INSTITUT FÜR INFORMATIK

Using One-Dimensional Compaction for Smaller Graph Drawings

Ulf Rüegg, Christoph Daniel Schulze,
Daniel Grevismühl, and Reinhard von Hanxleden

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Department of Computer Science
Kiel University
Olshausenstr. 40
24098 Kiel, Germany

Using One-Dimensional Compaction
for Smaller Graph Drawings

Ulf Rüegg, Christoph Daniel Schulze,
Daniel Grevismühl, and Reinhard von Hanxleden

E-mail: {uru,cds,dag,rvh}@informatik.uni-kiel.de

Abstract

We review the technique of one-dimensional compaction and use it as part of two new methods tackling problems in the context of automatic diagram layout: First, a post-processing of the layer-based layout algorithm, also known as Sugiyama layout, and second a placement algorithm for connected components with external extensions.

We apply our methods to data flow diagrams from practical applications and find that the first method significantly reduces the width of left-to-right drawn diagrams. The second method allows to properly arrange disconnected graphs that have hierarchy-crossing edges.

Keywords: one-dimensional compaction, diagram layout, layer-based layout, Sugiyama layout, disconnected graphs, dataflow diagrams

Contents

1 Introduction

Automatically drawing graph-based visual models has gained more and more acceptance over the past years, with industrial tools starting to incorporate automatic layout facilities, be it semi-automatic or fully-automatic, to support model-driven engineering or interactive browsing of models [5]. Example tools are *LabVIEW* (National Instruments), *EHANDBOOK* (ETAS), *Simulink* (The MathWorks, Inc.), and *Ptolemy* (UC Berkeley).

For applications where *hierarchical data flow diagrams* are used, the layout techniques have continuously been improved to handle most of the peculiarities of this type of diagram [19]. Still, further improvements are required regarding the compactness of the resulting drawings [10]. In this paper we show how the simple technique of *one-dimensional compaction* can be used to significantly improve the compactness of data flow diagrams drawn with state-of-the-art methods (see Figure 1.1 and Figure 1.2 for results). While we motivate our contributions from the perspective of data flow diagrams, they are not restricted to this type of diagram. The presented methods are implemented as part of the KIELER open-source project[1] and will make their way into the Eclipse Layout Kernel (ELK)[2].

Outline. We start by reviewing one-dimensional compaction in Section 1.1. Chapter 2 and Chapter 3 introduce our contributions and their evaluations. We conclude in Chapter 4.
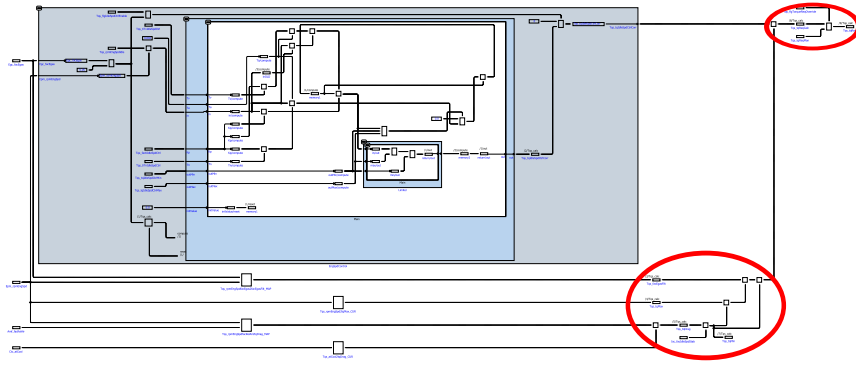
1.1 One-Dimensional Compaction

One-dimensional compaction is a well-known technique to minimize the area occupied by a set of objects in the plane. As opposed to the NP-hard two-dimensional compaction problem, it can be solved efficiently in time $O(n \log n)$, n being the number of objects [14]. Lengauer thoroughly discusses one-dimensional compaction in the context of VLSI-design and presents methods for several, quite general, variations [14]. In the following, we present the concepts that are relevant for the remainder of this paper and refer the reader to the book for further details.

Let $\mathcal{R}$ denote a set of rectangles in $\mathbb{R}^2$. A rectangle $r = (r_x, r_y, r_w, r_h)$ is a quadruple of the rectangle's top-left position $(r_x, r_y) \in \mathbb{R}^2$ and its size $(r_w, r_h) \in \mathbb{R}^{+2}$. Let the union $(\cup)$ of a pair of rectangles be defined as the set of points $(a, b) \in \mathbb{R}^2$ that are covered by either of the rectangles (hence the resulting set does not necessarily describe

---

[1]`http://rtsys.informatik.uni-kiel.de/kieler`
[2]`http://www.eclipse.org/elk`

(a)

(b)

Figure 1.1. Illustration of our first contribution. (a) An automatically drawn data flow diagram with the layer-based layo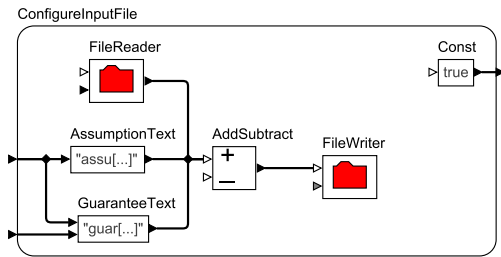ut methods by Schulze et al. [19]. Circled nodes are pushed to the right due to the method's nature. (b) The same diagram after our post-processing. The diagram's width is reduced by about 16% and the average edge length is reduced by over 50%.

(a)

(b)

Figure 1.2. Illustration of our second contribution. Placing a diagram's sub-graphs must assert that no *external edge* (an edge connected to the outer boundary) crosses a subgraph. In the example the Const node must not be placed to the left of the other nodes because its external edge connects to the right border of the compound node ConfigureInputFile. (a) A feasible placement. (b) The placement optimized with one-dimensional compaction.

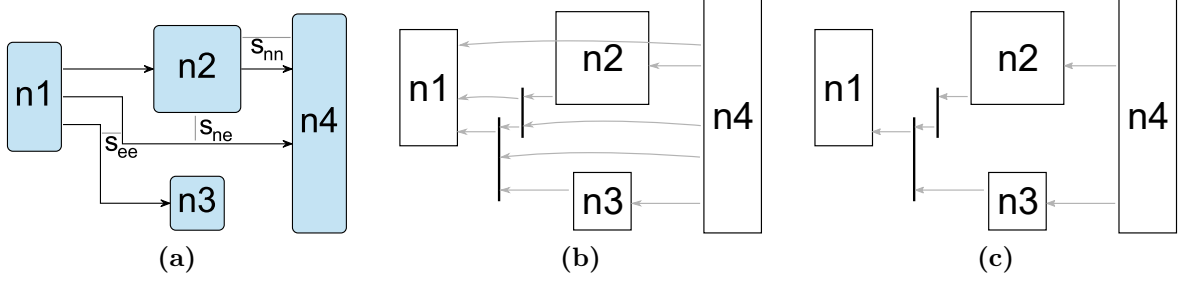**Figure 1.3.** A diagram with its corresponding constraint graph. (a) shows the diagram and illustrates different spacing values. Edges' vertical segments are converted into rectangles. The constraint graph in (b) contains, as opposed to (c), redundant constraints.

a rectangle); let the intersection ($\cap$) be the set of points that are covered by both of them. Similar to Lengauer, we say r is *left of* s, $r, s \in \mathcal{R}$ iff $(r_x + r_w < s_x)$ and say that r and s *overlap horizontally* ($r \prec s$) iff

$$(r_y < s_y + s_h) \ \wedge \ (s_y < r_y + r_h) \ \wedge \ r \text{ left of } s.$$

We consider a set of rectangles $\mathcal{R}$ to be *valid* if $\forall r \neq s \in \mathcal{R} : r \cap s = \emptyset$.

The one-dimensional compaction problem in the x-dimension seeks for a transformation of a valid set of rectangles $\mathcal{R}$ into a valid set of rectangles $\mathcal{R}'$ by changing x-coordinates only and preserving the order $\prec$ such that the width w is minimized, with $w = |\ \max_{r' \in \mathcal{R}'}\ (r'_x + r'_w)\ -\ \min_{r' \in \mathcal{R}'}\ r'_x\ |$. Note that the case to compact the height can be defined analogously.

To solve this problem two steps are executed. First, a *constraint graph* is derived from the given placement that encodes which rectangles horizontally overlap. Second, the graph is used to position the rectangles in a way that yields minimum width.

1.1.1 Constraint Graph

In the constraint graph $CG = (\mathcal{R}, C)$ a valid set of rectangles $\mathcal{R}$ represents the nodes. The constraints $C \subseteq \mathcal{R} \times \mathcal{R}$ form the directed edges of the graph. An edge $c = (s, r)$ with $s, r \in \mathcal{R}$ is added to CG iff $r \prec s$. Note that this graph is acyclic by construction.

Naively, the constraint graph can be constructed in $O(n^2)$ time by checking for every pair of rectangles if they overlap horizontally, n being the number of rectangles. However, the number of edges is $O(n^2)$ and several edges may be redundant. Consider Figure 1.3 where the constraint between rectangles n1 and n4 is transitively guaranteed by the two constraints (n1, n2) and (n2, n4). Figure 1.3c shows a constraint graph without redundant constraints.

Lengauer shows how, for a given set of rectangles, the constraint graph can be calculated using a scanline technique in time $O(n \log n)$ and with $O(n)$ edges [14]. The procedure is illustrated in Algorithm 1. Let $Y^- = \{r_y : r \in \mathcal{R}\}$ denote the set of the upper coordinates of all rectangles and let $Y^+ = \{r_y + r_h : r \in \mathcal{R}\}$ denote the set of

---

Algorithm 1. Constraint graph

Input: $\mathcal{R}$: set of rectangles
Data: cand[r]: constraint candidates indexed by rectangle, S: sorted set of rectangles
Output: C: set of constraints

1 points $\leftarrow Y^- \cup Y^+$
2 sort points ascendingly (prioritizing Y^+)
3 **for** p in points **do**
4   **if** $p \in Y^-$ **then**
5   r $\leftarrow$ r(p)
6   put r into S
7   cand[r] $\leftarrow$ S.left(r)
8   cand[S.right(r)] $\leftarrow$ r
9   **else**
10   **if** S.left(r) exists $\wedge$ S.left(r) = cand[r] **then**
11   add (r, S.left(r)) to C
12   **if** cand[S.right(r)] = r **then**
13   add (S.right(r), r) to C
14   remove r from S

---

lower coordinates. For a point $p \in Y^- \cup Y^+$, let $r(p)$ denote the rectangle for which p was added. The scanline processes the points in increasing order from top-to-bottom. An array cand is used to hold constraint candidates. The set S, ordered based on the rectangles' x-coordinates, allows to query the (current) predecessor (left(r)) and successor (right(r)) of a rectangle r according to the order. Its implementation must provide insert and delete operations that run in $O(\log n)$ time and constant time operations to access neighbor elements. When the scanline encounters an upper coordinate, it adds the corresponding rectangle to S and updates the constraint candidates. When a lower coordinate is encountered, the scanline "finishes" the corresponding rectangle by removing it from S and by checking the set of candidate constraints, possibly adding them to the constraints of the final graph. For the correctness of this procedure and minimality of the resulting constraint graph, see Lengauer [14].

1.1.2 Grouping

In certain use cases it is required that two or more rectangles keep their relative positioning to each other. While Lengauer calls them *grouping constraints*, we refer to them as *groups* and use a slightly more restricted definition. Formally, we extend a constraint graph $CG = (\mathcal{R}, C)$ to a *grouped constraint graph* $GCG = (\mathcal{R}, C, G)$ where every rectangle is part of a group. A group $g \in G$ is a non-empty set of rectangles $g \subseteq \mathcal{R}$ with their offset r_δ to an imaginary origin (g_x, g_y). We assume that the left-most rectangle of a group has an offset of zero. During compaction the relative positions between grouped rectangles are to be preserved by the algorithm. A grouping is *valid* if no rectangle is in more than one group: $\forall g_i \neq g_j \in G : g_i \cap g_j = \emptyset$. Any constraint $c = (r, s) \in C$ can be neglected if r and s share the same group. Let $g(r), r \in \mathcal{R}$, stand for r's group. $out(g)$ denotes the outgoing constraints of g, i.e. $\{(r, s) \in C : r, s \in \mathcal{R} \wedge r \in g\}$, and $in(g)$ denotes the incoming constraints. $\delta^+(g) = |out(g)|$ denotes the *out-degree* of group g, i.e. the number of constraints leaving g, $\delta^-(g) = |in(g)|$ denotes the *in-degree*.

Algorithm 2. Compact

Input: $GCG = (\mathcal{R}, C, G)$: grouped constraint graph

1   $g_x \leftarrow 0 \quad \forall g \in G$
2   sinks $\leftarrow \{\ g \in G : \delta^-(g) = 0\ \}$
3   **while** sinks not empty **do**
4   g $\leftarrow$ sinks poll
5   **for** $r \in g$ **do**
6         $r_x = g_x + r_\delta$
7   **for** $(s, r) \in \{(s, r) \in C : r \in g\}$ **do**
8         $g(s)_x = \max(g(s)_x, r_x + r_w)$
9   remove (s, r) from C
10   **if** $\delta^-(g(s)) = 0$ **then**
11   sinks add $g(s)$

1.1.3 Compaction

The minimum width of the grouped constraint graph GCG is bound by its longest path. To obtain a placement with minimum width we execute Algorithm 2. The method iteratively assigns a position to sinks of the graph, initially setting all positions to zero ($g_x = 0$). After all rectangles of g have been placed, all incoming edges $in(g)$ are removed from the graph and the potential position of a constrained group g' is set to $\max(g'_x, \max_{r \in g}(r_x + r_w))$. The procedure is repeated until all nodes have been placed, which takes linear time. It is guaranteed to terminate since the graph is acyclic.

2 Layer-Based Drawings

In 1981, Sugiyama et al. described the structure of a successful methodology to draw directed graphs in the plane [20]. It is known under various names such as Sugiyama-style layout, hierarchical layout, and layer-based layout. The approach and most of the related research is summarized in a recent book chapter by Healy and Nikolov [12]. Essentially, it consists of five consecutive phases: (1) *cycle breaking* makes cyclic graphs acyclic by reversing edges, (2) *layering* assigns nodes to indexed layers such that edges always connect layers of lower index to higher index, (3) *crossing minimization* aims at reducing the number of edge crossings, (4) *node coordinate assignment* determines explicit y-coordinates for nodes, and (5) *edge routing* determines paths for edges and assigns x-coordinates to nodes.

Most literature in this context assumes that nodes are of the same size. However, this is not the case with most practical applications, and it has been observed that the compactness of diagrams suffers in the presence of significant size differences [6, 10]. Existing methods to tackle this issue either result in unpleasant drawings or increase the complexity of subsequent steps of the approach [15, 16, 6]. A common idea is to assign large nodes to multiple layers, for instance, by splitting them into multiple small chunks. The crossing minimization phase then has to keep edges from crossing nodes, and the node coordinate assignment has to assert that all chunks receive the same y-coordinate.

Nonetheless, the problem becomes more and more imminent with diagram exploring approaches where nodes sizes may differ by factors of 10 or even 100 [5, 10].

Recently, Schulze et al. presented several extensions to the layer-based approach to handle data flow diagrams with *ports* (explicit attachment point of edges on a node's perimeter) and *orthogonally* routed edges [19]. See Figure 2.1 for an example. Working with these kind of diagrams, we observed that scenarios where wide nodes (Consumer) prevent more compact placements are quite common. The problem here is that the layer-based approach assigns nodes rigidly to layers, marked by dashed lines in Figure 2.1, and no pair of connected nodes may be placed in the same layer, thus pushing the Dropped node to the right. Here, one-dimensional compaction allows to reduce the diagram's overall width by breaking the rigid layering and pushing everything as far as possible to the left. Vertical segments of orthogonally routed edges may be regarded as rectangles with either zero or very small width. Since the compaction procedure can be applied to the final drawing, after the traditional layer-based approach has finished completely, no additional complexity is added to any of the layer-based phases.

In the remainder of this section, we show how to convert a drawing of a graph into a one-dimensional compaction problem to significantly reduce the drawing's width and, in particular, show how to address the peculiarities of data flow diagrams. An example was already shown in Figure 1.1.
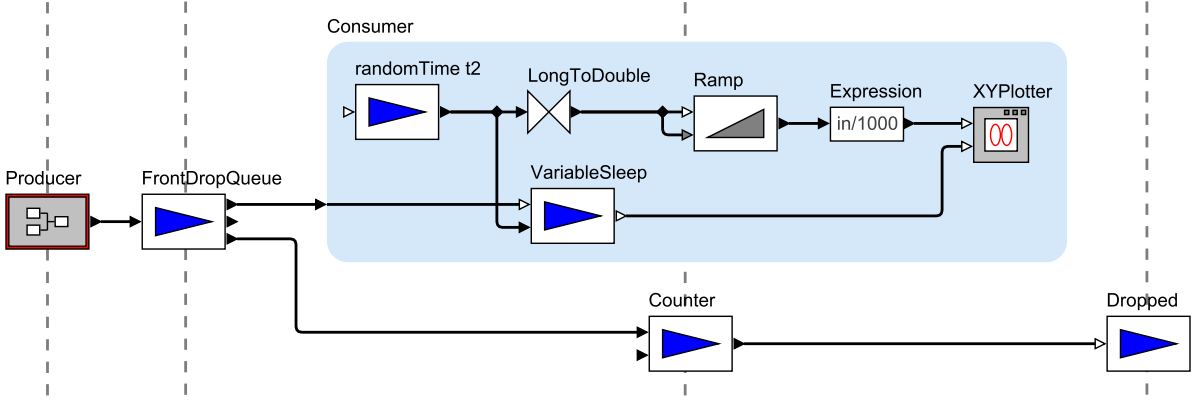
Figure 2.1. Drawing of a data flow diagram with ports, produced with the layer-based approach. The vertical dashed lines illustrate the created layers for the top-level graph. The layering of the nodes contained in the Consumer node is not shown here.

2.1 Width Reduction

Diagrams as the one seen in Figure 2.1 can be formalized as *directed hypergraphs*, which are pairs $HG = (V, H)$. V is a set of nodes and $H \subseteq (P(V) \times P(V))$ a set of hyperedges that are connected to nodes via one of the nodes' *ports*. Schulze et al. represent each hyperedge $h = (S, T) \in H$ by a set of edges, i. e. for every pair $s \in S$ and $t \in T$ a directed edge $e = (s, t)$ is introduced. This allows to use known layer-based methods without the requirement to specifically address hyperedges. Both nodes and edges can carry labels that contribute to their bounding boxes. Additionally, a drawing must adhere to certain spacings between nodes and edges. Figure 1.3 illustrates the three spacings s_{nn}, s_{ne}, and s_{ee}.

After applying the layer-based approach with the extensions by Schulze et al. [19] in conjunction with any orthogonal edge routing technique, e. g. the one by Sander [18], we can make some assumptions: (1) nodes and edges don't overlap; (2) prescribed spacings are satisfied; (3) the connection point of an edge on the corresponding node's perimeter is fixed. These assumptions allow to apply one-dimensional compaction to HG by transforming it into a set of rectangles $\mathcal{R}$: We add the bounding box of every node $v \in V$ as a rectangle to $\mathcal{R}$. Furthermore for every vertical segment of an edge $e \in H$, we add a rectangle with corresponding height and unit width to $\mathcal{R}$. To guarantee enough room for edge labels they can be added to the set of rectangles as well. Nevertheless, to get satisfying results, there are several subtleties to be addressed which we discuss in the next paragraphs.

2.1.1 Spacing

When it comes to spacings a layout algorithm has to leave between a diagram's elements, different scenarios are possible. We discuss three: (1) A single global spacing value s_g
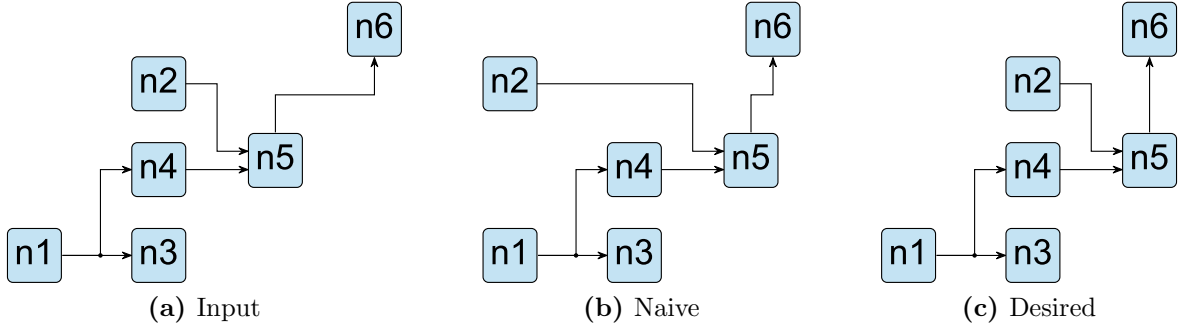
Figure 2.2. Example of applying one-dimensional compaction to an input graph (a) with different objectives of minimal width and minimal edge length in (b) and (c).

is used to separate any pair of elements from each other. (2) Every element has an individual spacing value s_e, possibly a different one for each side. (3) Several global spacing values are to be preserved between pairs of elements depending on their types. For instance, Figure 1.3a illustrates three spacings s_{nn}, s_{ne}, and s_{ee} between pairs of nodes, pairs of a node and an edge, and pairs of edges.

Scenario (1): The global spacing s_g can straight-forwardly be preserved during compaction by adding $\frac{s_g}{2}$ to either side of the rectangles that represent each element before the constraint graph is computed.

Scenario (2): As every element can specify its own spacing value s_e, only adding $\frac{s_e}{2}$ may result in spacing violations. One would thus have to add s_e to every side. However, this may result in an invalid input to the scanline algorithm. If spacing violations are not acceptable, a possible solution is to re-create a valid input using a dedicated overlap removal algorithm [2]. The algorithm would have to support groupings.

Scenario (3): Similar to the previous scenario, it is not always possible to guarantee valid spacings by enlarging the rectangles. Say $s_{ne} < \frac{s_{nn}}{2}$. Extending every node representing rectangle using $\frac{s_{nn}}{2}$ may result in an overlap between a node representing rectangle and a rectangle that represents a vertical segment. Again, this is an invalid input for the scanline algorithm.

To resolve this, we incrementally build a constraint graph by executing the scanline algorithm multiple times. This may result in more constraints than absolutely necessary. Each time a different subset of the overall set of rectangles is considered and each time the rectangles are enlarged by a different value. Consequently, a rectangle has no unique value by which it is enlarged, which is why we operate the subsequent compaction step on the original-sized rectangles. For this to be feasible, however, we associate an individual "length" with every constraint which corresponds to the minimum separation between the two involved elements and therefore re-assembles the enlarging of rectangles. Now, when executing the compaction step, the position of an element is determined by the current position of an outgoing constraint's target and the determined "length".

We run the scanline algorithm three times: First, only node representing rectangles are considered and enlarged by $\frac{s_{nn}}{2}$. Second, edge representing rectangles are enlarged
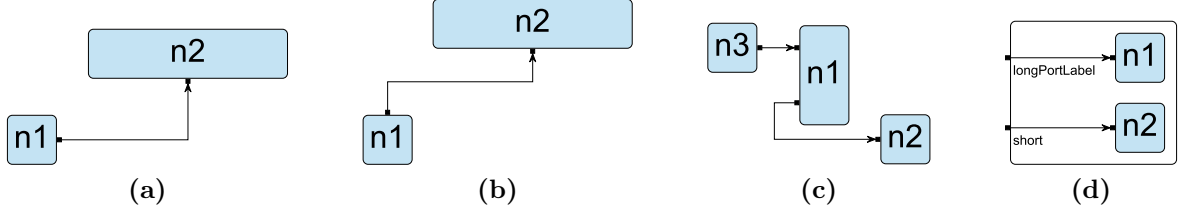
Figure 2.3. When aiming for a compact layout with short edges some special cases have to be considered. In (a) and (b) node n1 should be allowed to be placed below n2. In (c) the vertical segment of the edge (n1, n2) should stay close to n1. In (d) external ports' labels of different lengths reveal compaction potential; n2 can be moved further to the left.

by $\frac{s_{ee}}{2}$ and constraints are generated solely between pairs of edges. These two passes guarantee valid spacings between pairs of nodes and pairs of edge segments and a valid input to the scanline algorithm. The third pass is executed using all rectangles. This time the selection of the enlarging value is more intricate and we distinguish three cases based on the value of s_{ne}. Either of the three cases results in a valid input to the scanline algorithm by carefully enlarging rectangles such that no overlaps are introduced. Note that the first two cases are not exclusive. If both conditions hold, either case can be chosen. The third case permits constraints though that result in a compacted diagram with slightly violated prescribed spacings. This is because the rectangles are enlarged by the smallest spacing in order to avoid overlaps.

$s_{ne} < \frac{s_{nn}}{2}$: Add s_{ne} to all node representing rectangles.

$s_{ne} < \frac{s_{ee}}{2}$: Add s_{ne} to all edge representing rectangles.

$s_{ne} > \min\{s_{nn}, s_{ne}\}$: Add $\min\{s_{nn}, s_{ee}\}$ to the node representing rectangles.

A different way to ensure valid spacings in every scenario is to compute the constraint graph in the naive way mentioned in Section 1.1.1. While it takes quadratic time and yields a quadratic number of constraints, the pair-wise comparison of elements allows to handle every scenario discussed above. Again, a length has to be associated with every constraint.

2.1.2 North and South Ports

With the extensions by Schulze et al., edges are allowed to connect to the northern and southern border of a node. Consider the edge $e = (n5, n6)$ in Figure 2.2. One-dimensional compaction, as specified above, would allow the edges' vertical segments to detach from the nodes' perimeters. To prevent this, we create a common group for rectangles of a node and of any vertical segments of edges that attach to the northern or southern side. The offset between the two rectangles is defined by the segment's attachment point subtracted by the node's position. Additionally, for edges such as e,

Table 2.1. Results of applying one-dimensional compaction to layer-based drawings of dataflow diagrams. LR stands for left compaction followed by right compaction, and EL stands for compaction aiming for short edges. $\bar{n}$ and $\bar{e}$ denote the average number of nodes and edges. $\bar{w}$ denotes the average width after compaction in percent of the original width, $\bar{el}$ the average edge length. Standard deviations are given in brackets.

	$\bar{n}$	$\bar{e}$		$\bar{w}(\%)$	$\bar{el}(\%)$
EHANDBOOK	25.4 [15.0]	30.8 [18.3]	LR	83.7 [11.9]	78.2 [15.4]
			EL	85.3 [11.2]	76.6 [16.2]
Ptolemy	15.7 [7.4]	19.6 [11.5]	LR	93.6 [8.2]	88.3 [13.8]
			EL	94.3 [7.4]	87.1 [13.3]

where the edge directly connects a northern port with a southern port, we can improve the result by setting the spacing between the two involved vertical segments to zero if they belong to the same edge. Compare Figure 2.2b and Figure 2.2c.

2.1.3 Edge Length

One-dimensional compaction does not consider the length of an edge. In Figure 2.2b, the placement of nodes is reasonable when aiming for minimal width. To get a visually pleasing layout, however, n2 has to be positioned as far to the right as possible to reduce the edge length, without violating spacing constraints. The problem to minimize the total edge length can be formalized as a minimum cost flow problem and has been well studied in the context of VLSI design, where the length of wires should be minimized [14, 11]. Here, we suggest two simple solutions specifically tailored for graph layout.

LR. To achieve minimal width with some edge length reduction, we compact everything to the left, fix the positions of nodes that have no outgoing edge in the original graph, and execute another compaction pass to the right. This would yield the position for node n2 as seen in Figure 2.2c but may yield sub-optimal results for other diagrams.

EL. To guarantee minimal edge length, we simply take the constraint graph, add the edges of the original graph, and solve the problem using the network simplex algorithm presented by Gansner et al. [8]. The algorithm runs fast in practice and is already present in many implementations of the layer-based approach since it can be used for the layering and node placement step. Care has to be taken when adding the original edges though. Consider Figure 2.3 (a) and (b). Adding an edge from n1 to n2 would prevent n1 from being placed below n2. We solve this using two ideas presented by Gansner et al. We add an auxiliary node to the network simplex graph and two edges from it to both of the existing nodes. The minimal lengths of these edges are chosen such that they reflect the port offsets. Furthermore, vertical segments of *inverted ports*, such as the one of edge (n1, n2) in Figure 2.3c, should stay close to the port's node.
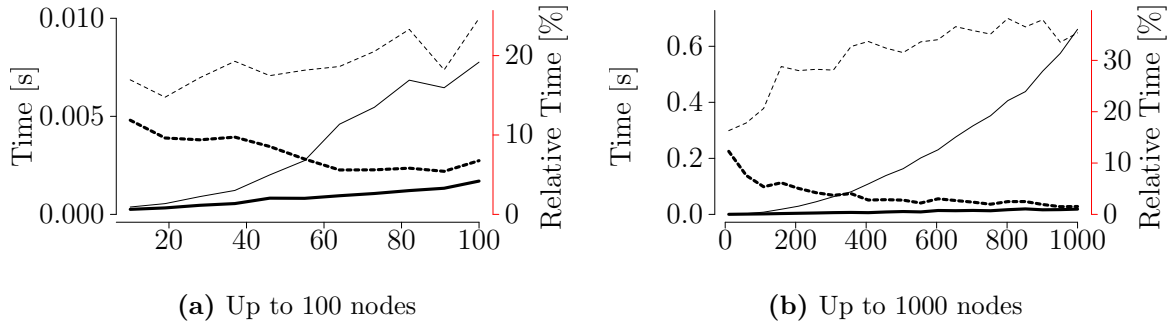
(a) Up to 100 nodes   **(b)** Up to 1000 nodes

Figure 2.4. Execution times of the two compaction strategies plotted against the number of nodes: LR (bold lines) and EL (thin lines). The solid lines show the absolute execution time in seconds (left y-axis) and the dashed lines show the relative execution time compared to the overall algorithm (right y-axis).

2.2 Discussion

Our main goal is to improve on diagrams that occur in practice. We therefore do not use randomly generated diagrams and diagrams from often-used graph sets, such as the *Rome graphs* where all node are of unit size. Our evaluation set consists of 69 diagrams from the commercial interactive model browsing solution EHANDBOOK[1] and a subset of 529 diagrams shipping with the academic Ptolemy project [17]. Note that both diagram types are hierarchical. Nodes can contain further nodes, i.e. sub-diagrams. We extracted such sub-diagrams and evaluate them separately, which is feasible since the layout algorithm considers every sub-diagram separately anyway. For both diagram types the number of nodes per hierarchy level averages between 15 and 25 with slightly more edges.

The results of applying our method can be seen in Table 2.1. We measured values for both compaction strategies mentioned in Section 2.1: subsequent left-right compaction with node locking (LR) and minimizing edge length (EL). The average width of the drawings decreased by about 16% and 6%, the edge lengths decreased by 22% and 12%. No significant difference can be observed between the two compaction strategies. Still, edges that can obviously be shortened are immediately noticed by users (cf. Figure 2.2). We thus suggest to use compaction with edge length minimization.

As seen in Figure 2.4, both methods finish in well under 10ms for up to 100 nodes, with EL using about a fifth of the overall execution time and LR about a tenth. For up to 1000 nodes EL's execution time increases significantly, which is expected since the network simplex algorithm is used. Still, it finishes in under 0.6s. Therefore all setups are fast enough for applications that involve user interaction. We ran the algorithm on an Intel i7 2GHz CPU and 8GB memory laptop using a 64bit JVM.

---

3 Connected Components With External Extensions

When a diagram consists of multiple sub-graphs that are not connected among each other (see Figure 1.2 for a simple example), the problem arises to place the sub-graphs in the plane such that little space is used. Each sub-graph can be approximated by its bounding box and the problem can be formulated as a rectangle packing problem. However, such problems are often NP-complete [14] and rectangles may be poor approximations. Freivalds et al. and Goehlsdorf et al. discuss relevant related work and present heuristics for the problem based on a polyomino representation, which approximates every sub-graph using squares on a grid [4, 9]. The approaches work well for flat diagrams. With data flow diagrams, a node can contain a sub-graph and nodes of the sub-graph can be connected to nodes on other hierarchy levels via so-called *external ports* on the hierarchical node's perimeter. The edge between FrontDropQueue and VariableSleep in Figure 2.1 represents such an *external edge*. When placing the sub-graphs in the plane these edges have to be considered. They are not allowed to cross other sub-graphs. This cannot be prevented using the previously mentioned methods. Furthermore, sub-graphs should be placed such that the overall length of external edges is as small as possible.

Lai et al. present a method based on the *sequence-pair representation*, which is solved using simulated annealing [13]. It allows to specify for a module of a VLSI design, i. e. a rectangle, that it has to touch one of the four boundaries. However, this is not sufficient for the previously described problem and again rectangles do not approximate sub-graphs well.

In the following, we generalize the problem, show how a placement of sub-graphs with connections to external ports can be constructed that is guaranteed to be overlap-free, and use one-dimensional compaction to improve the drawings. To better approximate a sub-graph, we construct its *rectilinear convex hull* and split it into a set of rectangles. Both can be done in $O(n \log n)$ time using a scanline method, where n is the number of points used to represent the area covered by a sub-graph in the first case, and the number of corners of the rectilinear convex hull in the second case.

Definitions. Let $\mathcal{C}$ be a set of *components*. Each component $c_i \in \mathcal{C}$ is a tuple $c_i = (\mathcal{R}_i, \mathcal{E}_i)$, where $\mathcal{R}_i$ is a non-empty set of *rectangles* and $\mathcal{E}_i$ is a (possibly empty) set of *external extensions*. Rectangles are 4-tuples. The k-th rectangle of c_i is r_i^k, with all elements in $\mathbb{R}$. We assume that all rectangles of the same component somewhere touch alongside their border. An external extension $e_i^l = (d_i^l, \delta_i^l, \epsilon_i^l)$ of a component c_i is a triple of a direction $d_i^l \in \{n, e, s, w\}$, an offset δ_i^l relative to r_i^0, and a width ϵ_i^l. The
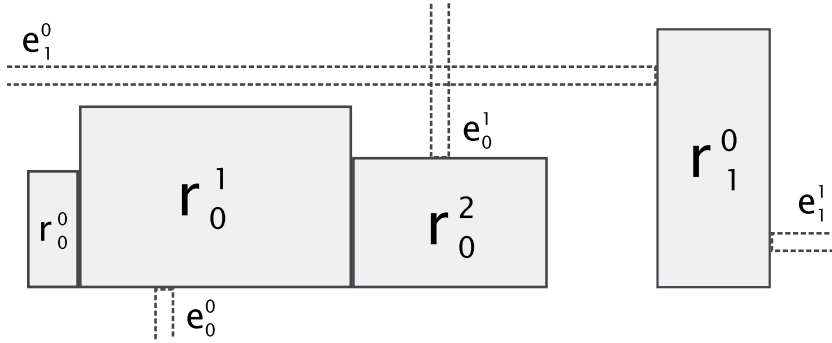
Figure 3.1. The diagram shows two components c_0 and c_1. c_0 consists of three rectangles and two external extensions and c_1 consists of a single rectangle and two extensions. The external extensions e_1^0 and e_0^1 are allowed to overlap since one is vertical and the other one is horizontal. They are not, however, allowed to overlap with any of the rectangles.

offset and the width describe an extension clockwise, i.e. for a south extension, the offset is its right-most point and the width points to the left. Intuitively it represents a line or a strip attached to the border of a rectangle which extends infinitely into the specified direction. We say an extension (d, δ, ϵ) is *horizontal* if $d \in \{w, e\}$ and *vertical* if $d \in \{n, s\}$. See Figure 3.1 for an illustration.

A set of components $\mathcal{C}$ is considered *proper* if no pair of components overlaps and no external extension overlaps a component.

3.1 Construction of a Proper Solution

Using compaction to minimize the area of a set of components requires a proper set of components to start with. We use a simple placement algorithm we call *cell packing* that turns a (possibly improper) set of components into a proper one by calculating sensible x and y-coordinates for all rectangles. Algorithm 3 shows the basic approach, and we will refer to the pseudocode in the following explanations.
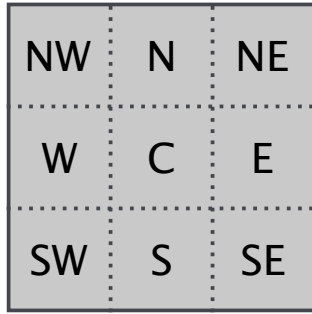
The algorithm takes a set of components to be placed as its input, and computes x and y-coordinates for each component. The basic idea is to add components to a container (*cont* function in the pseudocode) divided into nine *cells*, as shown in Figure 3.2. Each cell can hold components with certain types of external extensions: a component with extensions to the northern and western side will end up in the top left cell, while a component with no extensions at all will end up in the center cell (*cells* function). Each cell can conceptually hold arbitrarily many components since they can easily be placed inside the cell without illegal overlaps.

However, components that have external extensions to opposing sides cannot simply be added to a single cell: placing a component with external extensions to the eastern and western sides in only the left cell would possibly result in its external extensions overlapping with components added to the center and right cells (see Figure 3.2b). Therefore, it
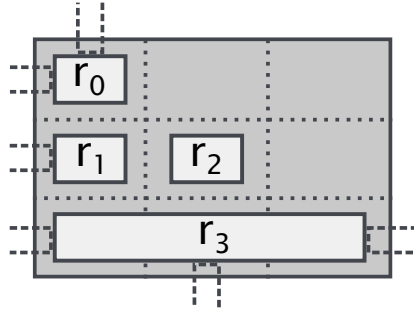

Algorithm 3. Cell Packing

Input: $\mathcal{C}$: set of components to be placed

Data: $cont : \mathcal{C} \to \mathbb{N}$: mapping of components to containers
Data: $cells : \mathcal{C} \to \mathcal{P}\{nw, n, ne, w, c, e, sw, s, se\}$: mapping of components to container cells
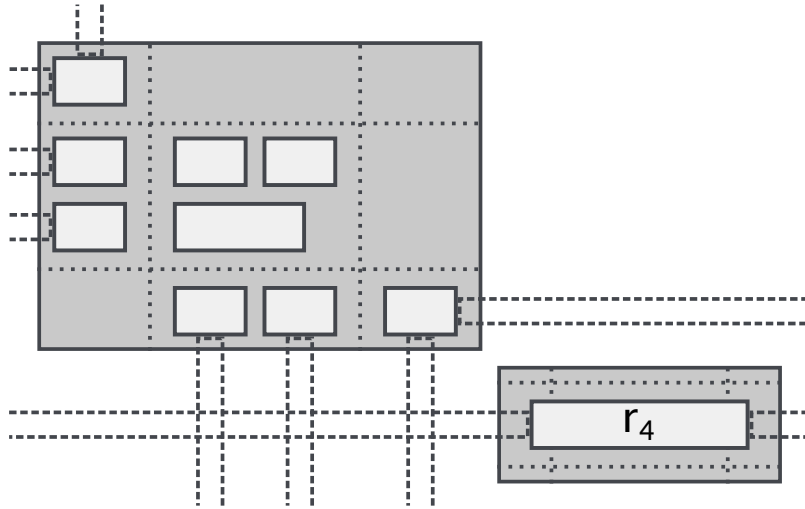
1 $cont(c) \leftarrow \perp \quad \forall c \in \mathcal{C}$
2 $cells(c) \leftarrow \emptyset \quad \forall c \in \mathcal{C}$

3 **for** $c \in \mathcal{C}$ **do**
4      $\quad n \leftarrow \min\left\{i \in \mathbb{N} : canAddTo(c, cont^{-1}(i))\right\}$
5      $\quad cont(c) \leftarrow n$
6      $\quad cells(c) \leftarrow computeCells(c)$

7 $(x, y) \leftarrow (0, 0)$
8 **for** $i \in cont(\mathcal{C})$ **do**
9      $\quad (x, y) \leftarrow placeComponents(f^{-1}(i), (x, y))$

(a) Container with nine cells.　　　**(b)** Container with nine cells.

(c) Final placement of containers.

Figure 3.2. (a) The *cell packing* algorithm places components in containers that are divided into nine *cells*. (b) Each cell can hold a certain type of component, depending on its external extensions. Having extensions on opposite sides, such as r_3, forces a component to span multiple cells. (c) In this example, the rectangle r_4 had to be placed in a new container since it spans the center row of container cells, but the center row in the top left container was not free.
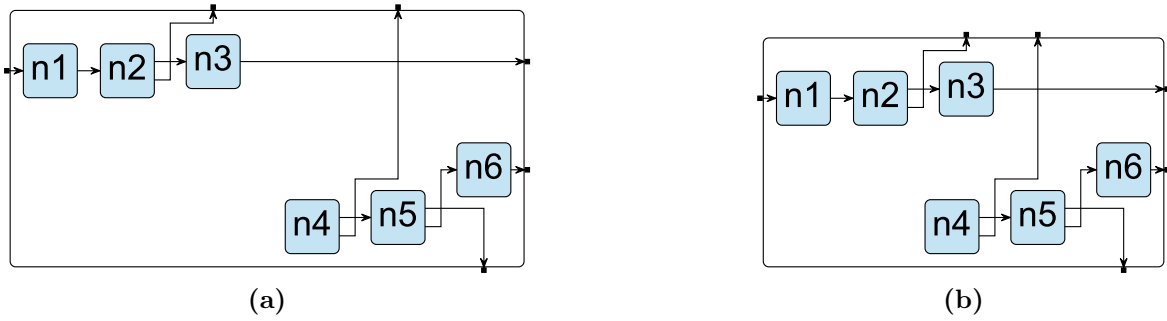
14

Figure 3.3. Placing a diagram's sub-graphs must assert that no external edge crosses a sub-graph. We show how to construct a feasible placement (a) and how to compact it using one-dimensional compaction (b).

needs to span all three cells and needs to prevent other components from being added to either them. We call such components *spanning components*. In Figure 3.2c, a spanning component had to be placed in a new container since the top left container's center row was already partly occupied.

If a component is to be added to a cell which already contains a spanning component, a *conflict* occurs because the cell can only hold the spanning component. Take for example the situation in Figure 3.2. Here, the component r_c has external extensions the the left and right side and would thus be placed in the middle row of the top left container. The two left cells of the middle row already contain components, however, and thereby prevent r_c from being added. The *canAddTo* function called in the pseudocode is defined on a component to be added to a container and a set of components already in that container and checks whether adding the new component would cause such a conflict or not. We simply look for the first container the component can be added to without causing a conflict and add the component to it (lines 4–6). This may actually be a completely new container: a component will never cause a conflict with an empty set of components.

Once all components have been added to containers, they can by design be easily placed inside their cells without illegal overlaps, which is what line 9 does. We end up with containers that potentially have external extensions protruding to all four sides. To avoid illegal overlaps, the containers are placed along a diagonal, which only produces legal overlaps among external extensions. We thus end up with a proper set of components that can be compacted.

Note that the behaviour of the cell packing algorithm could be changed to allow more components to share a container. If a component with external extensions to the western and eastern sides is to be added to a container, it could occupy a separate, fourth row regardless of whether the center row already has components in it or not. More components with horizontal external extensions could easily be added to the same container, but components with external extensions to the northern and southern side would cause a conflict.

3.2 Compaction

For the use case of compacting layer-based drawings described in Chapter 2, it is sufficient to compact along the x-dimension only. This time, however, it is necessary to compact in both dimensions, which is possible by continuously applying one-dimensional compaction in alternating dimensions and directions until no further, or little, progress is made. For a given set of components $\mathcal{C}$ we construct a grouped constraint graph. Each component is represented by a group and the component's rectangles are added to it. The external extensions are converted into finite rectangles: each external extension is cut at the point where it intersects with the bounding box surrounding all components of $\mathcal{C}$. After each compaction pass these lengths have to be adjusted to prevent components from permuting.

Obviously, rectangle representing horizontal and vertical extensions cannot be present at the same time during one-dimensional compaction since the set of rectangles may not be valid. Remember that external extensions are allowed to overlap with each other but the representing rectangles are not allowed to overlap. Still, it is important that the horizontal extensions are considered during vertical compaction, to prevent nodes from overlapping with external extensions (the same is true for vertical extensions during horizontal compaction). The independent application of horizontal and vertical compaction allows to use two different sets of rectangles depending on the compaction direction: $\mathcal{H} = \mathcal{R} \cup \{(d, \delta, \epsilon) \in \mathcal{E} : d \in \{n, s\}\}$ for horizontal compaction and $\mathcal{V} = \mathcal{R} \cup \{(d, \delta, \epsilon) \in \mathcal{E} : d \in \{e, w\}\}$ for vertical compaction.

As mentioned earlier, the goal may not only be a small area but also short external extensions, for instance, if they represent edges of a diagram. We support this by executing an additional compaction pass in each dimension and applying a locking strategy as discussed in Section 2.1's paragraph on edge length. Alternatively, the network simplex approach can be used with an artificial source and sink node connected to rectangle representing nodes with corresponding external edges.

3.3 Discussion

The methods presented in the preceding sections allow us to properly draw diagrams such as Figure 1.2 and Figure 3.3. So far, this was not possible. While we were satisfied with most results we have seen so far, we believe that both the construction and compaction can be significantly improved. Our current set of example diagrams is too small for a meaningful quantitative evaluation though.

4 Final Remarks

In this paper we show how one-dimensional compaction can be applied to two problems from the field of automatic diagram layout, more specifically, layer-based drawings and placement of disconnected graphs. We tested our methods with data flow diagrams from practice and found that the width of layer-based drawings can significantly be reduced and that they allow disconnected graphs with hierarchy-crossing edges that are part of hierarchical graphs to be placed.

In future work, we plan to examine the possibility to insert *jog*s during horizontal compaction, as discussed by Chen and Lee [1]. Mapped to our problem this means splitting a vertical segment and allowing the edge to take a detour, see Figure 4.1b. Furthermore, in scenarios such as Figure 4.1c it is desirable to let nodes jump over vertical segments. Furthermore, we plan to collect a larger set of diagrams from practice to evaluate to component compaction and plan to inspect further compaction strategies: a combination of our construction and polyomino packing [4, 9] as well as *stress-minimizing* methods that are capable of preventing or removing overlaps [3, 7].
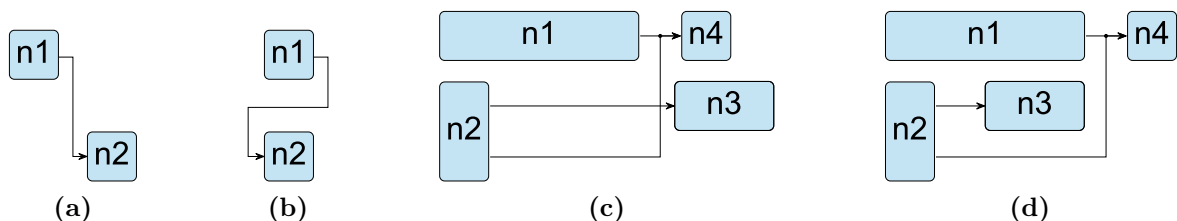

Figure 4.1. Two potential improvements of the layer-based compaction. In (b) the edge was elongated and an additional horizontal segment was introduced to further reduce the diagrams width. In (d) node n3 is allowed to jump over the vertical edge segment. Note that the order of n2's edges must not be changed.

Bibliography

[1] H.-F. S. Chen and D. Lee. A faster one-dimensional topological compaction algorithm. In H. Leong, H. Imai, and S. Jain, editors, *Algorithms and Computation*, volume 1350 of *Lecture Notes in Computer Science*, pages 303–313. Springer Berlin Heidelberg, 1997.

[2] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. In P. Healy and N. S. Nikolov, editors, *Proceedings of the 13th International Symposium on Graph Drawing (GD'05)*, volume 3843 of *LNCS*, pages 153–164. Springer, 2006.

[3] T. Dwyer, K. Marriott, and M. Wybrow. Topology preserving constrained graph layout. In *Revised Papers of the 16th International Symposium on Graph Drawing (GD'08)*, volume 5417 of *LNCS*, pages 230–241. Springer, 2009.

[4] K. Freivalds, U. Dogrusoz, and P. Kikusts. Disconnected graph layout and the polyomino packing approach. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 378–391. Springer Berlin Heidelberg, 2002.

[5] P. Frey, R. von Hanxleden, C. Krüger, U. Rüegg, C. Schneider, and M. Spönemann. Efficient exploration of complex data flow models. In *Proceedings of Modellierung 2014*, Vienna, Austria, Mar. 2014.

[6] C. Friedrich and F. Schreiber. Flexible layering in hierarchical drawings with nodes of arbitrary size. In *Proceedings of the 27th Australasian Conference on Computer Science (ACSC'04)*, pages 369–376. Australian Computer Society, Inc., 2004.

[7] E. R. Gansner and Y. Hu. Efficient node overlap removal using a proximity stress model. In I. G. Tollis and M. Patrignani, editors, *Graph Drawing*, pages 206–217, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[9] D. Goehlsdorf, M. Kaufmann, and M. Siebenhaller. Placing connected components of disconnected graphs. In *6th International Asia-Pacific Symposium on Visualization, 2007*, pages 101–108, Feb 2007.

[10] C. Gutwenger, R. von Hanxleden, P. Mutzel, U. Rüegg, and M. Spönemann. Examining the compactness of automatic layout algorithms for practical diagrams. In

Proceedings of the Workshop on Graph Visualization in Practice (GraphViP'14), Melbourne, Australia, July 2014.

[11] S. E. Hambrusch and H.-Y. Tu. Minimizing total wire length during 1-dimensional compaction. *Integration, the VLSI Journal*, 14(2):113 – 144, 1992.

[12] P. Healy and N. S. Nikolov. Hierarchical drawing algorithms. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 409–453. CRC Press, 2013.

[13] J. Lai, M.-S. Lin, T.-C. Wang, and L.-C. Wang. Module placement with boundary constraints using the sequence-pair representation. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 515–520. ACM, 2001.

[14] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[15] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995.

[16] S. C. North and G. Woodhull. Online hierarchical graph drawing. In *Revised Papers of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 232–246. Springer, 2002.

[17] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.

[18] G. Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, volume 2912 of *LNCS*, pages 381–386. Springer, 2004.

[19] C. D. Schulze, M. Spönemann, and R. von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout*, 25(2):89–106, 2014.

[20] 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, Feb. 1981.