Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 18, no. 5, pp. 677–708 (2014)
DOI: 10.7155/jgaa.00341

Column-Based Graph Layouts

Gregor Betz[1] *Andreas Gemsa*[1] *Christof Mathies* *Ignaz Rutter*[1]
Dorothea Wagner[1]

[1]Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany

Abstract

We consider orthogonal upward drawings of directed acyclic graphs with nodes of uniform width but node-specific height. One way to draw such graphs is to use a layering technique as provided by the Sugiyama framework [34]. To overcome one of the drawbacks of the Sugiyama Framework, namely, unnecessary edge crossings caused by an unfortunate layer assignment of the nodes, Chimani et al. integrated their layer-free upward crossing minimization algorithm [9] into the Sugiyama framework [10]. However, one drawback of the Sugiyama framework still remains. If the heights of the nodes are non-uniform, the result of the approach can be a non-compact layout. In contrast, we avoid both of these drawbacks by integrating layer-free upward crossing minimization into the topology-shape-metrics (TSM) framework introduced by Tamassia [35]. Our approach, in combination with an algorithm by Biedl and Kant [4] lets us generate *column-based layouts*, i. e., layouts where the plane is divided into uniform-width columns and every node is assigned to a column.

We study the complexity of the individual steps of the layout process systematically and propose efficient algorithms with provable guarantees. We show that our column-based approach allows to generate visually appealing, compact layouts with few edge crossing and at most four bends per edge. Furthermore, the resulting layouts exhibit a high degree of symmetry and implicitly support edge bundling. We evaluate our approach by applying it to several real-world examples.

Submitted:	Reviewed:	Revised:	Reviewed:	Revised:
August 2013	November 2013	January 2014	April 2014	August 2014
	Accepted:	Final:	Published:	
	December 2014	December 2014	December 2014	
	Article type:		Communicated by:	
	Regular paper		S.-H. Hong	

Preliminary results presented in this paper have appeared as “G. Betz, C. Doll, A. Gemsa, I. Rutter, and D. Wagner, “Column-Based Graph Layouts”, Proceedings of the 20th International Symposium on Graph Drawing (GD’12) , vol. 7704 of LNCS, pp. 236–247, Springer, 2013 [3].

E-mail addresses: gregor.betz@kit.edu (Gregor Betz) andreas.gemsa@kit.edu (Andreas Gemsa) (Christof Mathies) ignaz.rutter@kit.edu (Ignaz Rutter) dorothea.wagner@kit.edu (Dorothea Wagner)

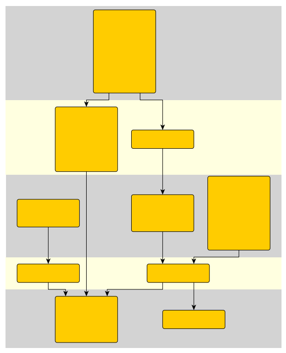
1 Introduction

One of the most well-known approaches for drawing directed acyclic graphs (DAGs) is the Sugiyama framework [34]. It consists of three steps, each of which can be solved individually: (i) layer assignment, (ii) determining relative positions within each layer to reduce edge crossings and (iii) positioning the vertices and edges. In the first step, the nodes are assigned to layers such that the target of each edge is in a layer above its source. The second step reduces the number of edge crossings by changing the order of the nodes within each layer. Finally, coordinates are assigned to the nodes as well as to bend-points of the edges.

The Sugiyama framework suffers mainly from two drawbacks. First, computing an unfortunate layering of the nodes in step (i) can enforce a large number of crossings in step (ii) which would not be necessary if another layering was chosen. Second, when applied for drawing graphs where the nodes are depicted as two-dimensional shapes (e. g., rectangles), a single node with large height can lead to non-compact layouts due to the way layers are defined; see Figure 1a for an example. One approach that resolves the first problem is the layer-free upward crossing minimization approach by Chimani et al. [9]. However, its integration into the Sugiyama framework [10] still leaves the second problem unresolved.

The motivation for the problems investigated in this paper stems from the application Argunet [30] that is used to create, display, and edit so called *argument maps*. Argument maps are used to visualize the structure of debates or argumentational analyses [8], and usually depicted as directed graphs with nodes of uniform width but node-specific height. Thus, we focus on orthogonal upward drawings of DAGs whose nodes are represented as rectangular boxes of uniform width. Graph drawing with given node sizes has been explored by Di Battista et al. [11]; they consider planar graphs with a fixed embedding and arbitrary node sizes. We deal with non-planar graphs without a given topology but with the restriction that the nodes have uniform width. For minimizing the number of crossings we use the algorithm by Chimani et al. [9] but, instead of integrating it into the Sugiyama framework [10], we integrate it into *topology-shape-metrics framework (TSM)* invented by Tamassia [35]. The TSM framework can be seen as a three-phase-method for orthogonal graph drawing: (i) fixing a planar embedding, (ii) computing an orthogonal description and (iii) compaction of the layout and coordinate assignment.

The integration of layer-free upward crossing minimization into the TSM framework lets us construct *column-based graph layouts.* In a column-based layout the plane 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 may span several columns. Due to the columns the final layouts have a clear look, which is exemplified in Figure 1b. The same graph with a layer-based layout computed by the graph editor yEd is shown in Figure 1a.


(a) A layer-based layout (computed by yEd).

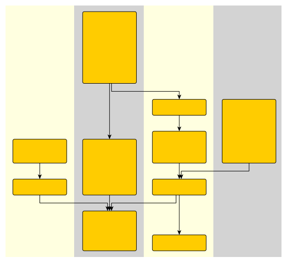
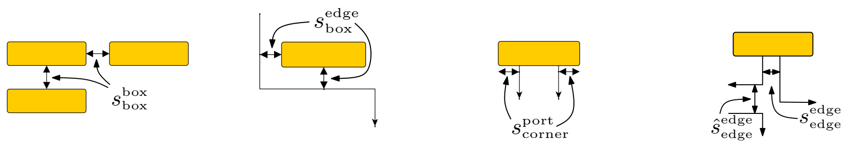

(b) A column-based layout (our approach).

Figure 1: A layer-based layout and a column-based layout of the same graph.

However, our approach differs from the usual TSM-based approach in one important detail. While we do compute a topology in the first step, we relax it already in the second step and consider only the left-to-right order of each node's outgoing edges and disregard crossing dummy nodes. Because of the relaxation of the topology, the final layout can have a higher number of crossings than the topology computed in the first step. However, this allows us to better optimize the vertical edge length in order to obtain more compact layouts. Thus, by relaxing the topology, we increase the significance of minimizing the number of bends and the total edge length as opposed to minimizing the number of edge crossings. This is reasonable since it has been established that crossings where edges cross at large angles (in this case *right angle crossings*) are only a minor obstacle for humans to understand the relationship between nodes in graphs [26].

We consider the following drawing style for our graph layouts: The drawings we generate are orthogonal drawings of DAGs whose nodes are represented by boxes of uniform width and individually prescribed heights. We require that edges leave their sources at the bottom and enter their targets at the top, and we further require that each edge is a monotonic downward polyline. We refer to such a drawing as an upward drawing. Note that upward drawings usually require strict monotonicity. However, we lessen this restriction in this paper to simple monotonicity, thus allowing also horizontal segments on the edges. We refer to the points where the edges enter the boxes as *ports*. We also include *minimum spacing constraints* that ensure that there is a minimum space between two boxes, a box and an edge, and between ports and corners of boxes, respectively. We denote the values of these spacing constraints by $s_{\text{box}}^{\text{box}}$, $s_{\text{box}}^{\text{edge}}$, and $s_{\text{corner}}^{\text{port}}$, respectively. We allow to specify two different spacing constraints for the case of two parallel edge segments: one for the distance between two edge segments whose edges have a common source or target and one for the remaining cases. We denote the former by $s_{\text{edge}}^{\text{edge}}$ and the latter by $\hat{s}_{\text{edge}}^{\text{edge}}$. By using these two different minimum spacings we enable *edge bundling*. For an illustration of the


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

(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 2: Illustration of the five types of spacing constraints.

different types of minimum spacing constraints see Figure 2.

Besides the mentioned drawing style, our approach guarantees certain properties that are aimed at creating visually appealing drawings. When looking at a single box of the layout, it is important that its predecessors can be found quickly. In order to give them a unifying look, we want as many predecessors as possible of each box to be *vertically aligned* at their bottom, which is similar to the requirement for an approach for layered graph drawing by Brandes and Köpf [7]. Additionally, we show how to assign nodes into columns such that, for each node v, its assigned column is the median of the columns assigned to the incoming edges of v as well as the median of its outgoing edges. We call this property *local symmetry*. Local symmetry makes it easy to identify a node's predecessors and successors, and provides well-structured looking layouts. Finally, we guarantee that each edge of the resulting layout will have at most *four bends*.

While satisfying these properties, we optimize the following three criteria, which are frequently employed as optimization goals in graph drawing [12]: (i) minimize the number of edge crossings, (ii) minimize the number of bends and (iii) minimize the total edge length.

A different approach to generate drawings of graphs that can be used to produce hierarchical or layered layouts is *constrained graph layout* [25], which is a general approach for drawing graphs. An input to constrained graph layout consists of the graph, a set of constraints for the coordinates of the vertices, and as third parameter suggested values for the variables corresponding to the coordinates of the vertices [15]. Constraints on the coordinates of the vertices can be used to model requirements similar to those we have for the drawings in this paper (e. g., minimum spacing constraints, columns). There have been several practical applications based on this general approach to layout different kind of graphs, e. g., dynamic biological networks [31], or engineering diagrams [14].

We note that, although our motivation stems from generating layouts for argument maps, our approach appears to be applicable to other kinds of diagrams such as the UML class diagrams. UML class diagrams depict the internal structure of a software system and are used extensively in software engineering [6]. A UML class diagrams has for each class a node, which is depicted as a box with height that usually depends on the number variables and methods of the class. Each class node may have an individual prescribed width, but it is not unreasonable to set a uniform value for the width of all boxes (which we require for our approach to work).

There have been several approaches to automatically layout UML diagrams already. However, they are mainly an adaption of layering approaches [32, 18], and thus may suffer from the same problems all such layered drawings may suffer. However, Eiglsperger et al. [19] propose two algorithms, both based on the TSM framework, to draw UML class diagrams with orthogonal edges. They generate a mixed-upward planarization of the input graph. For their algorithm they do not need to restrict the width of the boxes to some uniform value which is necessary for our approach to work.

We note that our approach does not directly support the drawing of UML class diagrams as there additional requirements compared to argument maps, (e. g., labeling of edges, hyperedges). Including these requirements might be an interesting problem for further research.

Contribution and Outline. Our main result is an algorithmic approach to compute column-based graph layouts for directed acyclic graphs. This approach is based on the topology-shape-metric framework and utilizes Chimani et al.'s layer-free upward crossing minimization. Contrary to the standard approach of the TSM framework, we do not fix the topology computed in the first step but use it only as a basis for computing a column-assignment of the nodes in the second step. In the last step we only use this column assignment and ignore the topology from the first step. This lets us focus on optimizing other criteria than the number of edge crossings.

Note that we use the "upward" terminology throughout this paper, because it is established in the graph drawing community. Nevertheless, since this drawing style originates from a particular application—which requires that all edges are directed downwards—all figures in this work have downward directed edges.

The remainder of the paper is structured as follows. In the next section we give a short introduction to the TSM framework and our approach. Sections 3–5 are structured along the TSM framework. In these sections we present our complexity results and give a detailed description of the algorithmic approaches we use in each step of the framework. In Section 6, we present a brief experimental evaluation of our approach and, finally, in Section 7 we give a short conclusion.

2 Preliminaries

The TSM framework consists of the three basic steps *topology*, *shape*, and *metrics*. In the first step an embedding of the input graph is computed with the goal of minimizing the number of crossings in the final layout. For non-planar graphs, crossings are replaced by dummy nodes of degree 4, which we refer to as *crossing dummies*. The planarization 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 optimized with respect to the fixed topology. The shape is the assignment of bends to edges, the angles of these bends, and the angles between edges (for our case the last information is irrelevant). Usually, the goal of this step is to minimize the number of bends. Finally, in the step *metrics*, the edge lengths, the node dimensions, and the node positions are determined.

Our approach to compute column-based graph layouts is based on the TSM framework. This lets us split the problem of generating a graph layout into smaller subproblems and either solve them optimally or apply heuristics if they are $\mathcal{NP}$-hard. In the topology step, we minimize the number of crossings while enforcing that all edges are directed upwards. The result of this step is a so-called upward planar representation, which prescribes the left-to-right order of the edges incident to each node separately for in- and outgoing edges. During the shape computation we build upon this upward planar representation in order to compute a column assignment of the nodes and edges, i. e., we divide the plane into disjoint columns of uniform width and assign each node and each edge to a column. The column assignment induces four different properties—namely local symmetry, orthogonal edges, at most four bends per edge, and the port distribution. In the last step of the topology-shape-metrics-framework we only rely on the column assignment computed in the second step to compute the final coordinates of the boxes and edge bends. Our approach here differs to the common usage of the topology-shape-metrics-framework in that 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. This lets us assign higher significance to bend and total edge length minimization than in usual applications of this framework.

The main result of this paper is stated in the following theorem.

Theorem 1 *For a given connected directed acyclic graph $G = (V, A)$ a locally-symmetric, column-based layout with at most four bends per edge can be computed in $\mathcal{O}(|A|^3|V|)$ time and $\mathcal{O}(|A||V|^2)$ space.*

In the next sections we study the complexity of each step and describe our algorithmic approaches in detail. Combining the main theorems of each section proves Theorem 1.

3 Topology

In this section we deal with the first step of the TSM framework, which aims to find a topology of the input graph with few crossings. Before we come to the main part of the topology-step, we conduct a preliminary step in which the input graph $G = (V, A)$ (which is directed and acyclic) is transformed into an s-t-graph $\widehat{G} = (\widehat{V}, \widehat{A})$. An *$s$-$t$-graph* is a directed graph having a single source and a single sink. The transformation is done by adding a super source $\hat{s}$ and a super sink $\hat{t}$ to G and connecting them with all of its original sources and sinks, respectively. In the end, when converting the layout of $\widehat{G}$ to G, we simply omit $\hat{s}$ and $\hat{t}$ as well as all their incident edges. Note that for the algorithm by Chimani et al. [9], which our approach is based on, it is sufficient to consider

s-T-graphs (which in contrast to s-t-graphs may have several sinks). The reason why we consider s-t-graphs instead is due to the special requirement of layouting argument maps that there are no nodes placed above sources and below sinks. We refer to this as the *free sources/sinks* property. For other applications, this requirement can be ignored, possibly resulting in more compact layouts.

While planarity of arbitrary graphs can be tested efficiently, testing upward planarity is $\mathcal{NP}$-complete for general graphs [24]. Therefore, upward crossing minimization is $\mathcal{NP}$-complete as well. However, for s-t-graphs, the test for upward planarity can be done in polynomial time [1], and thus upward crossing minimization might be efficiently solvable for s-t-graphs; we call this problem problem st-Upward Crossing Minimization. Unfortunately, st-Upward Crossing Minimization is $\mathcal{NP}$-hard, as there is an easy reduction from Bipartite Crossing Number, which is only a minor modification of the original $\mathcal{NP}$-hardness proof by Garey and Johnson [22] that showed hardness of Bipartite Crossing Number. We note that Bipartite Crossing Number is known under a variety of names. Eades et al. refer to it as Left Optimal Drawing [17] and it is also often denoted by 2-Layer Straight-Line Crossing Minimization [27, 29].

Theorem 2 ([22]) st-Upward Crossing Minimization *is $\mathcal{NP}$-hard.*

3.1 Layer-free Upward Crossing Minimization

In this section we describe our algorithmic approach for finding a topology of our modified input graph $\widehat{G}$. Note that $\widehat{G}$ may be non-planar and, thus, we cannot directly compute an upward planar embedding.

Since we know that st-Upward Crossing Minimization is $\mathcal{NP}$-complete, we cannot hope to find an efficient algorithm that solves st-Upward Crossing Minimization, unless $\mathcal{P} = \mathcal{NP}$. We are nevertheless interested in finding some reasonably good solution to the problem. For this we now describe an algorithm for which, unfortunately, we cannot give any quality guarantee. We build on an algorithmic approach called "layer-free upward crossing minimization" by Chimani et al. [9]. They developed an upward crossing minimization method that works without a prescribed layer assignment of the nodes. Their approach is based upon two steps: (i) Find a large subgraph $U = (\widehat{V}, A')$ of $\widehat{G} = (\widehat{V}, \widehat{A})$ that is upward planar and *feasible*, i. e., the deleted edges can be inserted such that the resulting drawing is an upward drawing. (ii) Insert the deleted edges one by one. For each edge insertion first a crossing-minimal insertion is conducted. However, a crossing-minimal insertion of an edge can prohibit the insertion of the remaining edges in an upward planar way. In this case, the insertion is undone and the edge is reinserted using a simple heuristic. Note that when inserting edges, we do not count the crossings with edges incident to $\hat{s}$ or $\hat{t}$ since these edges are only used for technical reasons.

After applying this algorithm we obtain an *upward planar representation* of the input graph. An upward planar representation $(\mathcal{U}, \Gamma)$ of a DAG G is an upward planar graph $\mathcal{U} = (V_\mathcal{U}, A_\mathcal{U})$ together with an upward planar embedding Γ of $\mathcal{U}$. Crossings in the original graph have been replaced by crossing dummies.

The Algorithm. We omit details regarding the algorithm for computing a feasible upward planar representation by Chimani et al. but give only a sketch of the approach. For details we refer the reader to the work by Chimani et al. [9].

To obtain the upward planar representation, we first compute a feasible upward planar subgraph $U = (\widehat{V}, A')$ of $\widehat{G} = (\widehat{V}, \widehat{A})$ that contains all edges incident 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. This is required for the free sources/sink property of the resulting layout. We start with U containing only the incident edges of $\hat{s}$ and $\hat{t}$. The neighbors 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.

Note that, by construction, the subgraph U must be an upward planar s-T-graph. Combining this insight with the *Feasibility Lemma* [9, Lemma 3.6], leads to the conclusion that U is a feasible upward planar subgraph of $\widehat{G}$.

After computing the intermediate feasible upward planar subgraph U, we try to increase the number of edges U contains. To this end, we try to add the edges of $\widehat{A}$ that are not in U one by one and check after each insertion whether it remains a feasible upward planar subgraph. If this check fails, we undo the insertion and postpone inserting the edge to the second phase. In the second phase (of the approach), we compute the upward planar representation $(\mathcal{U}, \Gamma)$ of $\widehat{G}$. We begin by setting $\mathcal{U} = U$. Then, we add the edges of $\widehat{A}$ that are in $\widehat{G}$ but have not yet been added to $\mathcal{U}$. In this step also the crossing dummies are inserted to $\mathcal{U}$. As in [9] we reinsert the missing edges in a crossing-minimal way. If reinserting an edge in a crossing-minimal way leads to an embedding which prohibits inserting all remaining edges in an upward planar way, we remove the edge and use a heuristic proposed by Chimani et al. to insert this edge. This heuristic guarantees a valid upward planar embedding but may produce more crossings.

The algorithm that computes a subgraph that is upward and feasible requires $\mathcal{O}(|\widehat{A}|^2)$ time. The algorithm for crossing-minimal insertion of an edge a_i requires $\mathcal{O}(|\widehat{V}| + r)$ time, where r is the number of edges that have to be inserted after a_i, whereas the heuristic requires $\mathcal{O}(|\widehat{V}|^2 + r|\widehat{V}|)$ time. The value of r is in $\mathcal{O}(|\widehat{A}|^2)$ since per insertion step at most $\mathcal{O}(|\widehat{A}|)$ edges (along with $\mathcal{O}(|\widehat{A}|)$ dummy nodes) can be inserted. We need to repeat the insertion step $\mathcal{O}(|\widehat{A}|)$ times, hence the worst-case running time is $\mathcal{O}(|\widehat{A}|^3 \cdot |\widehat{V}|)$. We summarize the results in the following theorem.

Theorem 3 *The above algorithm computes the upward planar representation of an s-t-graph $\widehat{G} = (\widehat{V}, \widehat{A})$ in $\mathcal{O}(|\widehat{A}|^3 \cdot |\widehat{V}|)$ time.*

Chimani et al. suggest to randomize the algorithm for the computation of the feasible upward planar subgraph by changing the order in which the edges are considered during the depth first searches, and the order in which the remaining edges are reinserted. Among several runs the crossing-minimal upward planar representation is chosen as the overall result. We will take up this idea in the evaluation of our algorithm in Section 6.

4 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 for a given upward planar representation $(\mathcal{U}, \Gamma)$ of $\widehat{G}$. In this context, shape describes the number of bends on each edge and their bend directions.

The algorithm we present takes up the idea of Biedl and Kant on how to draw a graph with few bends in linear time [4] and adapts it to orthogonal upward drawings with at most four bends per edge. This approach leads to column-based drawings. Since there are at most four bends per edge and ports are at the bottom and the top of the source and target, respectively, an edge can have at most three vertical segments. The column of the first (last) vertical segment is already determined by the column assigned to the source (target, respectively). The position of the middle vertical segment is determined by assigning it to some column. A *column assignment* of a DAG $G = (V, A)$ is a mapping $col\colon V \cup A \to \mathbb{Z}$ that assigns each node $v \in V$ and each edge $e \in A$ a column.

Note that a column assignment already induces the shape, i. e., the information about edge bends and their directions. In the metrics step we will realize the column assignment, i. e., we compute a layout that respects the column assignment. Let L be a *valid layout* of a DAG $G = (V, A)$, i. e., L satisfies the local symmetry and four bends per edge constraints. By subdividing vertical segments we may assume that every edge has exactly three vertical segments. A layout L is a *realization* of a column assignment col, if we can divide the plane into disjoint columns of uniform width and number the columns from left to right such that each node $v \in V$ is positioned within column $col(v)$ and for each edge $e \in A$ the middle vertical segment is positioned within column $col(e)$.

Our algorithm positions the nodes beginning from the topmost node to the bottommost node. Hence, we require an order of the nodes for our algorithm that ensures that, when placing a node, that all its predecessors have been placed already. To do this we can simply compute a topological order for all vertices beginning with $\hat{s}$, which can be done in linear time.

Now, the column assignment algorithm works as follows: The nodes of $\widehat{G}$ are treated according to the computed topological order given by $\hat{s} = v_0, \ldots, v_{n+1} = \hat{t}$. Thus, the algorithm that computes the column assignment starts 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 $(\mathcal{U}, \Gamma)$ such that they are distributed evenly to the left and right of column 0 without introducing gaps.

The invariant of our algorithm is that, when processing vertex v_i, all incoming edges of v_i are already assigned to columns. Then we assign v_i to column m—the

Algorithm 1: 4-Bend Column Assignment

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

column of the median incoming edge—and assign the outgoing edges of v_i to columns according to their left-to-right order induced by $(\mathcal{U}, \Gamma)$. 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$(v_i) - 1$ empty columns to which the outgoing edges of v_i can be assigned, according to the left-to-right order prescribed by the upward planar representation $(\mathcal{U}, \Gamma)$. For a pseudo-code description of this algorithm see Algorithm 1. We now prove that the column assignment computed by our algorithm is realizable.

Theorem 4 *The above algorithm computes a realizable column assignment for a given DAG $\widehat{G} = (\widehat{V}, \widehat{A})$ in $\mathcal{O}(|\widehat{A}|)$ time and space.*

Proof: A layout of $\widehat{G}$ can be constructed inductively during the execution of our shape algorithm. Since the x-coordinates are fixed by the column assignment, we only need to deal 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. Thus, no box can be intersected by an edge, and hence, this approach always yields a valid layout.

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. Furthermore, the shape algorithm 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 layout satisfies the local symmetry property.

We now turn towards the running time analysis. The algorithm that computes the topological order of the nodes in $\mathcal{U}$ runs in linear time. The time complexity of the shape algorithm itself depends on the implementation of the two methods SHIFTLEFT and SHIFTRIGHT that shift the already existing columns to the left or to the right. These methods are called $|V_{\mathcal{U}}|$ times in total. Biedl and Kant suggest an approach that yields an overall running time of $\mathcal{O}(|\widehat{V}|)$ [4]. 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 are going to be inserted are known SHIFTLEFT and SHIFTRIGHT can then be implemented to run in constant time. We also need to compute the median m of the incoming edges for a node. This can be done in $\mathcal{O}(\text{in-degree}(v_i))$ time [5]. We sum up over all iterations:

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

Thus, the shape algorithm can be implemented as an $\mathcal{O}(|\widehat{V}| + |\widehat{A}|) = \mathcal{O}(|\widehat{A}|)$-algorithm.

Of course, we need $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ space in order to store the assigned columns. Additionally, we need $\mathcal{O}(|\widehat{A}|)$ space in order to store the doubly linked listed that represents the columns. Thus, in total the shape algorithm needs $\mathcal{O}(|\widehat{V}| + |\widehat{A}| + |\widehat{A}|) = \mathcal{O}(|\widehat{A}|)$ space. □

Note that if the upward planar representation $(\mathcal{U}, \Gamma)$ contains no crossing dummies, then the realization constructed in the proof of Theorem 4 is upward planar as well.

Observation 1 *The realizations of the column assignments of an upward planar representation $(\mathcal{U}, \Gamma)$ that contains no crossing dummies are upward planar layouts.*

As we already have mentioned, during the construction of the column assignment, we only respect the left-to-right order of the outgoing edges at each node of $\widehat{G}$ which is prescribed by $(\mathcal{U}, \Gamma)$. We do not consider the crossing dummies in $\mathcal{U}$. Thereby, we relax the topology that we computed in the first step. Because of the relaxation of the topology, the final layout can have a higher number of crossings than the topology. On the other hand, this lets us focus on producing more compact layouts. Since we still maintain parts of the result of the first step, which minimized the number of crossings, we expect only a small increase in the number of crossings. Moreover, the crossings in our drawings are only right-angle crossings, and thus do not hinder readability that much.

5 Metrics

In the last phase of the approach, we compute the final coordinates of boxes and edges while minimizing the total edge length. The topology that we originally computed in the first step has no direct influence anymore. Instead, we only rely on the column assignment computed by the shape algorithm.

The metrics step consists mainly of two substeps. First, we focus on minimizing the vertical edge length. Since we can show that this problem is $\mathcal{NP}$-complete, we suggest a simple heuristic in order to solve it. Afterwards, we minimize the horizontal edge length. Note that all operations are performed with respect to the columns so that the final result is a column-based layout.

5.1 Vertical Edge Length Minimization

We begin by showing that finding a realization of a column-assignment with minimum total vertical edge length (or, equivalently, minimum total height) is $\mathcal{NP}$-complete. Afterwards we propose a simple greedy algorithm to reduce the total vertical edge length.

5.1.1 Computational Complexity

We call the problem of finding a realization of a column assignment with minimum total vertical edge length Vertical Edge Length Minimization. The formal definition of the problem is as follows.

Instance: An s-t-graph $\widehat{G} = (\widehat{V}, \widehat{A})$, a column assignment for each node in $\widehat{V}$ and each edge 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 edge bends of $\widehat{G}$ such that the resulting layout is valid and the total vertical edge length is at most k?

We prove $\mathcal{NP}$-completeness of Vertical Edge Length Minimization by reduction from 3-Partition, which is defined as follows [21].

Instance: A finite set $A = \{a_1, \ldots, a_{3m}\}$ 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 $S_1, S_2, \ldots, S_m$ 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 $\mathcal{NP}$-complete in the strong sense. 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.

In the following, we describe how to transform a 3-Partition instance to a Vertical Edge Length Minimization instance. This means we need to describe how to construct an s-t-graph and a corresponding column assignment of all vertices and edges of the s-t-graph in polynomial-time such that a solution of Vertical Edge Length Minimization of this instance induces a solution of 3-Partition.

The key idea behind our reduction is to use a single column that we divide into m areas, which we call *cells*, and we want that the each of the m cells corresponds to one of the m sets in a solution to a 3-Partition instance. The elements in A are represented in our reduction by nodes in the same column as the cells and whose heights is their "size" in the 3-Partition instance. Ultimately, we want that in an optimal solution of Vertical Edge Length Minimization (in our reduction) from a 3-Partition instance $\mathcal{I}$ there are exactly three nodes in each cell whose heights add up to exactly B if and only if $\mathcal{I}$ is a Yes-instance.

For ease of argumentation we set all minimum spacing constraints to 0, but we note that the reduction itself does not strictly require that all minimum spacing constraints are 0, and only minor modifications are necessary for the reduction still to work for arbitrary spacing constraints. However, setting the minimum spacings constraints to 0 has the unfortunate side-effect that the drawings are not visually appealing. For the figures in this section we assume that all minimum spacing constraints are set to a small positive value.

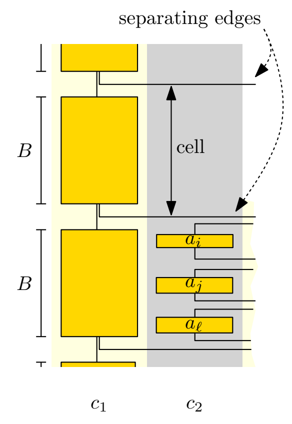


Figure 3: Sketch of the cells.

In our reduction all nodes are placed within $|A| + 3$ adjacent columns, which we denote by $c_1, \ldots, c_{|A|+3}$. All cells will be inside column c_2. Unless stated otherwise, nodes in our graph have the same height h, where h is a small but fixed value.

We now discuss the construction of the cells. For each cell we add two nodes with height B to our graph, one assigned to column c_1 and one assigned to the rightmost column $c_{|A|+3}$ (we need the columns between c_1 and $c_{|A|+3}$ for additional nodes, which we describe later). For the i-th cell we denote the node placed in column c_1 by ℓ_i, and the node placed in column $c_{|A|+3}$ by r_i. Now, we divide column c_2 by adding a directed edge (ℓ_i, r_{i+1}) for every i, $0 \leq i \leq m+1$. We call these edges *separating edges*; for a sketch see Figures 3 and 4. Note that we require nodes ℓ_i and r_i for $i = 0, \ldots, m+1$ in order to achieve that each of the m cells is bounded by a separating edge from above and from below. Finally, we connect the nodes $\ell_0, \ldots, \ell_{m+1}$ such that they form a directed path from ℓ_0 to ℓ_{m+1} and do the same for the nodes $r_0, \ldots, r_{m+1}$. Since the topmost and bottommost nodes of each path do not themselves span a cell, we set their heights to h.

For each $a_i \in A$ we add a node v_i, which we refer to as *element node*, with height $s(a_i)$ to our graph. Since the cells are meant to correspond to the sets in the 3-Partition instance, we must be able to position any of the element nodes v_i in any of the cells. On the other hand, we must be sure that placing a node outside of the cells increases the total vertical edge length. To ensure this we

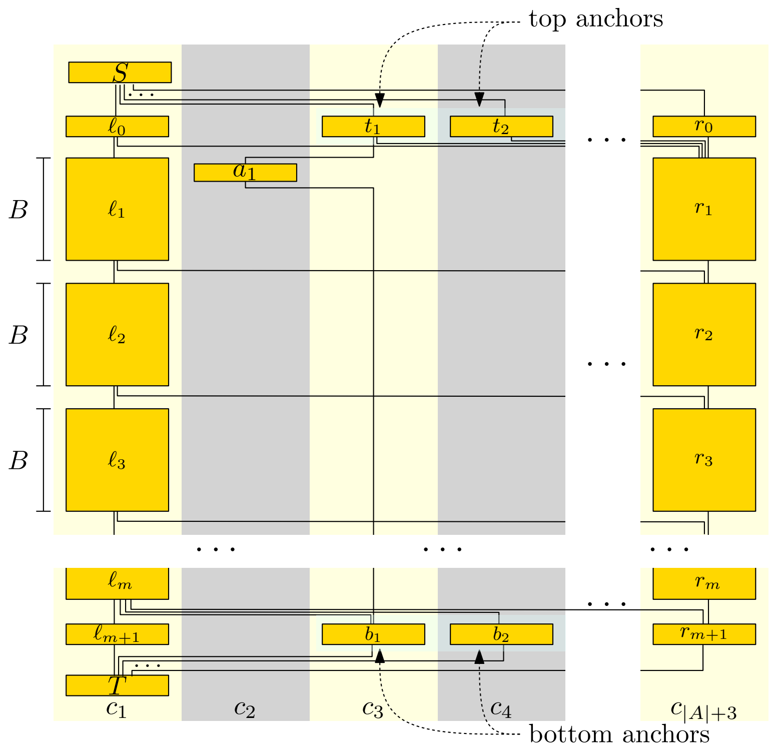


Figure 4: Sketch of the Hardness Proof.

make use of *anchor nodes*. For each element node v_i we add two anchor nodes, a *top anchor* t_i and a *bottom anchor* b_i, both assigned to column c_{i+2}, along with directed edges (t_i, v_i) and (v_i, b_i) to our graph. To ensure that the anchor nodes themselves stay above/below the cells we add for each top anchor t_i a directed edge (t_i, r_1) and for each bottom anchor a directed edge (ℓ_m, b_i).

Since the top (bottom) anchors stay above (below) the cells, moving the element nodes upwards or downwards is cost neutral, because the sum of the vertical edge lengths on the two edges incident to the anchors remains unchanged. Further, note that there are no edges between the different element nodes, i. e., their vertical ordering is not constrained. Since a element node v_i has height $s(a_i)$, finding three nodes whose total height sums up to B means we can place them in the same cell without increasing the total vertical edge length.

We are nearly finished with the construction of our graph, the only thing left is to ensure that the graph we constructed is an s-t-graph. This can easily be achieved as follows. Add two nodes S and T to the graph, assign them both to column c_1 and add directed edges from S to all top anchors, as well as ℓ_0 and r_0, and add directed edges from all bottom anchors, as well as ℓ_{m+1} and r_{m+1} to T.

Further, to obtain a complete column assignment we assign each edge to the column of its source. We have now finished the construction of the graph and the column assignment. It remains to compute the total vertical edge length k in an optimal solution to our Vertical Edge Length Minimization instance if the corresponding 3-Partition is solvable. Since we have chosen all spacing constraints to be 0, and each element a_i contributes $mB - s(a_i)$ to the vertical edge length, the minimum possible vertical edge length of the edges incident to the element nodes is $3m^2B - mB$. All remaining edges may have vertical edge length 0. We set $k = 3m^2B - mB$.

Correctness. We now show, that the 3-Partition instance has a solution if and only if the Vertical Edge Length Minimization instance is solvable. Obviously, if the 3-Partition instance has a solution, then it can be transformed to a solution of the Vertical Edge Length Minimization instance by vertically ordering the element nodes. If the Vertical Edge Length Minimization instance is solvable, we need to show that the 3-Partition instance has a solution as well. If the solution of Vertical Edge Length Minimization contains nodes that are not placed inside one of the cells, then this increases the total vertical edge length since there is no possibility of decreasing the total edge length elsewhere. Further, should there be more than three nodes in one cell, then, the sum of the heights of those nodes must add up to more than B (recall that $B/4 < s(a_i) < B/2$). Since the minimum height of each cell is B this increase in vertical edge length cannot be compensated elsewhere. Hence, we can conclude that in an optimal solution to Vertical Edge Length Minimization of a Yes-instance of 3-Partition there are exactly three nodes in each cell. This directly induces a solution to the corresponding 3-Partition instance.

Hence, we can conclude that Vertical Edge Length Minimization is $\mathcal{NP}$-hard. Since it is easy to see that Vertical Edge Length Minimization is contained in $\mathcal{NP}$, Vertical Edge Length Minimization is $\mathcal{NP}$-complete. We summarize this result in the following theorem.

Theorem 5 Vertical Edge Length Minimization *is $\mathcal{NP}$-complete.*

In the following we suggest a three-step approach for coping with the problem of vertical edge length minimization.

5.1.2 Greedy Algorithm

Since Vertical Edge Length Minimization is $\mathcal{NP}$-complete, we propose a greedy algorithm to minimize the vertical edge length. Recall that we want each node to have as many predecessors as possible vertically aligned at their bottom. We integrate this requirement into our vertical edge length minimization approach. In the first step we determine a partition of the nodes into sets which we call *groups*, each of which contains only nodes that can be aligned at their bottom. Hence, we need to be careful which nodes belong to the same group. It is easy to see that there must not be a directed path between any two nodes belonging to the same group. Further, it is necessary that the groups can be topologically ordered, i. e., the graph that we obtain by contracting each group to a single node must remain acyclic. In the second step, we compute a reverse topological order of these groups and finally, in the third step, we position the groups according to this order.

Grouping. We call a set of pairwise disjoint groups of nodes $g_1, \dots, g_k$ *feasible* if the graph that is obtained from $\widehat{G}$ by contracting each group to a single node is acyclic. A node $s \in \widehat{V}$ is an *immediate predecessor* of another node $s' \in \widehat{V}$ if there is an edge $(s, s') \in \widehat{A}$, and there is no other directed path from s to s' in $\widehat{G}$. We denote the set of immediate predecessors of a node s that have not been assigned to a group by $\textsc{Pred}(s)$.

The algorithm works in several iterations, each of which greedily computes a feasible set groups. We describe the ith iteration. If all nodes have been assigned to a group, the algorithm terminates. Otherwise, the algorithm creates a group g_i that contains an arbitrary node of $\widehat{G}$ that has not yet been assigned to a group. The algorithm then tries to increase the size of g_i as much as possible by determining nodes that are potential candidates for inclusion in g_i. Because we want the nodes in each group to be vertically aligned at their bottom, potential candidates are nodes that share the same successor with a node already contained in g_i. More specifically, the algorithm determines for each node v in g the set $\textsc{Succ}$ of their successors. For each node $s \in \textsc{Succ}$ the algorithm determines the set of immediate predecessors $\textsc{Pred}(s)$ that are not yet assigned to a group, and whether adding $\textsc{Pred}(s)$ to g_i yields a feasible set of groups, i.e., whether $g_1, \dots, g_{i-1}, g_i \cup \textsc{Pred}(s)$ is feasible. If this is the case, then we replace g_i by $g_i \cup \textsc{Pred}(s)$; otherwise the algorithm continues with the next successor $s' \in \textsc{Succ}$. Note that adding new nodes to g_i can require adding new nodes to $\textsc{Succ}$. When no new nodes can be added to g_i in this way, the ith iterations finishes. For a pseudo-code description of this algorithm see Algorithm 2.

Lemma 1 *The grouping algorithm can be implemented to run in* $\mathcal{O}(|\widehat{V}|^2 \cdot |\widehat{A}|)$ *time and* $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ *space.*

Proof: In each iteration i of the algorithm, the key operation is to determine whether $g_1, \dots, g_{i-1}, g_i \cup \textsc{Pred}(s)$ is a feasible set of groups. To check this, we maintain the DAG that is obtained from $\widehat{G}$ by contracting each group g_i to a single node G_i. Including $\textsc{Pred}(s)$ into g_i introduces a cycle in this graph if and only if there is a path from a node in $\textsc{Pred}(s)$ to G_i and a path from G_i to a node in $\textsc{Pred}(s)$. This can be easily checked with two breadth-first searches. Note that the contracted graph has size $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$, and hence the test can be performed in time $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$. When we include a set $\textsc{Pred}(s)$ into g_i, we immediately contract the nodes in $\textsc{Pred}(s)$ into G_i. Clearly such a contraction is performed only once per node, and hence this takes total time $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ over all iterations.

Algorithm 2: Grouping Algorithm

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

The running time of the algorithm is dominated by lines 4-11 by the two nested loops and the feasibility check. The outer loop can be executed $\mathcal{O}(|\widehat{V}|)$ times since there are $\mathcal{O}(|\widehat{V}|)$ groups. We can also bound the number of iterations for the inner while loop by $\mathcal{O}(|\widehat{V}|)$ if we can bound the number of elements that are added to Succ by $\mathcal{O}(|\widehat{V}|)$. We can guarantee this if we store during each iteration for each node if it has already been added to Succ before. Then, before adding a node to Succ (l. 10) we first check whether we have added the node before already. If this is the case we do not add it to Succ again. Finally, for the feasibility check we require two breadth-first searches that require $\mathcal{O}(|\widehat{V}|+|\widehat{A}|)$ time each. Since $\mathcal{O}(|\widehat{V}|) \subseteq \mathcal{O}(|\widehat{A}|)$ the running time of our algorithm is $\mathcal{O}(|\widehat{V}|^2 \cdot |\widehat{A}|)$. □

Computing the Order. After computing the grouping, we compute an order of the groups in the second step. 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 already positioned. To do this we contract each group and sort them topologically from bottom to top. Recall that our algorithm ensures that when adding new nodes to a group the group remains feasible, i. e., there is no directed path in the graph from one node in the group to another in the same group. This directly implies that we can always order the groups topologically.

Coordinate Assignment. 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 previous step. The y-coordinates of the ports and the bends of an edge are computed in two parts. When the target of an edge is positioned, we compute the y-coordinates of the target port and the first two bends. The first bend is in the target's column, whereas the other one is in the column assigned to the edge. The two last bends and the source port are computed, when its source is positioned.

First, we compute the smallest possible y-coordinate $\hat{y}$ for the lower boundary of the nodes in g_i by setting it to the maximum of the smallest possible y-coordinate for each node $v_j \in g_i$. When computing this y-coordinate for a node $v_j \in g_i$, we need to consider that we have not yet routed the second part of the edges whose targets are in g_i. For this, we need to compute the smallest possible y-coordinate for the horizontal edge segment of the edges we still need to draw. In particular, we need to find the smallest possible y-coordinate such that all minimum spacing constraints are respected. This can be done straightforwardly.

After determining $\hat{y}$, 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. We draw the horizontal segments as high as possible. Afterwards, we draw the incoming edges as low as possible. To this end, 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 running time of this step is $\mathcal{O}(|\widehat{V}| + |\widehat{A}| \cdot b)$, where b is the number of columns used by the assignment. Note that since every column contains a vertex it follows that b is in $\mathcal{O}(|\widehat{V}|)$. 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 smallest possible y-coordinate needs to be determined, which requires as many operations as the drawn horizontal segment spans columns. We simply bound this number of columns by $|\widehat{V}|$.

Theorem 6 *The vertical edge length minimization heuristic can be implemented to run in $\mathcal{O}(|\widehat{V}|^2 \cdot |\widehat{A}|)$ time and $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ space.*

Proof: The computation of the groups in step one takes $\mathcal{O}(|\widehat{V}|^2 \cdot |\widehat{A}|)$ time, afterwards the groups can be ordered in $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ time. For the final y-coordinate assignment we need $\mathcal{O}(|\widehat{V}| + |\widehat{A}| \cdot |\widehat{V}|)$ time. Thus, the greedy approach runs in $\mathcal{O}(|\widehat{V}|^2 \cdot |\widehat{A}|)$ time in total.

For the computation of the groups $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ 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 at most $\widehat{V}$ columns the highest object that is already drawn. Thus, in total our algorithm requires $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ space. □

5.2 Horizontal Edge Length Minimization

There has been some research into improvement of orthogonal drawings which is related to the approaches used in this section. Six et al. [33] discuss several refinement steps for orthogonal drawings. They propose heuristics to remove U-Turns (which we refer to as *bows*; see Figure 5), superfluous bends, self-crossings,

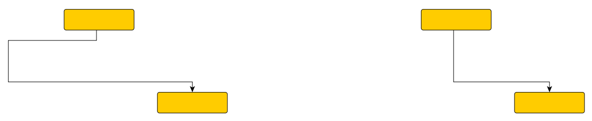


Figure 5: An edge that is a bow (left) and the edge after bow reduction (right).

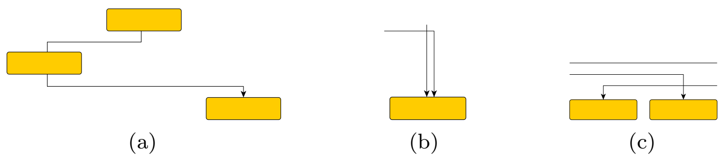


Figure 6: Visually unappealing parts of a drawing which can be resolved in a post-processing step.

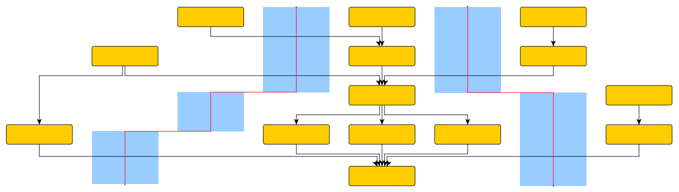
and several other unappealing structures. However, we encounter, due to the way our algorithm works, only few of the problems described in this. The only structure we remove are bows, by using a simple linear time algorithm. However, there are other examples of structures that are not visually appealing and that can easily be resolved but which we did not consider for the implementation; see Figure 6 for some examples.

In the second step, we employ a heuristic approach to minimize the width of the whole layout. Due to the shifting in each iteration of the shape algorithm, we introduce new columns. Some of them are necessary while others will be partly empty in the final layout and may be removed while maintaining a valid layout. Thus, the final layout is possibly wider than actually necessary. In Figure 7a 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.

Our approach is somewhat similar to the *moving* part of the 4M-algorithm proposed to improve orthogonal drawings [20]. The authors define a *moving line* which is directed through the drawing and is used to identify empty space in the drawing which can be removed safely. However, the approach requires the node heights to be a multiple of a certain base value and the nodes must be aligned on a grid. Both does not apply to our drawings.

We perform width compaction along so-called *compaction paths*. A compaction path is a piecewise axis-parallel y-monotone path passing through a layout from top to bottom. It may cut horizontal edge segments but neither vertical edge segments nor nodes. 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 7 we illustrate how a compaction along a compaction path is performed. First, 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


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

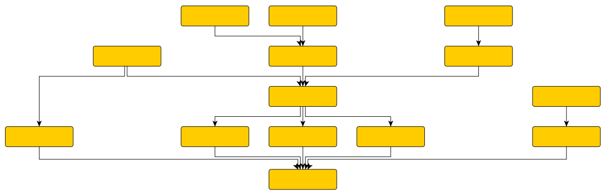
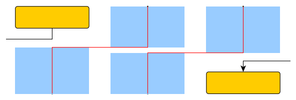

(b) How to perform compaction along the left compaction path.

Figure 7: Compaction paths and compaction along a compaction path.


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

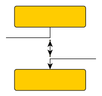

(b) After the compaction the spacing constraint between the two edges can be violated.

Figure 8: An invalid set of compaction paths.

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

A seemingly obvious approach to compute a maximum set of compaction paths would be by using a flow algorithm. Traditionally, flow algorithms have been used in graph drawing to solve such compaction problems [23, 35]. However, in our case there are examples of compaction paths where each path by itself is valid, but compacting two such paths at the same time yields a layout where an edge-edge spacing constraint is violated; see Figure 8. Although it is possible to avoid this problem when using flow networks (e. g., by increasing the size of the drawing), the flow network itself is very complicated (see [13] for a description) and a simpler approach appears to be reasonable. We propose an approach that iteratively removes columns instead of removing multiple columns at once. This helps us to avoid the problem stated above.

In the following, we describe an approach that computes a cardinality-maximal *valid* set of compaction paths using right-first search. We call a set $\mathcal{P}$ of compaction paths valid if there is no pair of paths P_i, P_j for $i \neq j$ in $\mathcal{P} = \{P_1, \dots, P_k\}$ that cross each other, or overlap. Since the paths do not cross or overlap, we can order them from right to left as $P_1, \dots, P_k$. We also require that for $i = 1, \dots, k-1$ there is no path P' distinct from $P_1, \dots, P_{i+1}$ whose nodes lie to the right of, or on P_{i+1} such that $\{P_1, \dots, P_i, P'\}$ is a valid set of compaction paths. For the first path P_1 this means that no subpath of it can be replaced by a path that is right of P_1 such that the resulting path is a valid compaction path. We call a valid set of compaction paths that has these properties *rightmost*. In Figure 9 we depict a rightmost valid set of compaction paths for the example of Figure 7a. Note that since all paths are directed downwards, a right-first search yields paths that are as left as possible in the drawing.

It remains to argue that by using right-first search we actually find a cardinality-maximal valid set of compaction paths. In the following we show that we can transform any valid set of compaction paths $\mathcal{P} = \{P_1, \dots, P_k\}$ to a rightmost set of compaction paths of the same cardinality. First, we remove all crossings by swapping the suffixes of any two paths in the set that cross. Note that by doing this we cannot introduce a new crossing. Since the resulting paths $P_1, \dots, P_k$ do not cross, we can order them from right to left. We treat the paths P_i in increasing order of their indices (from right to left) and try to move each path rightwards. More specifically, for each treated path P_ℓ we a replace a subpath $(p_i, \dots, p_j)$ of it with a path $(q_1, \dots, q_l)$ that is right of P_ℓ and does not overlap or cross a path $P_{\ell'}$ in the set $\mathcal{P}$ with $\ell' < \ell$.

After these normalization steps, we obtain a rightmost valid set of compaction paths. Note that we do not change the number of paths during the normalization. Thus, there exists a unique maximum set of compaction paths that is rightmost, which can be found by the right-first approach we present in the following paragraph.

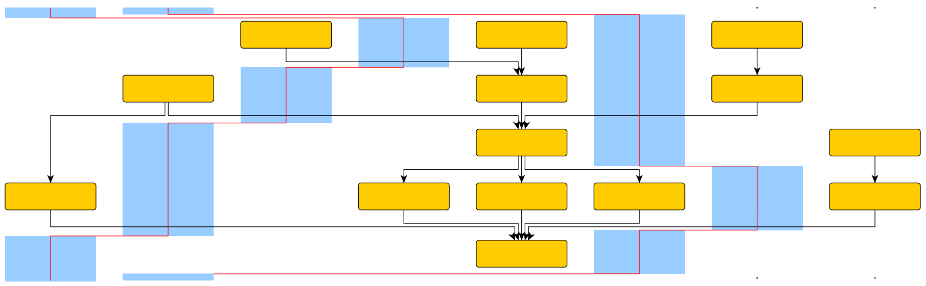


Figure 9: A rightmost valid set of compaction paths.

The Algorithm. In this paragraph we present the algorithm that performs the width compaction. Before we run the algorithm, we remove the super source $\hat{s}$ and the super sink $\hat{t}$. As already mentioned, 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.

First, we describe the initialization of the graph $\mathcal{D}$, which we refer to as the *compaction network*. 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 contains the boxes and edge segments that are contained in 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 of the (at most 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.

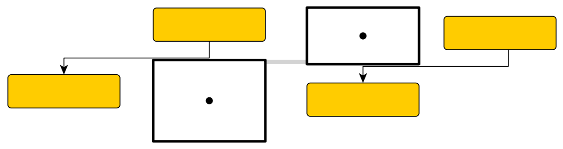
Afterwards, we detect the free rectangles in the columns, i. e., the gaps between two entries in the list whose y-ranges do not touch. Furthermore, we assume that on the top and on the bottom of each column is a large free rectangle. For each free rectangle we add a node to $\mathcal{D}$. For ease of notation we identify each node of $\mathcal{D}$ with its corresponding free rectangle.

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 neighboring columns if their y-ranges overlap. Consider two overlapping rectangles in neighboring columns as depicted in Figure 10b. If a compaction path p runs through these rectangles as shown in Figure 10a, compacting along p would move the right box under the left one. Then the distance between the two edges needs to be at least $s^{\text{edge}}_{\text{edge}}$ or $\hat{s}^{\text{edge}}_{\text{edge}}$, respectively. Note that this distance after the compaction equals to the overlap of the y-ranges of the two rectangles. Therefore, we only add an edge from the right to the left rectangle, if the overlap 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; see Figure 10a.

As the final step of the construction of $\mathcal{D}$, we add two nodes to $\mathcal{D}$ which we denote by $\overline{s}$ and $\overline{t}$. We connect $\overline{s}$ to all rectangles on the top of the columns and connect all rectangles on the bottom of the columns with $\overline{t}$.

Obviously, $\mathcal{D}$ is a planar graph, i. e., it has at most $\mathcal{O}(|\widehat{A}| \cdot b)$ edges. Also, the number of rectangle pairs we need to consider to find all edges is small. In fact this number is in $\mathcal{O}(|\widehat{A}| \cdot b)$ since by construction edges are either completely contained in a column or connect nodes in two neighboring columns. Since we


(a) Overlap of both boxes is smaller than $s_{\text{edge}}^{\text{edge}}$.

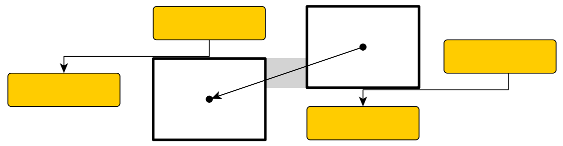
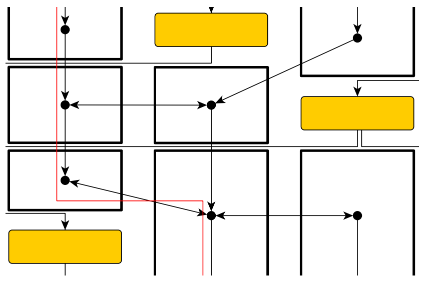

(b) Overlap of both boxes is larger than $s_{\text{edge}}^{\text{edge}}$.

Figure 10: Edges between overlapping boxes of neighboring columns.


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

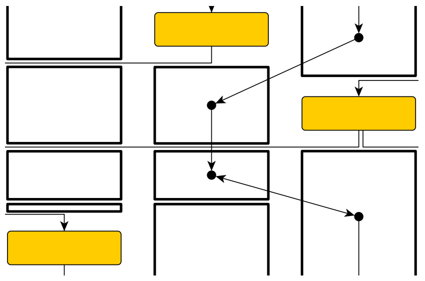

(b) The modified network after the compaction along p.

Figure 11: The compaction network $\mathcal{D}$ and how it is modified.

have sorted the nodes of $\mathcal{D}$ already in a previous step, we find all edges of $\mathcal{D}$ in $\mathcal{O}(|\widehat{A}| \cdot b)$ time with a simple sweep-line algorithm from top to bottom. In Figure 11a we depict a clipping of a layout and the corresponding compaction network $\mathcal{D}$.

We use the compaction network $\mathcal{D}$ in order to find compaction paths using a right-first depth-first search starting at $\overline{s}$. Since we need to ensure that the computed compaction path is y-monotone, we always keep track of the current y-coordinate. We are only allowed to move to a horizontally neighboring rectangle if this requires no upward movement of the compaction path. If the search reaches $\overline{t}$, we have 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 11):

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 11b 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.

Theorem 7 *The width compaction algorithm can be implemented to run in* $\mathcal{O}(|\widehat{A}| \cdot b^2 \cdot (\log(|\widehat{V}|) + b))$ *time and* $\mathcal{O}(|\widehat{A}| \cdot b^2)$ *space.*

Proof: 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. We now estimate the number of rectangles we need to create at most.

Note that a compaction path cannot split a rectangle at any point, but rather it can only split it at a y-coordinate at which a compaction path leaves a column and enters a new one. These y-coordinates are the y-coordinates of the upper and lower boundaries of nodes in $\mathcal{D}$. This insight allows us to find an upper bound on the number of nodes the compaction network can have. As we have argued, initially the number of nodes in $\mathcal{D}$ is in $\mathcal{O}(|\widehat{A}| \cdot b)$. To obtain a worst-case upper bound for the number of new nodes, assume we split all rectangles at the y-coordinates of the upper and lower boundary of a single node. This can create at most two additional rectangles per column which means $\mathcal{O}(b)$ additional rectangles in total. When we repeat this for all of the original nodes in $\mathcal{D}$ this leads to a total of $\mathcal{O}(|\widehat{A}| \cdot b^2)$ nodes in $\mathcal{D}$. Note that we do not need to do this for the newly created nodes, because the y-coordinate of either their upper or lower boundary was already used to cut other nodes. Since $\mathcal{D}$ remains planar, the number of edges is also in $\mathcal{O}(|\widehat{A}| \cdot b^2)$. In order to analyze 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 graph $\mathcal{D}$. Thus, it runs in $\mathcal{O}(|\widehat{A}| \cdot b^2)$ time.

When all compaction paths are found, we need to compact along them. Compacting along one path can easily be performed in $\mathcal{O}(|\widehat{V}| + |\widehat{A}| \cdot b^2) = \mathcal{O}(|\widehat{A}| \cdot b^2)$ time. Thus, compacting along all paths needs $\mathcal{O}(|\widehat{A}| \cdot b^3)$ time.

Summing up the running time of the initial construction of $\mathcal{D}$, the computation of the compaction paths and the compaction itself, we get $\mathcal{O}(|\widehat{A}| \cdot b \cdot (\log(|\widehat{A}| \cdot b) + b^2))$

as total runtime. Since we deal with simple graphs, $|\widehat{A}| \leq |\widehat{V}|^2$ holds. Thus, we can further simplify the time complexity to $\mathcal{O}(|\widehat{A}| \cdot b^2 \cdot (\log(|\widehat{V}|) + b))$.

We need $\mathcal{O}(|\widehat{V}| + |\widehat{A}|)$ space to store the column assignment, but this term is dominated by the potential size of $\mathcal{D}$ which is in $\mathcal{O}(|\widehat{A}| \cdot b^2)$. □

Note that in the case that the node sizes do not differ too much, and should all of them be integer, the time complexity of the algorithm can be reduced to $\mathcal{O}(|\widehat{A}| \cdot b^3)$ by employing a linear-time sorting algorithm for this case.

We have now all results required to prove our main result which we stated in Theorem 1, and which we repeat here for the reader's convenience. The proof follows by combining the main theorems of each section, namely, Theorems 3, 4, 6, and 7. The running time is dominated by the first step of our approach (see Theorem 3), and the space consumption is dominated by the last step (see Theorem 7).

Theorem 1. *For a given connected directed acyclic graph $G = (V, A)$ a locally-symmetric, column-based layout with at most four bends per edge can be computed in $\mathcal{O}(|A|^3|V|)$ time and $\mathcal{O}(|A||V|^2)$ space.*

6 Evaluation

The drawing style used here stems from a particular application for drawings of so-called 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. They originate from argumentation theory but are used in many more fields like philosophy and politics. For detailed background information about argument maps we refer the reader to [2, 30]. As we mentioned at the beginning of Section 3, for layouts of argument maps, we have the additional constraints of *free sources and sinks.* Recall that, by this, we denote the property that above a source and below a sink no other box is positioned. We can enforce this constraint by not removing the edges incident to $\hat{s}$ and $\hat{t}$ before the width compaction. Then, these avoid that another box is shifted above a source or below a sink. Further, argument maps may contain cycles (although they rarely do). In a preprocessing step we remove cycles by reversing the direction of edges in G such that it becomes cycle-free. Since computing a cardinality minimal set of such edges is $\mathcal{NP}$-complete [28], we use a well-known heuristic by Eades et al. [16] to solve this problem.

In the last step, when we remove the nodes $\hat{s}$ and $\hat{t}$, we need to correct the direction of those edges we have reversed in the preprocessing step.

As already mentioned in Section 3, we randomize the computation of the topology. We do this at two points: (i) for the construction of the feasible upward planar subgraph and (ii) we randomize the order in which the deleted edges are reinserted. Each of these two steps is performed ten times, i. e., we execute the computation of a feasible upward planar subgraph 100 times in total. Over all runs we take the crossing-minimal upward planar representation as the result.

The basis of this evaluation is a set of 51 argument maps differing in size, purpose of usage and experience of the creator (uploaded to http://www.graph-archive.org). For a more detailed analysis of these input graphs we refer to [13]. First, we present statistics about measurable quality criteria, and then discuss the aesthetic qualities of these layouts.

Implementation Details. We implemented our algorithms using C++ relying on the Open Graph Drawing Framework)[1]. For the topology step we used the OGDF implementation of the layer-free upward crossing minimization algorithm, as well as several data structures included in the OGDF framework (e. g., Graph, GraphAttributes, EdgeArray, NodeArray). As already mentioned we use a heuristic [16] to compute a cardinality-minimal set of edges whose directions we can reverse to obtain a cycle-free graph. We use the implementation of this heuristic that has been integrated into the OGDF framework.

We compiled our C++ implementation with GCC 4.7.1, using optimization level 3. All experiments were performed on a single core of an Intel Xeon E5-2670 processor that is clocked at 2.66 GHz. The machine is running Linux 3.4.28-2.20 and has 64 GiB of RAM.

Statistics. We start by analyzing the number of crossings in the final layout as well as in the upward planar representation $(\mathcal{U}, \Gamma)$. Although it is theoretically possible that the final layout contains fewer crossings than $(\mathcal{U}, \Gamma)$, for all instances there are at least as many crossing in the final layout as in the upward planar representation $(\mathcal{U}, \Gamma)$. However, if $(\mathcal{U}, \Gamma)$ is free of crossings, then the final layout is planar as well (compare to Lemma 1).

We note that, in our experiments, on average there are 69% more crossings in the final layouts than in the upward planar representations. Although this number may seem high, this is only because most of our input drawings have little or even no crossings. Then, even a slight increase in the number of crossings yields a high relative increase of crossings.

Due to the shape algorithm there can be at most four bends per edge. Except for three bend-free layouts, the average number of bends per edge is between 0.4 and 1.5. Only 13 of 51 layouts have edges with four bends. Taking the average over all layouts, we have 1.06 bends per edge.

Finally, we turn towards the running time analysis. The maximum running time for the 51 input instances is 10.25s for an instance having 134 nodes and 158 edges, whereas the average is 0.32s. The average running time is strongly influenced by the few large instances. For instances with at most 30 nodes, the running time is 0.02s on average while the maximum running time is 0.19s.

It is interesting to note the contribution of the three phases topology, shape and metrics to the overall running time. Shape and metrics make only insignificant contributions of less than 0.01% and 0.2%, respectively, whereas the topology step needs 99.6% of the overall running time. Thus, the topology step is the bottleneck.

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

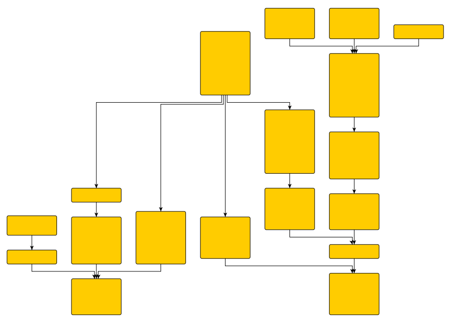


Figure 12: Instance D-2.5.

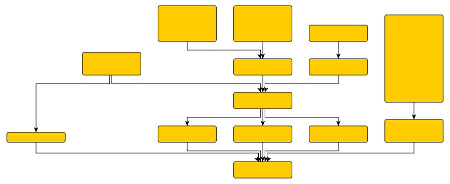


Figure 13: Instance D-5.3.

We repeated our experiments with a decreased number of randomized runs in the topology step from 100 to 9. The maximum and average running time for the topology step decreased from 10.25s to 0.98s and from 0.32s to 0.03s, respectively. This decrease to roughly 1/10 in running time is not surprising as it simply reflects the fact that the topology step requires nearly 100% of the time for computing the layouts. Nevertheless, this is a significant improvement in running time while the average number of bends per edge increases only slightly from 1.05 to 1.07, and the average number of crossings increases from 2.64 to 2.76 for our instances.

Case Studies. Here, we present three layouts that are created by the algorithmic approach described in Section 2–5. Using these layouts as examples, we discuss the benefits and the disadvantages of our algorithms. Figure 12 shows a layout for which we have no suggestions for improvement. The layout is free of crossings, the number of bends are close to the minimum, it is compact and locally symmetric. Beside the hard facts, it is well structured and gives a good impression of the internal structure. The other two layouts, while looking good in general, have minor drawbacks, which we discuss in the following.

In Figure 13 on the left side there is a configuration that is similar to a bow, which we treated in Section 5. However, this time, we have a bow containing a box. We could expand our concept to paths in $\widehat{G}$ that contain only nodes whose in-degree and out-degree equals one. Such a scenario is not treated in our algorithm. Since there can be many different special cases which might be resolved easily, we propose to investigate and integrate additional algorithms for optimizing orthogonal drawings into our approach.

In Figure 14 there is a set of boxes on the right side of the layout that is positioned quite high. It seems as it might be possible to 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, 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 Section 5. Besides

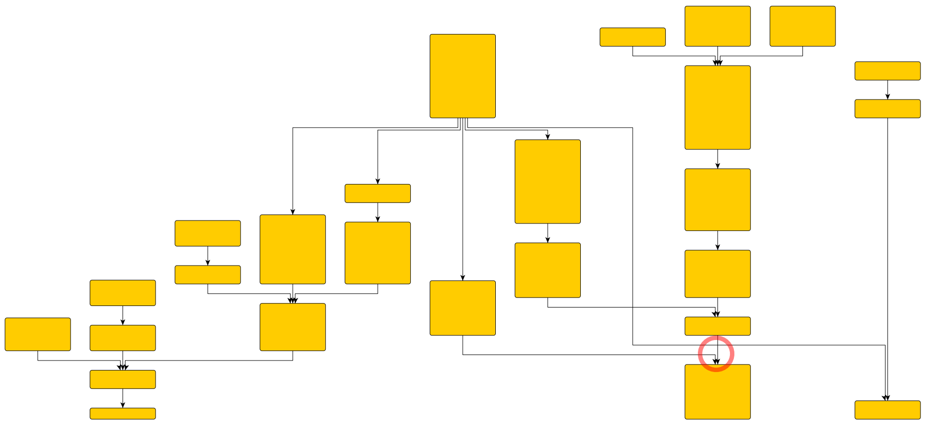


Figure 14: Instance D-2.7. Circle added to highlight the effect of different edge-edge spacings.

this drawback, the example gives a good impression of the edge bundling enabled by the minimum edge-edge spacings. The effect the two different edge-edge spacing constraints have on the final layout can be seen inside the red circle. The two edges that have the same target have a small distance between them while the third horizontal edge-segment belonging to the third edge is placed at a greater distance to the other horizontal edge-segment.

Overall, we conclude that the computed layouts are of high quality from an aesthetic point of view as well as regarding the objective quality measures.

7 Conclusion

Our main result is an algorithmic approach to generate column-based graph layouts for directed acyclic graphs. Our approach integrates the layer-free upward crossing minimization technique by Chimani et al. [9] into the well-known topology-shape-metrics framework. We showed that several of the arising subproblems are $\mathcal{NP}$-complete and devised efficient algorithms for all steps, many with provable quality guarantees. In the first step we minimize the number of crossings as well as the total source/sink distance using layer-free upward crossing minimization. Afterwards, we use a modified graph drawing heuristic by Biedl and Kant [4] such that there are at most four bends per edge. Thereby, nodes and edges are assigned to columns. In the last step, we first minimize the vertical edge length using a greedy approach and then the horizontal edge length based on iteratively removing rightmost compaction paths. During the edge length minimization 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 focus of the algorithm is shifted away from minimizing the number of edge crossings. We fix the edge crossings again when minimizing the vertical edge length. In usual applications of the topology-shape-metrics framework crossing minimization is the first aspect that is considered and all remaining optimization criteria are treated afterwards and therefore must handle the choices made in the first step. This is not the case for our approach, where we increase the significance of bend and total edge length minimization.

The layouts we compute are of high quality. They have a clear, well-structured look and are compact. Input instances that have a typical size for argument maps can be layouted quickly. Thus, our algorithm is feasible for practical purposes. In fact, the algorithm has, in the meantime, been integrated into the tool argunet[2], which is used for creating and manipulating argument maps, and is now available as a plugin.

Although the generated layouts are of high quality, certain unappealing features remain and as a next step it may be worthwhile to explore if it is possible to adapt algorithms for optimizing orthogonal drawings layouts; e. g., the 4M algorithm [20]. Coming from the practical application another interesting step is to investigate how to handle dynamic argument maps, i. e., generate drawings of argument maps that change over time while maintaining the user's mental map.

Acknowledgments. Andreas Gemsa is financially supported by the *Concept for the Future* of KIT within the framework of the German Excellence Initiative.

[2]`http://www.argunet.org`, argunet is open source and published under the GPL

References

[1] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, 27:132–169, 1998. `doi:10.1137/S0097539794279626`.

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

[3] G. Betz, C. Doll, A. Gemsa, I. Rutter, and D. Wagner. Column-based graph layouts. In *Graph Drawing*, volume 7704 of *Lecture Notes in Compute Science*, pages 236–247. Springer, 2013. `doi:10.1007/978-3-642-36763-2_21`.

[4] T. Biedl and G. Kant. A better heuristic for orthogonal graph drawings. *Computational Geometry*, 9(3):159–180, 1998. `doi:10.1016/S0925-7721(97)00026-6`.

[5] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973. `doi:10.1016/S0022-0000(73)80033-9`.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[7] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2002. `doi:10.1007/3-540-45848-4_3`.

[8] D. Cartwright and K. Atkinson. Using computational argumentation to support e-participation. *Intelligent Systems, IEEE*, 24(5):42–52, 2009. `doi:10.1109/MIS.2009.104`.

[9] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Layer-free upward crossing minimization. *Journal of Experimental Algorithmics*, 15, 2010. `doi:10.1145/1671970.1671975`.

[10] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Upward planarization layout. *Journal of Graph Algorithms and Applications*, 15(1):127–155, 2011. `doi:10.7155/jgaa.00220`.

[11] G. Di Battista, W. Didimo, M. Patrignani, and M. Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size. In *Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 297–310. Springer, 1999. `doi:10.1007/3-540-46648-7_31`.

[12] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994. `doi:10.1016/0925-7721(94)00014-X`.

[13] C. Doll. Automatic layout generation for argument maps. Master's thesis, Karlsruhe Institute of Technology, February 2012.

[14] T. Dwyer, K. Marriott, and M. Wybrow. Interactive, constraint-based layout of engineering diagrams. *Electronic Communications of the EASST*, 13, 2008. URL: `http://journal.ub.tu-berlin.de/eceasst/article/view/168/163`.

[15] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[16] P. Eades, X. Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47:319–323, 1993. `doi:10.1016/0020-0190(93)90079-O`.

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

[18] H. Eichelberger. Aesthetics of class diagrams. In *Visualizing Software for Understanding and Analysis, 2002*, pages 23–31, 2002. `doi:10.1109/VISSOF.2002.1019791`.

[19] M. Eiglsperger, C. Gutwenger, M. Kaufmann, J. Kupke, M. Jünger, S. Leipert, K. Klein, P. Mutzel, and M. Siebenhaller. Automatic layout of UML class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004. `doi:10.1057/palgrave.ivs.9500078`.

[20] U. Fößmeier, C. Heß, and M. Kaufmann. On improving orthogonal drawings: The 4M-algorithm. In *Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 125–137. Springer, 1998. `doi:10.1007/3-540-37623-2_10`.

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

[22] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *Siam Journal On Algebraic And Discrete Methods*, 4(3):312–316, 1983. `doi:10.1137/0604033`.

[23] A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In *Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 1997. `doi:10.1007/3-540-62495-3_49`.

[24] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM Journal on Computing*, 31(2):601–625, 2001. `doi:10.1137/S0097539794277123`.

[25] W. He and K. Marriott. Constrained graph layout. In *Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 1997. doi:10.1007/3-540-62495-3_50.

[26] W. Huang, S.-H. Hong, and P. Eades. Effects of crossing angles. In *Visualization Symposium, 2008. PacificVIS '08. IEEE Pacific*, pages 41–46, 2008. doi:10.1109/PACIFICVIS.2008.4475457.

[27] M. Jünger and P. Mutzel. *2-layer straightline crossing minimization: Performance of exact and heuristic algorithms.* MPI Informatik, Bibliothek & Dokumentation, 1996.

[28] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. doi:10.1007/978-1-4684-2001-2_9.

[29] R. Marti and M. Laguna. Heuristics and meta-heuristics for 2-layer straight line crossing minimization. *Discrete Applied Mathematics*, 127(3):665–678, 2003. doi:10.1016/S0166-218X(02)00397-9.

[30] D. C. Schneider, C. Voigt, and G. Betz. Argunet—a software tool for collaborative argumentation analysis and research. In *7th Workshop on Computational Models of Natural Argument (CMNA VII)*, 2007.

[31] F. Schreiber, T. Dwyer, K. Marriott, and M. Wybrow. A generic algorithm for layout of biological networks. *BMC Bioinformatics*, 10(1):375, 2009. doi:10.1186/1471-2105-10-375.

[32] J. Seemann. Extending the sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 415–424. Springer, 1997. doi:10.1007/3-540-63938-1_86.

[33] J. M. Six, K. G. Kakoulis, and I. G. Tollis. Refinement of orthogonal graph drawings. In *Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 302–315. Springer, 1998. doi:10.1007/3-540-37623-2_23.

[34] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981. doi:10.1109/TSMC.1981.4308636.

[35] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16:421–444, 1987. doi:10.1137/0216030.