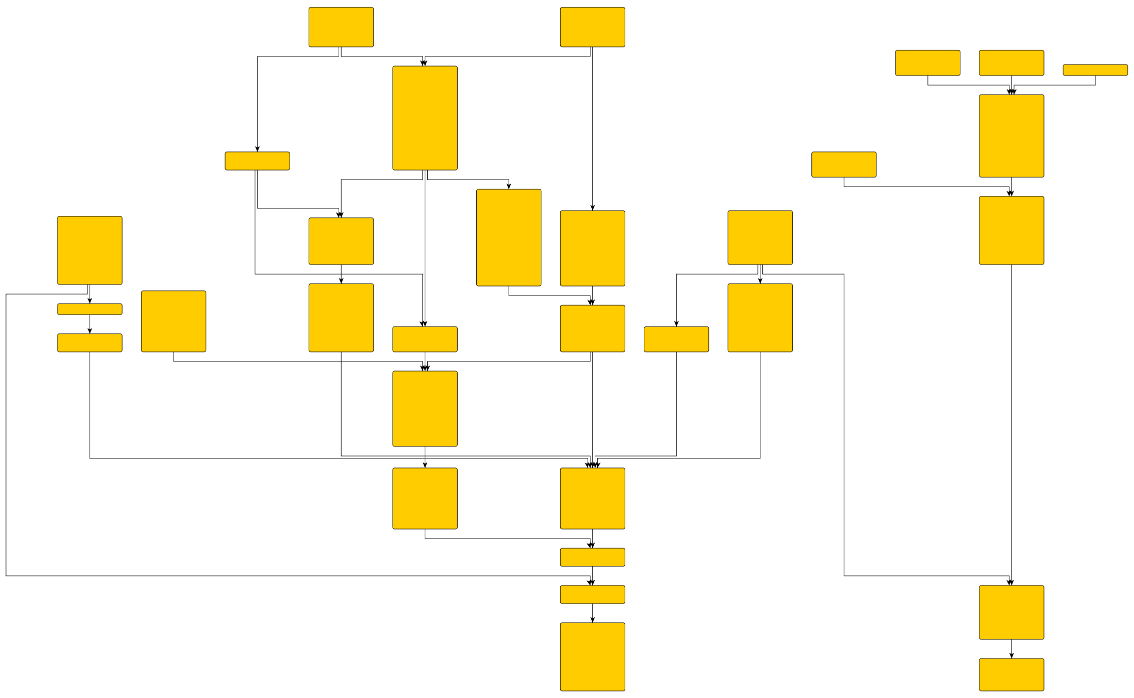


Figure 6.4: Instance D-4.2.

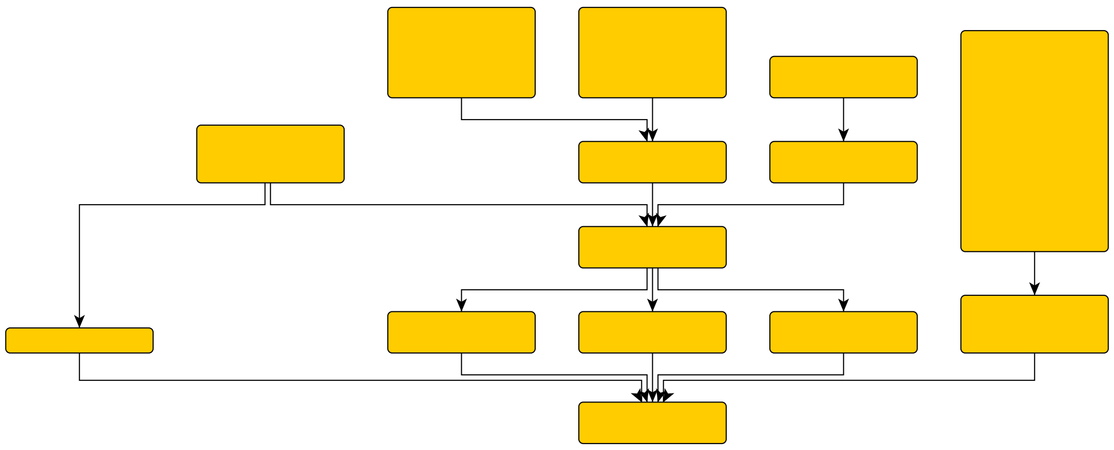


Figure 6.5: Instance D-5.3.

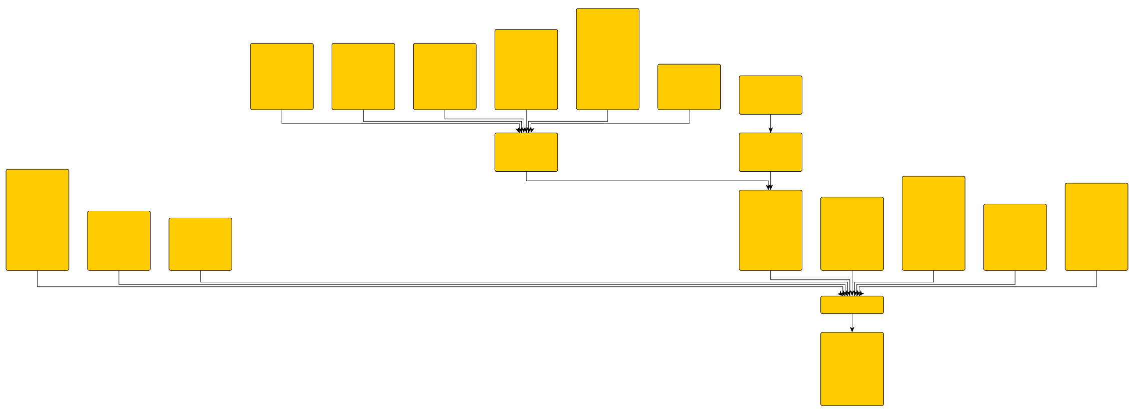


Figure 6.6: Instance CE-1.2.

final layout contains less crossings than $\mathcal{U}$, for all instances there are at least as many crossing in the final layout as in the upward planar representation $\mathcal{U}$. However, if $\mathcal{U}$ is free of crossings, then the final layout is planar as well (see Lemma 5.9). On average there are 69% more crossings in the final layouts than in the upward planar representations. Breaking down the number of crossings to the number of edges, we have 0.14 crossings per edge.
Furthermore, it is notable that for the vast majority of argument maps the computed layout is planar. There are thirteen non-planar layouts but only five instances that are not upward-planar. Thus, one could reduce the number of crossings for eight instances. Note that for these eight instances the number of crossings in $\mathcal{U}$ is non-zero as well. One approach in order to reduce the number of crossings is to increase the number of randomised runs during the computation of the topology. This increases the probability that the computed topology is optimal with respect to the number of crossings and the total source/sink distance. However, this will increase the runtime of this step as well.
In Figure 6.8 we depict the distribution of the number of crossings on an edge for all instances. Note that in contrast to Figure 6.7 we now count each crossing twice, i.e. one time for the one edge and one time for the other edge. The filled diamond indicates the mean of the number of crossings on an edge. In most cases it is very close to zero. However, there are four instances having around one crossing on each edge. The maximum number of crossings on an edge is ten. Nevertheless, most non-planar layouts have at most two crossings on an edge.

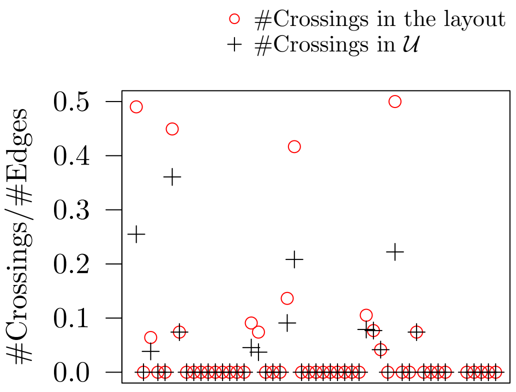


Figure 6.7: Number of crossings divided by the number of edges.

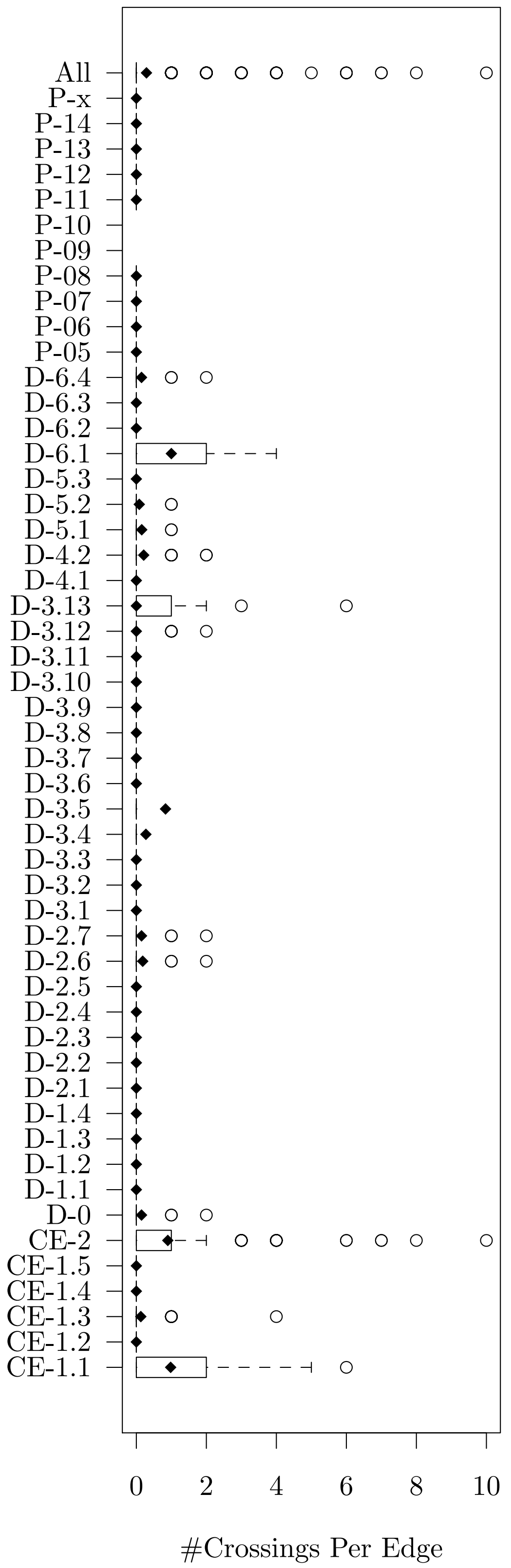


Figure 6.8: Distribution of the number of crossings on an edge.

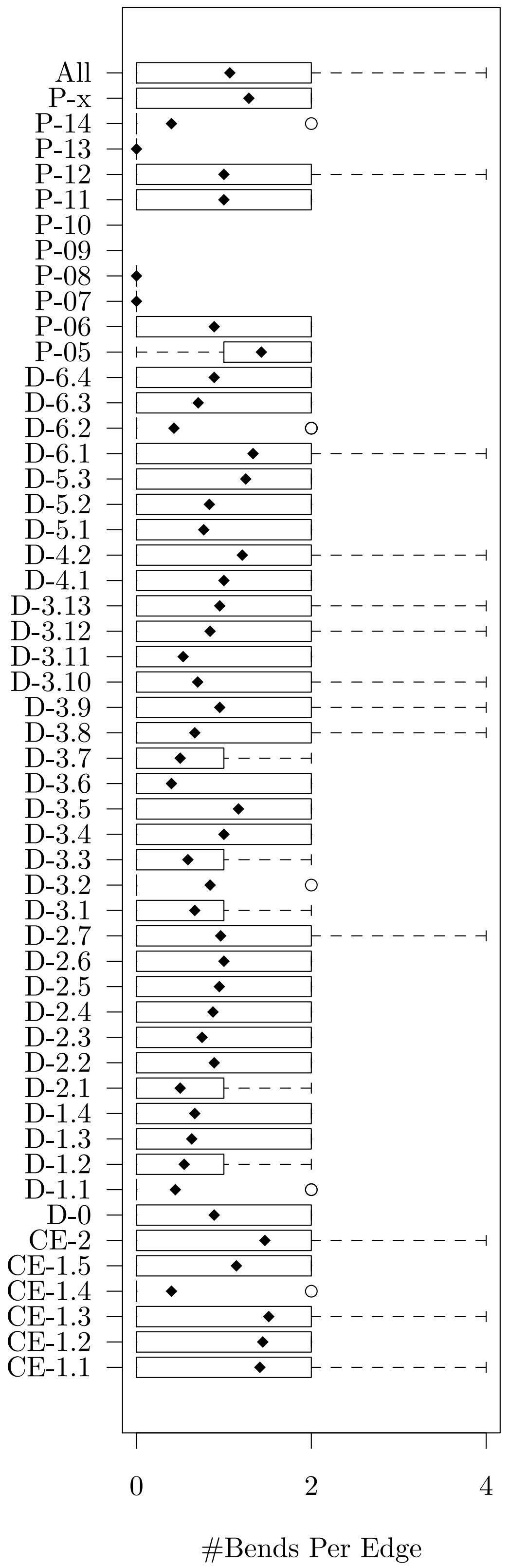


Figure 6.9: Distribution of the number of bends on an edge.

Now, we turn towards the bends. Again, we plot the distribution of the number of bends on an edge broken down to the instances (see Figure 6.9). Due to Algorithm 5 there can be at most four bends per edge. Except of three bend-free layouts, the average number of bends per edge lies between 0.40 and 1.52. Only 13 of 51 layouts have edges that have four bends. Taking the average over all layouts, we have 1.07 bends per edge.

The total source/sink distance is notably low. Since we align all sinks at their bottom in Section 5.4.1.3, the sink distances are always zero. Except of two layouts all source distances are zero. In these two layouts there are one and ten sources, respectively, whose source distance is one. In the upward planar representation the edges connecting the super source $\hat{s}$ and these sources were free of crossings. However, since the topology is relaxed in the shape and metrics steps, these eleven boxes were moved into an internal face again (see Figure 5.11 on page 66 for illustration).

As already discussed in the foregoing section, the number of sinks and the number of sources are lower bounds for the number of columns. Therefore, we analyse the number of columns divided by the maximum of the number of sources and the number of sinks. In most cases the layouts have at most twice as many columns as the lower bound. The average is 1.38 times the maximum of number of sources and sinks. There are two outliers that have four and five times as many columns. These two instances contain cycles which are removed by the method described in Section 5.1. However, we count the sources and sinks before the removal of cycles. Due to the cycle removal the number of sources and sinks in these two instances increase. Thus, the two outliers are comprehensible.

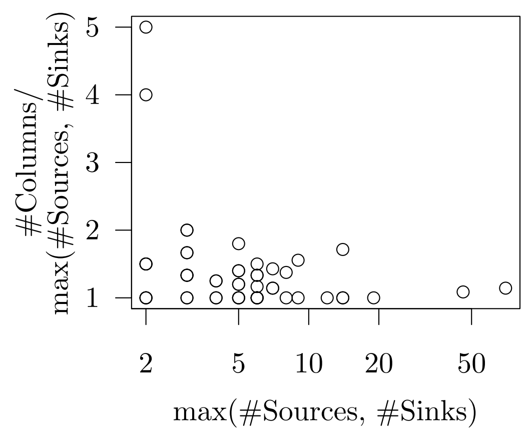


Figure 6.10: Number of columns in relation to the known lower bounds.

We measure the area of a layout by the area of the smallest bounding box containing the whole layout and set it into relation to the number of nodes and the number of edges. In most cases, this measure is less than 60, 000. However, there are two outliers. These two layouts have many shallow columns and few tall columns which force the bounding box to be tall as well. Thus, if the area would be measured using the contour of the layout instead of the bounding box, these outliers would assimilate.

Furthermore, we are interested in the aspect ratio of the layouts, i.e. the ratio of width to height (see Figure 6.12). Our preferred interval for the aspect ratio is from 0.5 to 2. There are 18 layouts, i.e. 35.3%, that are outside this interval. Only one of them has an aspect ratio that is too small, whereas 17 are too wide in comparison to their height. The far outliers having an aspect ratio of 7.9 or even more almost only consist of singleton nodes. Thus, the number of columns is high, whereas each column is only as tall as a single node.

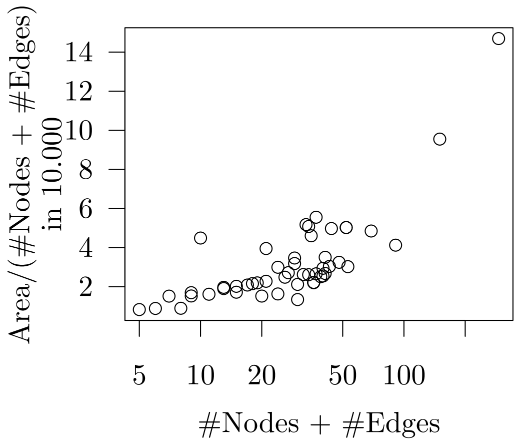


Figure 6.11: Area per node and edge.

Therefore, the aspect ratio is high as well. Ignoring these outliers, the aspect ratio of the layouts lie in an interval from 0.47 to 5.64. The average aspect ratio is 3.11.

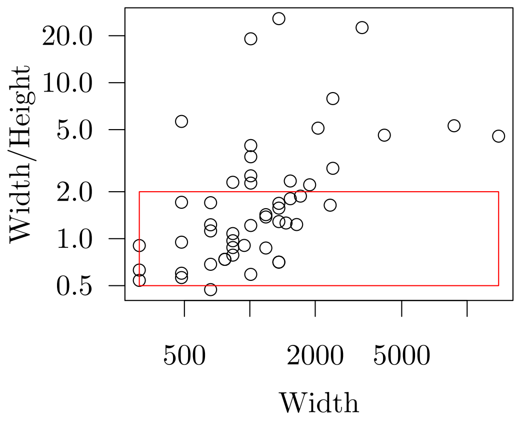


Figure 6.12: Aspect ratios of the computed layouts. The red rectangle indicates the preferred range for the aspect ratio.

Finally, we turn towards the runtime analysis. In Figure 6.13 we depict the maximum runtime over the number of nodes. The maximum runtime is 25,667ms for an instance having 134 nodes and 158 edges, whereas the average is 769.84ms. However, the average runtime is strongly influenced by the few large instances. Therefore, we give the maximum and the mean runtime for instances in our evaluation set of 51 argument maps that have up to 10, 20, 30 and 40 nodes in Table 6.1.

As stated in Chapter 2, 96% of the argument maps consist of at most 40 nodes and 78% have at most 20 nodes. Thus, for 96% of the instances a layout can be computed in 124ms on average. For 78% the mean of the computation time is 24ms.

It is interesting, how high the contribution of the three phases topology, shape and metrics to the overall runtime is. Shape and metrics make only insignificant contributions of 0.003% and 0.494%, respectively, whereas topology needs 99.503% of the overall runtime. Thus, the topology step is the bottleneck of the whole topology-shape-metrics approach.

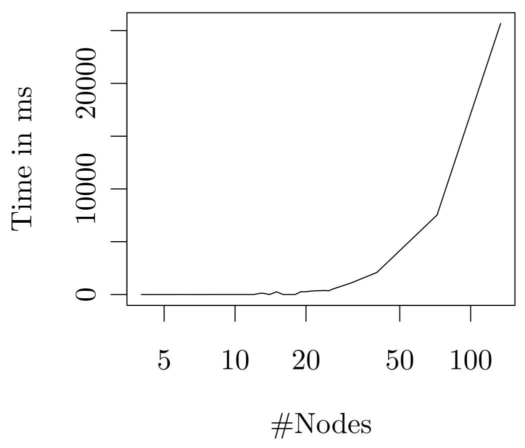


Figure 6.13: Maximum runtime broken down by the number of nodes.

#Nodes	Max in ms	Mean in ms
≤ 10	1	1
≤ 20	262	24
≤ 30	524	61
≤ 40	2095	124
All	25,667	769.84

Table 6.1: Maximum and mean runtime in our test suite broken down by the number of nodes.

Therefore, we decreased the number of randomised runs of both, the feasible upward planar subgraph and the topology computation, to three. Recall that for each topology computation run we compute a feasible upward planar subgraph. Thus, we compute $3 \cdot 3 = 9$ subgraphs in total – in comparison to 100 computations when using ten randomised runs. This reduced the average runtime over all 51 instances to 72.59ms, i.e. to 9.4%. Surprisingly, this severe runtime optimisation had no influence on 47 layouts – they did not change. On the 4 remaining instances the runtime reduction had only small impact, i.e. the number of crossings increased by 17% on average, whereas the other properties remained almost unchanged. For one instance having 20 nodes the number of crossings was increased from ten to eleven. A strong influence of this optimisation step is noticeable with three layouts containing $40, 72$ and 134 nodes, respectively. Therefore, we suggest to adjust the number of randomised runs to the size of the argument map that is to be layouted.

We conclude that the computed layouts are of high quality from an aesthetic point of view as well as regarding the statistics. Both, the number of crossings and the number of bends, are small. The total source/sink distance is 0 for 49 of 51 layouts. Furthermore, the layouts are compact and there are 33 layouts that have an aspect ratio in our preferred interval ranging from 0.5 to 2.
However, one could reconsider the drawing style in two points. Layouts having a large number of sources sinks are forced to have many columns. For these argument maps the aspect ratio exceeds the value two and the resulting layouts are unnaturally flat. Furthermore, the alignment of predecessors can lead to confusion, if a box is aligned with another that is several columns apart. In the following chapter we present our ideas how to cope with these shortcomings of the drawing style.

7. Conclusion

In this work we developed the characteristics of good layouts of argument maps and presented an algorithm that computes such layouts. In general, layouts of argument maps are orthogonal upward drawings. Since the boxes are of uniform width, we focus on column-based layouts, i.e. the canvas is divided into disjoint columns of uniform width that corresponds to the uniform width of the boxes. During the layout computation the boxes and vertical edge segments are positioned within these columns. Furthermore, we enforce alignment of predecessors, free sources and sinks, minimum spacing between boxes and edges as well as local symmetry. Besides the number of crossings, number of bends and total edge length, which are typical optimisation goals, we minimise the total source/sink distance as well. Thereby, we basically denote the distance from the sources and sinks of the graph to the outer face of the layout.

Our approach integrates the layer-free upward crossing minimisation technique recently published by Chimani et al. [CGMW10] into the well-known topology-shape-metrics framework. In the first step we minimise the number of crossings as well as the total source/sink distance. Afterwards, we use a modified graph drawing heuristic by Biedl and Kant [BK94] such that there are at most four bends per edge. Thereby, nodes and edges are assigned to columns. In the last step, we first minimise vertical edge length using a greedy approach and then horizontal edge length basing on a flow network. During the edge length minimisation we remain true to the columns such that the resulting layouts are column based. Unconventionally, we do not fix the topology computed in the first step throughout the remaining phases. When computing the column assignment in the second step, we relax some of the decisions we made before, and hence, the edge crossings are not determined anymore. We fix the edge crossings again when minimising the vertical edge length. In usual applications of the topology-shape-metrics framework, bend minimisation and total edge length minimisation are subordinated to crossing minimisation. However, using our approach we increase their significance.

Note that our algorithms for crossing and total source/sink distance minimisation, for bend minimisation and for vertical edge length minimisation are heuristics. For the first and third problem we proved $\mathcal{NP}$-completeness. However, a proof of the $\mathcal{NP}$-completeness of bend minimisation is missing.

The layouts we compute using this approach are of high quality and outperform our expectations. They have a clear, well-structured look and are compact as well. Input instances that have a typical size for argument maps can be layouted in 124ms in average. Thus, our algorithm is feasible for practical purposes as well and will be integrated into an editor for argument maps called argunet[1].

Future Work

There are problems in the context of layouts for argument maps we did not handle in this work. One of them is already described in Section 3.2: the grouping of boxes into a bounding box that contains no other boxes. However, we offer no methods to enforce this type of constraint.

In Section 6.1 we already broached that one could reconsider the drawing style of argument maps concerning two points: (i) The alignment of predecessors can lead to boxes that seem to be layouted unnecessary high, because they are aligned with boxes that are many columns apart (see the right side of Figure 6.3 and 6.4). This could be improved by taking the number of columns between the predecessors into account. Then, only boxes that are close to each other would be aligned. (ii) The constraint "free sources and sinks" enforces a high number of columns if the number of sources or sinks is high. Especially for large trees, the resulting layouts look unnatural (see Figure 6.6). One could either relax this constraint for some sources and sinks or omit it totally. We can easily adapt our width-compaction algorithm in Section 5.4.2.2 to omit the constraint for single sources or sinks by simply removing the edge connecting the source (sink) with the super source (super sink), because these edges keep the columns above the sources and below the sinks free.

Furthermore, it is of interest to separate the incoming edges such that the supporting edges are en bloc and the argument is centred below these edges. Then, the attacking edges would come from the left or right side. This would increase readability, because the supporting paths are layouted straightly top-down.

Finally, since the construction of argument maps is a dynamic process, it would be a great support for the user to create incremental layouts, i.e. to adapt the former layout after an insertion or deletion has been proceeded.

[1]See `www.argunet.org`

Bibliography

[BC11] G. Betz and S. Cacean, *The moral controversy about Climate Engineering - an argument map.* KIT, Karlsruhe, 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022371

[BDLM94] P. Bertolazzi, G. DiBattista, G. Liotta, and C. Mannino, "Upward drawings of triconnected digraphs," *Algorithmica*, vol. 12, pp. 476–497, 1994.

[BDMT98] P. Bertolazzi, G. DiBattista, C. Mannino, and R. Tamassia, "Optimal upward planarity testing of single-source digraphs," *SIAM Journal on Computing*, vol. 27, pp. 132–169, 1998.

[Bet10] G. Betz, *Theorie dialektischer Strukturen.* Klostermann, 2010.

[BFP$^+$73] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, vol. 7, pp. 448–461, 1973.

[BK94] T. Biedl and G. Kant, "A better heuristic for orthogonal graph drawings," in *Algorithms (ESA '94)*, ser. Lecture Notes in Computer Science, vol. 855. Springer, 1994, pp. 24–35.

[BMT97] T. Biedl, B. Madden, and I. G. Tollis, "The three-phase method: A unified approach to orthogonal graph drawing," in *Graph Drawing*, ser. Lecture Notes in Computer Science, G. DiBattista, Ed. Springer, 1997, vol. 1353, pp. 391–402.

[BNT86] C. Batini, E. Nardelli, and R. Tamassia, "A layout algorithm for data flow diagrams," *IEEE Transactions on Software Engineering*, vol. 12, pp. 538–546, 1986.

[Bra02] U. Brandes, "Eager *st*-ordering," in *Proceedings of the 10th Annual European Symposium on Algorithms (ESA '02).* Springer, 2002, pp. 247–256.

[BTT84] C. Batini, M. Talamo, and R. Tamassia, "Computer aided layout of entity relationship diagrams," *The Journal of Systems and Software*, vol. 4, pp. 163–173, 1984.

[CGMW10] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong, "Layer-free upward crossing minimization," *Journal of Experimental Algorithmics*, vol. 15, 2010.

[CGMW11] ——, "Upward planarization layout," *Journal of Graph Algorithms and Applications*, vol. 15, no. 1, pp. 127–155, 2011.

[DDPP02] G. DiBattista, W. Didimo, M. Patrignani, and M. Pizzonia, "Drawing database schemas," *Software: Practice and Experience*, vol. 32, no. 11, pp. 1065–1098, 2002.

[Des08] R. Descartes, *Meditationes de prima philosophia. Meditationen über die Grundlagen der Philosophie*, C. Wohlers, Ed. Meiner, 2008.

[DPP02] W. Didimo, M. Patrignani, and M. Pizzonia, "Industrial plant drawer," in *Graph Drawing*, ser. Lecture Notes in Computer Science, P. Mutzel, M. Jünger, and S. Leipert, Eds. Springer, 2002, vol. 2265, pp. 391–395.

[DT88] G. DiBattista and R. Tamassia, "Algorithms for plane representations of acyiclic digraphs," *Theoretical Computer Science*, pp. 175–198, 1988.

[EEK03] M. Eiglsperger, F. Eppinger, and M. Kaufmann, "An approach for mixed upward planarization," in *Proceedings of the 7th International Workshop on Algorithms and Data Structures (WADS' 01).* Springer, 2003, pp. 352–364.

[EFK00] M. Eiglsperger, U. Fößmeier, and M. Kaufmann, "Orthogonal graph drawing with constraints," in *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, 2000, pp. 3–11.

[Eig03] M. Eiglsperger, "Automatic layout of uml class diagrams: A topology-shape-metrics approach," Ph.D. dissertation, Eberhard-Karls-Universität zu Tübingen, 2003.

[EKS03] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller, "A topology-shape-metrics approach for the automatic layout of uml class diagrams," in *Proceedings of the 2003 ACM symposium on Software visualization (SoftVis '03)*, 2003, pp. 189–198.

[ELS93] P. Eades, X. Lin, and W. F. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Information Processing Letters*, vol. 47, pp. 319–323, 1993.

[EMW86] P. Eades, B. D. McKay, and N. C. Wormald, "On an edge crossing problem," in *Proceedings of the 9th Australian Computer Science Conference.* Australian National University, 1986, pp. 327–334.

[ET76] S. Even and R. E. Tarjan, "Computing an *st*-numbering," *Theoretical Computer Science*, vol. 2, no. 3, pp. 339–344, 1976.

[ET77] ——, "Corrigendum: Computing an *st*-numbering," *Theoretical Computer Science*, vol. 4, no. 1, p. 123, 1977.

[ET88] P. Eades and R. Tamassia, "Algorithms for drawing graphs: An annotated bibliography," Brown University, Providence, RI, USA, Tech. Rep., 1988.

[FHH$^+$90] M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F. T. Leighton, A. Simvonis, E. Welzl, and G. Woeginger, "Drawing graphs in the plane with high resolution," in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS '90)*, 1990.

[FK95] U. Fößmeier and M. Kaufman, "Drawing high degree graphs with low bend numbers," in *Proceedings of the 4th Symposium on Graph Drawing (GD '95)*, ser. Lecture Notes in Computer Science, vol. 1027. Springer, 1995, pp. 254–266.

[FMF$^+$01] S. P. Fekete, C. Matuszewski, R. Fleischer, M. Müller-Hannemann, C. Hirsch, G. Neyer, G. W. Klau, R. Weiskircher, B. Landgraf, and T. Willhalm, *Drawing Graphs - Methods and Models*, M. Kaufmann and D. Wagner, Eds. Springer-Verlag, 2001.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[GJ83] ——, “Crossing number is np-complete,” *Siam Journal On Algebraic And Discrete Methods*, vol. 4, no. 3, pp. 312–316, 1983.

[GM04] C. Gutwenger and P. Mutzel, “An experimental study of crossing minimization heuristics,” in *Graph Drawing*, ser. Lecture Notes in Computer Science, G. Liotta, Ed. Springer, 2004, vol. 2912, pp. 13–24.

[GT94] A. Garg and R. Tamassia, “On the computational complexity of upward and rectilinear planarity testing (extended abstract),” 1994.

[GT02] ——, “On the computational complexity of upward and rectilinear planarity testing,” *SIAM Journal on Computing*, vol. 31, pp. 601–625, 2002.

[HEH04] S.-H. Hong, P. Eades, and J. Hillman, “Drawing series parallel digraphs symmetrically,” *Comput. Geom. Theory Appl.*, vol. 29, no. 3, pp. 191–221, 2004.

[Kar72] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations.* Plenum Press, 1972, pp. 85–103.

[LNS85] R. J. Lipton, S. C. North, and J. S. Sandberg, “A method for drawing graphs,” in *Proceedings of the first annual symposium on Computational geometry*, 1985, pp. 153–160.

[OAS10] T. Opsahl, F. Agneessens, and J. Skvoretz, “Node centrality in weighted networks: Generalizing degree and shortest paths,” *Social Networks*, vol. 32, no. 3, pp. 245–251, 2010.

[PCJ96] H. C. Purchase, R. Cohen, and M. James, “Validating graph drawing aesthetics,” in *Graph Drawing*, ser. Lecture Notes in Computer Science, F. Brandenburg, Ed. Springer, 1996, vol. 1027, pp. 435–446.

[PCJ97] H. C. Purchase, R. F. Cohen, and M. I. James, “An experimental study of the basis for graph drawing algorithms,” *J. Exp. Algorithmics*, vol. 2, 1997.

[Pur97] H. C. Purchase, “Which aesthetic has the greatest effect on human understanding?” in *Proceedings of the 5th International Symposium on Graph Drawing (GD ’97).* Springer, 1997, pp. 248–261.

[See97] J. Seemann, “Extending the sugiyama algorithm for drawing uml class diagrams: Towards automatic layout of object-oriented software diagrams,” in *Proceedings of the 6th Symposium on Graph Drawing (GD ’97)*, ser. Lecture Notes in Computer Science, G. DiBattista, Ed., vol. 1353. Springer, 1997, pp. 415–424.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda, “Methods for visual understanding of hierarchical system structures,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[Tam87] R. Tamassia, “On embedding a graph in the grid with the minimum number of bends,” *SIAM Journal on Computing*, vol. 16, pp. 421–444, 1987.

[TDET98] I. G. Tollis, G. DiBattista, P. Eades, and R. Tamassia, *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1998.