Size- and Port-Aware Horizontal Node Coordinate Assignment

Ulf Rüegg, Christoph Daniel Schulze,
John Julian Carstens, and Reinhard von Hanxleden

Dept. of Computer Science, Christian-Albrechts-Universität zu Kiel, Germany
{uru,cds,jjc,rvh}@informatik.uni-kiel.de

Abstract. The approach by Sugiyama et al. is widely used to automatically draw directed graphs. One of its steps is to assign horizontal coordinates to nodes. Brandes and Koepf presented a method that proved to work well in practice. We extend this method to make it possible to draw diagrams with nodes that have considerably different sizes and with edges that have fixed attachment points on a node's perimeter (ports). Our extensions integrate seamlessly with the original method and preserve the linear execution time.

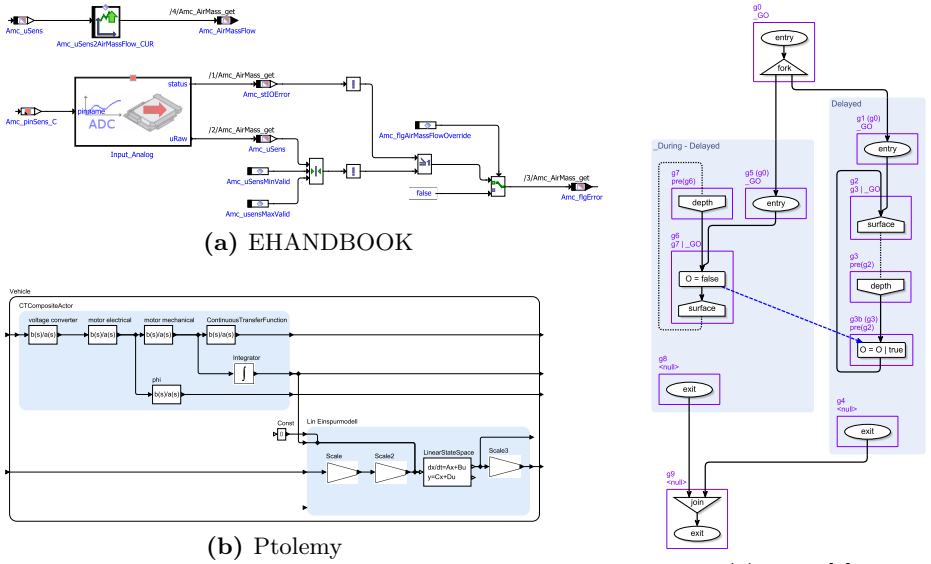
1 Introduction

The layer-based approach to graph layout as introduced by Sugiyama et al. [6] is a well-established methodology to automatically draw directed graphs in the plane. It is defined as a pipeline of three subsequent phases: *node layering* distributes the nodes into subsequent layers such that edges only point from lower to higher layers; *crossing minimization* orders the nodes in each layer such that the number of edge crossings is minimized; finally x-coordinate assignment (or *node placement*) determines x coordinates for nodes. In practice, an initial *cycle breaking* phase as well as a final *edge routing* phase are often added to support cyclic graphs and non-simple edge routing styles.

In the area of model-driven engineering (MDE), graphical languages are often used to model complex software systems. For instance, tools such as *LabVIEW* (National Instruments), *EHANDBOOK* (ETAS), and *Ptolemy* (UC Berkeley) allow to model systems using *data flow diagrams* and make use of automatic layout algorithms to arrange nodes and edges. In such diagrams edges are usually routed in an orthogonal fashion and connect to nodes through dedicated attachment points on a node's boundary (so-called *ports*). Also, nodes have considerably different sizes, see Fig. 1 for examples.

All of these characteristics pose challenges for automatic graph drawing algorithms that are rarely addressed by existing solutions. Previous work by Schulze et al. [5] introduced methods that extend the layer-based approach to support the special requirements of data flow diagrams, focusing on crossing minimization and edge routing. In this paper, we focus on node placement.

While we refer to Healy and Nikolov [3] for a general overview of existing node placement approaches, it is worth noting that most of them try to a certain



(a) EHANDBOOK

(b) Ptolemy

(c) SCG [7]

Fig. 1. Exemplary drawings using the methods presented here. They would not be drawable solely with the existing algorithm [1] since they contain ports and nodes of considerably different sizes.

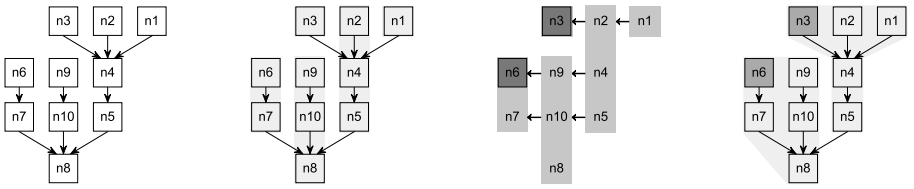
extent to reduce the number of edge bend points. For one thing, the approach introduced by Sander [4] ensures that long edges are always drawn straight, but uses a barycenter-like balanced placement for all other edges. Once a node has more than one outgoing edge, this usually results in two bend points per edge. For another thing, the approach introduced by Brandes and Koepf [1], extending ideas of Buchheim et al. [2], tries to draw as many edges straight as possible.

Contributions. Brandes and Koepf assume that all nodes have the same size and do not take ports into account; thus their algorithm straightens at most one outgoing edge per node. In this paper, we extend the approach by Brandes and Koepf to remove these restrictions and take the opportunity to place nodes such that more than one outgoing edge per node can be drawn straight. This leads to drawings as seen in Fig. 1. Throughout the paper we will assume that the node placement algorithm cannot change the size of nodes and the position of ports.

Outline. Following the usual conventions, we start by introducing the required terminology in the next section. Sec. 3 then gives an overview of the algorithm by Brandes and Koepf before Sec. 4 introduces our extensions. We evaluate our algorithm in Sec. 5 and close with a conclusion and future work in Sec. 6.

2 Preliminaries

Let $G = (V, P, \pi, E)$ denote a directed graph with ports, where V is a set of nodes and P a set of ports, i. e. attachment points on a node's boundary. $\pi : P \mapsto V$



(a) Graph **(b)** Blocks **(c)** Block graph **(d)** Classes

Fig. 2. Gray boxes in (b) show the calculated alignments (blocks) for the graph (a). In (c) the block graph is depicted with the two sinks in darker gray, (d) shows the corresponding classes.

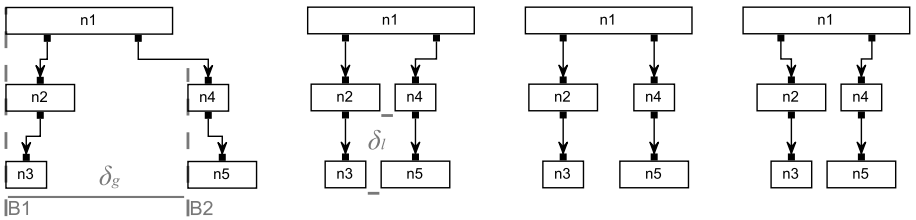
assigns each port to a node. $E \subseteq P \times P$ is a set of directed edges connecting the ports.

During the first steps of the layer-based approach cyclic graphs are made acyclic, a *layering* is calculated, and an ordering is determined for each layer. A layering $\mathcal{L}$ is an ordered partition of V into non-empty *layers* $L_1, \ldots, L_{|\mathcal{L}|}$ and $\mathcal{L}(v) \to \{1, \ldots, |\mathcal{L}|\}$ maps each node $v \in V$ to the index of its respective layer. Since all edges must point in the same direction, $\mathcal{L}(\pi(p)) < \mathcal{L}(\pi(q))$ must hold for all edges $(p, q) \in E$. An edge (p, q) is *short* if $\mathcal{L}(\pi(q)) - \mathcal{L}(\pi(p)) = 1$; it is *long* otherwise. A layering is *proper* if all edges are short. Note that a layering can be made proper by splitting long edges and introducing *dummy nodes*. We refer to the short edges of a proper layering as *edge segments*. That is, an original edge can be represented by one or more edge segments. Each layer $L_i \in \mathcal{L}$ is an ordered tuple of nodes $(v_1^i, \ldots, v_n^i)$, where $n = |L_i|$. The position of a node v_j^i in layer i is $pos(v_j^i) = j$ and the predecessor of a node v_j^i with $j > 1$ is $pred(v_j^i) = v_{j-1}^i$. This gives a properly layered, directed, acyclic graph with ports (LDAGP) $G' = (V', P', \pi', E', \mathcal{L})$. The set of nodes now includes a set of dummy nodes $\mathcal{D}$ such that $V' = V \cup \mathcal{D}$. For each dummy node two ports are introduced and edges are added and reconnected accordingly.

Finally, let $width : V' \mapsto \mathbb{R}$ assign a width to each node. Throughout this paper, we assume that for an edge $(p, q) \in E$, p is on the lower boundary of $\pi(p)$ and q is on the upper boundary of $\pi(q)$ to prevent edges from crossing nodes. Let $x_p : P' \mapsto \mathbb{R}$ assign positions to ports relative to the leftmost point on their respective boundary.

3 The Original Algorithm

In this section we give a brief summary of the original algorithm of Brandes and Koepf. For further details we refer to the paper itself [1]. The basic idea of the algorithm is to traverse a given graph in different directions to calculate four extremal layouts and combine them into a balanced final layout. The algorithm is divided into the following steps: 1) During *Vertical Alignment* nodes are combined into so-called *blocks*. Different directions may result in different



(a) Original **(b)** Compact **(c)** Straight **(d)** Balanced

Fig. 3. Illustration of the additional challenges for the node placement phase imposed by ports. In (a) a global separation value δ_g is used to space blocks B1 and B2, and ports are neglected. (b) shows a compact drawing where ports are considered and the same blocks can flow into one another using a local separation δ_l. (d) shows the result of executing a balancing step (as it is part of the original algorithm [1]) in conjunction with orthogonally drawn edges that, as opposed to (c), yields more edge bends.

blocks. Edges between the nodes in a block will be drawn straight. 2) *Horizontal Compaction* moves the calculated blocks as close to each other as possible and assigns explicit x coordinates to nodes. Depending on the direction nodes are either compacted leftwards or rightwards. 3) *Balancing* combines the four extremal layouts resulting from the previous two steps to a final drawing.

A direction is a combination of traversing the layers of $\mathcal{L}$ either downwards or upwards and traversing the nodes in each layer either rightwards or leftwards. For brevity, we will limit our explanations and examples throughout this paper to the combination of downwards and rightwards. The other three combinations are easy to infer.

During the alignment step nodes are aligned with their median neighbor in the preceding layer. Consecutively aligned nodes are referred to as a block, see Fig. 2b for an illustration. Let $\mathcal{B}$ denote the set of blocks of an LDAGP G', where each block $b \in \mathcal{B}$ is represented as an ordered tuple of edges $(e_0, \ldots, e_n)$. For compaction, an auxiliary *block graph* is constructed as seen in Fig. 2c. Blocks are the nodes in the block graph and are connected by an edge if two nodes of different blocks are consecutive in their layer. Within the block graph, blocks are divided into *classes*. A class is defined by a unique sink that is reachable by all of the class's nodes. Positions subject to a global separation value δ_g are then assigned to blocks using a longest path layering within each class, which recursively assigns positions relative to the class's sink. If two adjacent blocks are part of the same class, their relative positions can be determined immediately. If they belong to different classes, the blocks impose a minimum required separation between the involved classes. This separation is remembered and applied after all blocks have been placed.

As mentioned earlier, the original approach does not cater for varying node sizes and ports. For one thing, ports reveal two problems that are illustrated in Fig. 3a. First, in the depicted graph no edge is drawn straight even though all nodes of the blocks B1 and B2 are neatly left-aligned. Second, node n1 has two

Algorithm 1. inner_shift

Input: LDAGP with blocks $\mathcal{B}$
Output: innerShift[v] (inner-block offset of node v), blockSize[b] (size of block b)

```
1  function inner_shift()
2      innerShift[v] ← 0   ∀v ∈ V
3      for b ∈ B do
4          left ← 0; right ← 0
5          for (p, q) ∈ b do
6              s ← innerShift[π(p)] + x_p(p) − x_p(q)
7              innerShift[π(q)] ← s
8              left ← min(left, s)
9              right ← max(right, s+width(π(q)))
10         for all nodes v in block b do
11             innerShift[v] −= left
12         blockSize[b] ← right − left
```

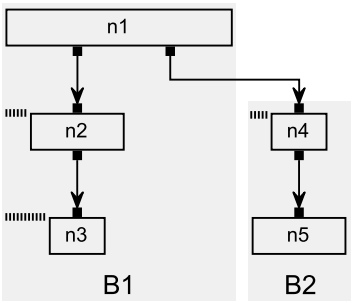


Fig. 4. Illustration of the inner shift. Nodes n2, n3, and n4 have an inner shift value different from zero. It defines the offset within the node's block and is depicted by the dashed lines. Also, both blocks B1 and B2 have an extent.

ports both of which would allow the connected edge to be drawn straight. Yet, n1 and n4 are part of different blocks that will be separated during the compaction step. In addition, different node sizes increase the two aforementioned problems and render the global separation value δ_g impractical. δ_g would have to be larger than the widest node of the graph to avoid overlapping nodes, possibly leaving a lot of whitespace. Figures 3b and 3c show two drawings that would be more desirable using a local separation δ_l. Furthermore, in conjunction with orthogonally drawn edges, as opposed to general polylines, the balancing step often yields undesirable bendpoints (see Fig. 3d). For this reason we consider the balancing step to be optional and, if discarded, choose the final layout out of the four possible candidates based on the smallest width.

During the rest of this paper, we will keep our explanations and pseudo code as close as possible to the style and notation of the original paper. There, the following data records are used for a node $v \in V$: $root[v]$ denotes the root node of v's block; $align[v]$ maps to the next node within v's block in the current iteration direction and represents a cyclically linked list; $sink[v]$ stores the sink of the class v belongs to; $shift[v]$ holds the distance by which the class of v should be moved during compaction.

4 Size- and Port-Aware Node Coordinate Assignment

In this section we present our extensions. First, we add a step that we call *inner shift*. It calculates offsets for nodes within a block to account for ports and simultaneously determines the width of the blocks, which is required to calculate the size of a layout. Second, we extend the compaction phase to consider node sizes when calculating explicit x-coordinates. Third, we modify the objective

Algorithm 2. place_block

```
   Input: v (root node of a block)
 1 function place_block(v)
 2     if x[v] undefined then
 3         x[v] ← 0; initial ← true; w ← v
 4         repeat
 5             if pos[w] > 0 then
 6                 n ← pred[w]; u ← root[n]
 7                 place_block(u)
 8                 if sink[v] = v then sink[v] ← sink[u]
 9                 if sink[v] ≠ sink[u] then
10                     s_c ← x[v] + innerShift[w] − x[u] − innerShift[n] − width[n] − δ_l
11                     shift[sink[u]] ← min(shift[sink[u]], s_c)
12                 else
13                     s_b ← x[u] + innerShift[n] + width[n] − innerShift[w] + δ_l
14                     if initial then x[v] ← s_b else x[v] ← max(x[v], s_b)
15                     initial ← false
16             w ← align[w]
17         until w = v
```

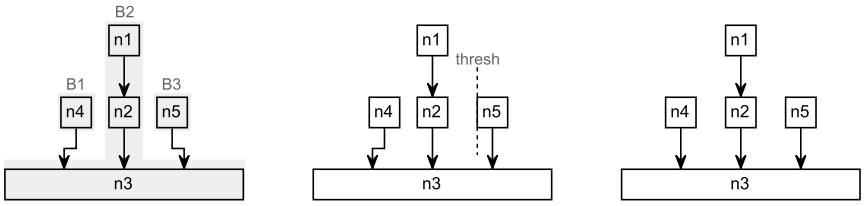
such that more straight edges are, to a certain extent, favored over achieving the most compact layout possible. All additions integrate seamlessly with the original algorithm and preserve its linear execution time.

While the first two modifications share the objectives of the original algorithm, the third one considers straight edges to be more important than compactness. Since different diagram types demand different aesthetics, the third change is optional in our algorithm.

Node Size and Port Support. The original algorithm assigns the same x-coordinate to all nodes within a block. This automatically yields straight edges if all nodes have the same size and the same attachment points for edges. Here we extend this in two ways. First, blocks have a width that depends on the sizes of the block's nodes. Second, each node has an *inner shift*, which is an offset relative to a block's left border. The inner shift is used to properly deal with ports.

Given a set of blocks $\mathcal{B}$ calculated by the vertical alignment method of the original algorithm, we execute Alg. 1. For each block, it iterates through the block's edges (p, q), considers p to be fixed and determines an offset value for $\pi(q)$ such that (p, q) can be drawn straight. Additionally, the maximum extent of the nodes to either side of the starting node's leftmost coordinate is recorded. Using these values, the size of each block is calculated and all inner shift values are shifted to be relative to the leftmost coordinate of any of the block's nodes. The block size is used to determine the width of each extremal layout. Fig. 4 illustrates the effect of the inner shift.

Given an inner shift for the nodes of each block, the horizontal compaction technique is applied with the alterations seen in lines 10 and 13 of Alg. 2. Contrary to the original method, the inner shift and the width of the nodes are considered while iterating through the block. Note that we consider the individ-



(a) Without straightening **(b)** After place_block **(c)** After post_process

Fig. 5. Illustration of the procedure to straighten additional edges during the compaction step. A threshold value is used to prevent n5 in (b) from compacting "too far" in order to get an additional straight edge.

ual width of every node and do not use the overall width of a block. This allows blocks to "flow" into each other, as seen in Fig. 3b.

Moreover, the inner shift of a node and its size have to be considered during the final balancing step, which is easy to incorporate into the original algorithm.

Improving Straightness. A wider node can allow for more than one edge to be drawn straight. The original algorithm did not have to address this since nodes were considered to be uniform. We solve this as follows. Remember that our extended compaction step as shown in Alg. 2 compacts blocks and classes as much as possible. This implies that for a given iteration direction only such edges are possible candidates for additional straightening where one of the involved blocks was moved "too far" (for instance node n4 in Fig. 3b). In other words, we have to prevent the blocks of such edges to be compacted too far in order to get more straight edges.

The procedure we apply can be seen in Fig. 5. In (a) everything is compacted as much as possible. In (b) a threshold value thresh is used to prevent node n5 from moving further to the left, resulting in a straight edge.

A threshold value can, however, only be determined if the connected block is already placed. Consider Fig. 5a and the iteration direction down and right. The algorithm starts by placing block B2, but has to place block B1 before it can finish B2. Now, when placing node n4, no threshold can be calculated because node n3 has not been placed yet. In such a case we delay the straightening of the outgoing edge of n4 until all blocks have been placed. A queue is used to store such edges. Imagine a further node n4' connected to n3 and located between n4 and n2. Just as n4 it will be delayed. To give both edges a chance to be straightened later, it is important to post-process n4 prior to n4'. Using a queue allows to do exactly this. When all blocks have been placed we fetch edge by edge from the queue and check for the involved block how far it can be moved without exceeding the threshold or overlapping other nodes. This way the edge becomes either straight or shortens as much as possible (see Fig. 5c for a final result). Alg. 3 shows the modification of the place_block function. A threshold value is calculated and used as an additional bound for a new block position s_b.

Algorithm 3. place_block with straightening

```
   Input: v: root node of a block
   function place_block(v)
       2.5: thresh ← −∞
       7.5, 15.5 (as else of if in line 5): thresh ← calculate_threshold(v, u, thresh)
       13: s_b ← max(thresh, x[u] + innerShift[n] + width[n] − innerShift[w] + δ_l)

   Input: v (root node of a block); w (current node); ot (current threshold value); Q (queue)
1  function calculate_threshold(v, w, ot)
2      thresh ← ot
3      if v = w then
4          (p, q) ← pick incoming edge of w
5          if block of root[π(p)] placed then
6              thresh ← x[root[π(p)]] + innerShift[π(p)] + x_p(p) − innerShift[π(q)] − x_p(q)
7          else if w has incoming edges then
8              enqueue w to Q
9      if thresh = −∞ and align[w] = v then
10         symmetric to before, this time picking an outgoing edge
11     return thresh

12 // the following method is called after all blocks have been placed
13 function post_process()
14     while Q not empty do
15         w ← dequeue from Q
16         (p, q) ← previously picked edge for w (line 4)
17         t_1 ← x[root[π(p)]] + innerShift[π(p)] + x_p(p)
                   − x[root[π(q)] − innerShift[π(q)] - x_p(q)
18         t_2 ← minimum distance between block of w and its neighbors
19         t ← if abs(t_1) < abs(t_2) then t_1 else t_2
20         move all nodes v in w's block by t
```

We only list code that is added to Alg. 2 and prefix code lines by a fractional number, e. g. `2.5` to denote an addition between lines 2 and 3.

As it is now, thresholds are only calculated for edges incident to either the root or the last node of a block. Note that blocks can consist of a single node in which case this node is both the root and the last node of the block. To pick an edge in line 4 of Alg. 3 we use the first edge incident to `w` that connects to an already placed node. There are three points by which this procedure might be improved in the future: 1) check whether edges between nodes that are neither root nor last in a block can be drawn straight, i. e. calculate thresholds for those nodes as well 2) when multiple edges incident to a block are candidates to be drawn straight, choose the one that allows the most compact layout, i. e. the one with the smallest difference of node and threshold value, and 3) be more intelligent in picking an edge instead of just using the first one that is encountered. Nevertheless, the described method removes bends on edges that, to a human, are obvious candidates for straightening. For instance, nodes that are connected by a single edge to a larger node.

Execution Time. For an LDAGP G', the original algorithm runs in time linear to the number of nodes and edge segments, $O(|V'| + |E'|)$. Alg. 1 is linear in the number of edge segments that are involved in blocks. Alg. 2 only adds constant

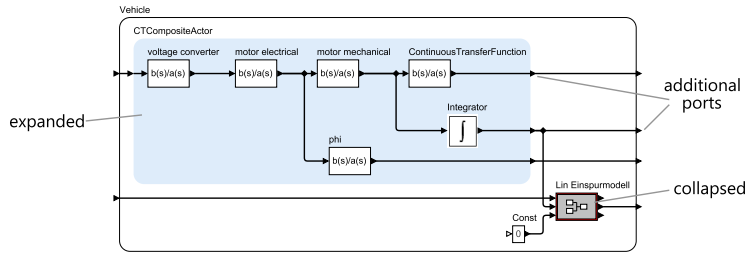


Fig. 6. Same diagram as in Fig. 1b, however, now with the bottom right node collapsed. Note how it is not possible anymore, as opposed to Fig. 1b, to draw the edge of the Const node straight.

time operations to the procedure of the original algorithm. Alg. 3 additionally calculates the threshold value which influences which edge will be picked later. To pick an edge, for every node the incident edge segments are touched at most once. Adding elements to and removing them from a queue can be done in constant time and the post processing step is bounded by the number of nodes and edge segments. Therefore, the overall execution time remains linear in the number of nodes and edge segments.

5 Evaluation

All drawings seen in Fig. 1 were created using the methods of Schulze et al. [5] in combination with our extensions. The methods are implemented in the KLay Layered algorithm and the drawings are created using the KLighD framework, both of which are part of the KIELER open source project.[1]

Recall that we present two contributions here: 1) Supporting varying node sizes and ports, which allows us to draw more diagram types in the first place. 2) The possibility to further increase the number of straight edges if desired. To measure the performance of our second contribution, we need to know how many edges can be drawn straight theoretically for a given graph without violating any overlap and separation constraints. To obtain such numbers we formulated an optimization problem and solved it using CPLEX. In 38 out of 9729 layout executions the solver did not finish within our set time limit of one hour. Nevertheless, we use the reported results since they are always equal to or better than the results of the constructive methods discussed above and thus provide reasonable bounds. As a second metric we use the width of a drawing. It was already noted by Brandes and Koepf that straightening edges might hamper reducing the width of a drawing [1]. A drawing with minimum width can be achieved by placing all nodes of a layers as close to each other as possible and centering every layer in the drawing. Now, given an optimum number of straight edges and a minimum width for a certain graph, we can compare the performance of our extensions, once without straightening (BK) and once with straightening (BKS). It

[1] `http://www.rtsys.informatik.uni-kiel.de/en/research/kieler`

did not make much sense to try and compare our algorithm to the plain original or to other node placement algorithms as they either do not support ports and variable node sizes or do not try to maximize the number of straight edges, or both.

We use four different diagram types for the evaluation: 1) randomly generated graphs with same-sized nodes, 2) data flow diagrams shipping with the academic Ptolemy project,[2] 3) data flow diagrams from the commercial interactive model browsing solution EHANDBOOK[3], and 4) SCGs, which are specialized control flow graphs for sequentially constructive programs [7]. The Ptolemy and EHANDBOOK diagrams are meant to be navigated using an expand/collapse mechanism. Fig. 6 shows a diagram with both an expanded hierarchical node and a collapsed one. Scenarios where more than one edge can be drawn straight are more likely in the presence of expanded nodes as they are wider. We therefore fully expanded existing diagrams for our evaluation and then extracted each hierarchical level into a separate diagram. KLay Layered supports hierarchical graphs by introducing additional ports where an edge crosses a hierarchy boundary, see for instance Fig. 6. The layout is then performed in a bottom-up fashion and additional ports are considered to be dummy nodes. After the evaluations we realized that the aforementioned extraction of subdiagrams kept several edges from being drawn straight since additional ports were fixed at disadvantageous positions. We believe the results could be better than reported.

Tab. 1 summarizes the characteristics of each type of diagram and Fig. 7 shows a scatter plot for each one of them. It can be seen that for diagrams with same-sized nodes BK finds optimal or near-optimal solutions. The other three plots indicate that while BK's overall performance is still very good, there are diagrams for which the number of straight edges can be improved. This is due to variable node sizes. BKS performs better here. The overall number of straight edges increases as well as the number of diagrams for which an optimum solution is found. For SCGs BKS produces more straight edges for almost every diagram. The average width of the tested diagrams on the other hand does not increase notably, which implies that for the tested graphs the additionally straightened edges did not negatively affect the width.

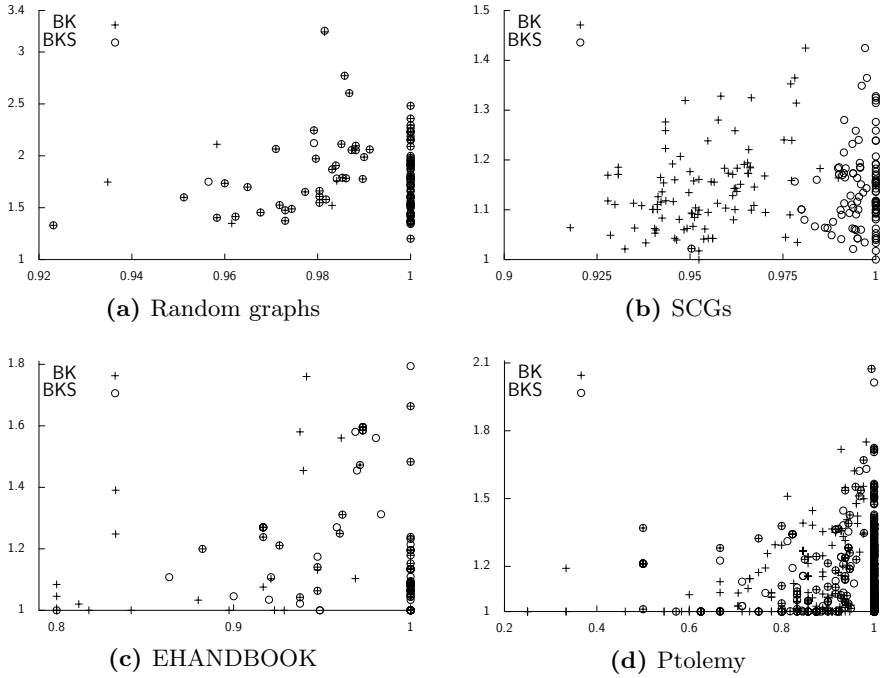
Execution Time. We measured the execution time of BK and BKS using randomly generated graphs with 40 different node counts between 10 and 1000, 1.5 edges per node, and node widths varying between 20 and 100. For each graph size, we generated 10 random graphs and ran the algorithm 10 times, using the average execution time as result. The tests were executed using a 64bit JVM on a laptop with an Intel i7 2GHz CPU and 8GB memory.

For graphs with up to 100 nodes both strategies finish in under 2.5ms and require about 62ms for graphs with 1000 nodes. The average difference between BK and BKS is below 1ms. Therefore, both strategies are fast enough to be used in interactive modeling and browsing tools.

[2] `http://ptolemy.eecs.berkeley.edu/`
[3] `http://www.etas.com/de/products/ehandbook.php`

Table 1. Summary of the evaluation data. For each diagram type the number of diagrams d is listed alongside the average number of nodes $\bar{n}$ and edges $\bar{m}$ per diagram. IE is the percentage increase for BKS compared to BK in the overall sum of all diagrams' straight edges. IS indicates the increase of the average diagram size. By size we mean the width of top-down drawings and the height of left-right drawings. ID represents the number of diagrams for which BKS found more straight edges than BK. OBK and $OBKS$ represent the number of diagrams for which BK and BKS found the optimum number of straight edges.

Type	d	$\bar{n}$	$\bar{m}$	$IE(\%)$	$IS(\%)$	$ID(\%)$	$OBK(\%)$	$OBKS(\%)$
Random	106	29.5	46.5	0.1	0.0	4.7	58.5	60.4
SCGs	107	134.4	268.7	3.5	0.0	96.3	2.8	47.7
EHANDBOOK	97	21.6	24.1	3.7	2.1	18.6	58.8	66.0
Ptolemy	1140	10.6	13.7	2.3	0.2	15.4	74.6	87.0



(a) Random graphs **(b)** SCGs

(c) EHANDBOOK **(d)** Ptolemy

Fig. 7. A scatter plot for each diagram type. The performance in terms of straight edges (x-axis) is plotted against the diagram size (y-axis). The diagram size is either the width or the height of the diagram depending on the layout direction. Each data point represents the performance of BK (or BKS) for a given graph instance relative to the optimum performance for that graph. Thus, the closer a data point is to the bottom right corner of the coordinate system (or 1.0) the better.

6 Final Remarks

We presented extensions to the node placement algorithm presented by Brandes and Koepf [1] to support different node sizes and ports. These extensions make the algorithm usable for a wider range of diagram types, including data flow diagrams. We evaluated our extensions on randomly generated diagrams as well as on three sets of real-world diagrams and found that the results often were near the optimum in terms of straight edges and compactness does not suffer. Performance-wise, the algorithm fares well enough to be used in interactive applications.

For certain graphs, straightening edges may still lead to less compact diagrams. Our intuition is that drawing very few edges in a given diagram non-straight would often lead to a more compact layout. Future work could go into confirming or refuting this intuition and developing methods to find such edges.

Acknowledgements. This work was supported by the German Research Foundation under the project *Compact Graph Drawing with Port Constraints* (ComDraPor, DFG HA 4407/8-1).

References

1. U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *LNCS*, pages 33–36. Springer, 2002.
2. C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. In J. Marks, editor, *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of *LNCS*, pages 229–240. Springer, 2001.
3. 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.
4. G. Sander. A fast heuristic for hierarchical Manhattan layout. In F. J. Brandenburg, editor, *Proceedings of the Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 447–458. Springer, 1996.
5. 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.
6. 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.
7. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design*, 13(4s):144:1–144:26, July 2014.