INSTITUT FÜR INFORMATIK

Drawing Layered Hypergraphs

Miro Spönemann, Christoph Daniel Schulze,
Ulf Rüegg, and Reinhard von Hanxleden

Bericht Nr. 1404
April 2014
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Drawing Layered Hypergraphs

Miro Spönemann, Christoph Daniel Schulze,
Ulf Rüegg, and Reinhard von Hanxleden

Bericht Nr. 1404
April 2014
ISSN 2192-6247

e-mail: {msp,cds,uru,rvh}@informatik.uni-kiel.de

An extended abstract of this work is published at the
8th International Conference on the Theory and Application of Diagrams,
Melbourne, Australia, July 2014

Abstract

Orthogonally drawn hypergraphs have important applications, e. g. in actor-oriented data flow diagrams for modeling complex software systems. Graph drawing algorithms based on the approach by Sugiyama et al. place nodes into consecutive layers and try to minimize the number of edge crossings by finding suitable orderings of the nodes in each layer. With orthogonal hyperedges, however, the exact number of crossings is not determined until the edges are actually routed in a later phase of the algorithm, which makes it hard to evaluate the quality of a given node ordering beforehand.

In this report, we present and evaluate two crossing counting algorithms that predict the number of crossings between orthogonally routed hyperedges much more accurately than previous methods. We also describe methods for routing hyperedges that span multiple layers and for handling junction points.

Contents

1 Introduction

Embedded software domains such as the automotive, rail, or aerospace industries increasingly take advantage of graphical modeling based on the *actor-oriented* approach [9]. Therein, data flow diagrams are used to represent software systems through actors that receive and send data, drawn as nodes, and the data connections between them, drawn as directed edges. However, such diagrams are only helpful as a development tool if they are easy to understand.

The readability of diagrams is usually measured through a set of aesthetic criteria of which the number of edge crossings is considered to be among the most important ones [11, 17]. Both the placement of nodes and the routing of edges thus determine if a diagram is easy to understand or not. Given that, according to Klauske [8], developers spend an estimated 25% of their time on manual layout adjustments to enhance readability, algorithms for computing readable diagram layouts can significantly increase developer productivity.

Actor-oriented data flow diagrams can be formalized as *directed hypergraphs*. A directed hypergraph is a pair $G = (V, H)$ where V is a set of nodes and $H \subseteq \mathcal{P}(V) \times \mathcal{P}(V)$ is a set of *hyperedges*. Each hyperedge $(S, T) \in H$ has a set of *sources* S and a set of *targets* T. For a given directed hypergraph, a layout algorithm computes an embedding in the plane by assigning positions to all nodes and by routing edges between their end points through the computation of bend points.

The well-known layer-based approach to graph drawing proposed by Sugiyama et al. [15] lends itself well to drawing data flow diagrams because it emphasizes the flow of data by placing nodes in consecutive layers on the basis of the connections between them. However, it assumes edges between layers to be drawn as straight lines and thus can lead to inferior layouts with orthogonal hyperedges, especially when it comes to minimizing the number of edge crossings. This comes from the fact that the number of crossings between straight-line edges is a bad predictor for the number of crossings between orthogonal hyperedges in the final drawing.

Contributions. In this report, we introduce and evaluate methods to remedy these shortcomings. We propose two methods for counting crossings that predict the number of crossings much more accurately, as our evaluation shows. Furthermore, we propose a method for combining the segments of hyperedges that span multiple layers to reduce visual clutter and introduce a method for computing and drawing the junction points of hyperedges to remove ambiguity.

Examples of actor-oriented data flow diagrams drawn with our method can be seen in Figure 1.

Outline. We continue by describing the layer-based approach and by reviewing related work. Section 2 describes the general ideas for integrating hyperedges into the layer-based approach, including the combination of hyperedge segments and the computation of junction points. Section 3 then focusses on how to predict the number of edge crossings between orthogonal hyperedges. After an experimental evaluation in Section 4, we conclude the report in Section 5.

1.1 The Layer-Based Approach

The main goal of the layer-based approach to graph drawing is to produce drawings with the majority of edges pointing to the same direction. For this report, we assume that they should point from left to right, which implies that the nodes in each layer are placed below one another. While this is inconsistent with most of the graph drawing literature which assumes a top-down layout, the left-to-right layout is commonly used for data flow diagrams.

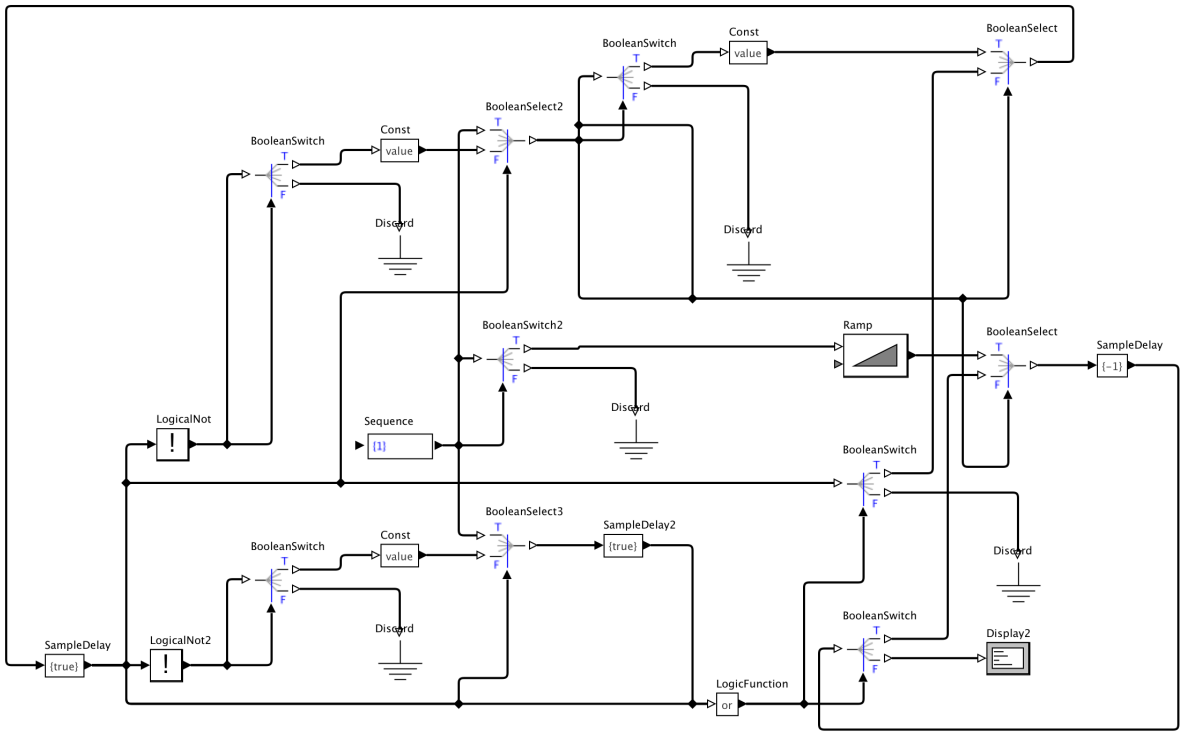
The layer-based approach accepts an abstract acyclic directed graph as its input and produces an embedding in the plane. It is divided into three phases:

1. *Layer Assignment.* Nodes are assigned to layers $L_1, \ldots, L_k$ such that edges point from layers of lower index to layers of higher index. A *proper layering* is obtained by inserting *dummy nodes* such that all edges connect nodes in consecutive layers.

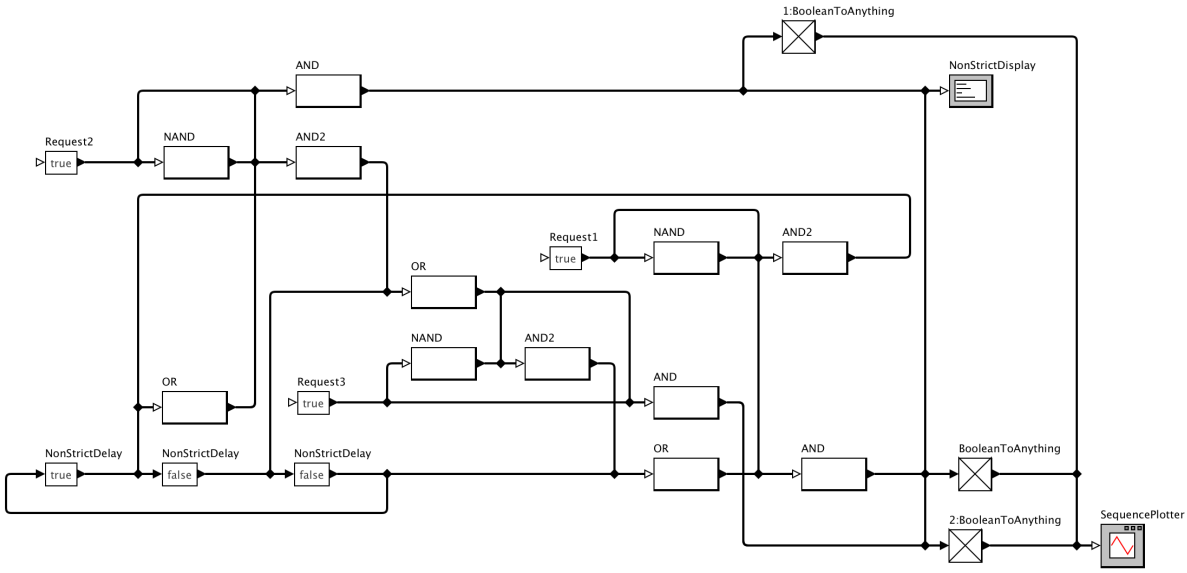
2. *Crossing Minimization.* The nodes in each layer L_i are ordered to minimize edge crossings between L_i and one of its neighboring layers (e. g. using the *barycenter* heuristic). *Layer sweeps* are performed that iteratively minimize edge crossings for all pairs of consecutive layers. It is in this phase that we need an algorithm for counting edge crossings, such as the ones we introduce in Section 3.

3. *Node Placement.* The nodes of each layer are assigned explicit vertical positions within the layer subject to the previously determined order. Goals can be to minimize edge length or to minimize the number of bend points. With orthogonal hypergraphs, it usually is the latter.

This approach is usually extended with two additional phases. A *cycle elimination* phase at the beginning allows input graphs to be cyclic and tries to reverse as few edges as possible to make the graph acyclic. Orthogonal edge routing, which is the standard for actor-oriented data flow diagrams, can be realized by adding an additional *edge routing* phase. The phase processes each pair (L_i, L_{i+1}) of consecutive layers, adding two bend points to each edge that points from L_i to L_{i+1}, except for those where the start and end point are at the same vertical position. This introduces either one horizontal or one vertical and two horizontal line segments for each edge. Since the line segments of different edges may cross, it is important to arrange the vertical segments in such an order that the number of crossings is minimized. Algorithms for ordering vertical segments have been proposed by Sander [12] and Baburin [1].



(a) *Stack*



(b) *Token Ring*

Figure 1. Two diagrams taken from the set of demo models shipping with the Ptolemy tool [5] drawn with the methods presented in this report.

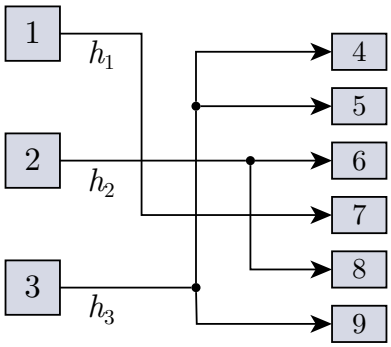


Figure 2. Hyperedges may have cyclic dependencies in the auxiliary graph (V^*, E^*). In this example $E^* = \{(h_1, h_3), (h_3, h_2), (h_2, h_1)\}$, hence we have a cycle $h_1 \to h_3 \to h_2 \to h_1$. No matter how the vertical line segments are ordered, the number of crossings is 4.

1.2 Related Work

To draw hypergraphs with orthogonal edges, Eschbach et al. use standard methods for the layer-based drawing approach [7]. They give no details however on what kind of dummy nodes and edges to create for representing hyperedges, and when to merge the dummy nodes. A more complex solution for representing hyperedges was proposed by Sander [13]. His representation includes a *layering graph* that is constructed to compute the layer assignment, and a *crossing reduction graph* constructed for the crossing minimization phase. Furthermore, Sander represents the orthogonal routing of hyperedges with an efficient data structure that stores horizontal and vertical line segments. Both Eschbach et al. and Sander note that the number of crossings determined during the crossing minimization phase is only an approximation, but give no proposals on how to solve this problem. In this report, we present algorithms that give much more accurate approximations.

Orthogonal edge routing with layers requires the vertical line segments of edges between each pair of layers to be ordered to obtain a minimal number of crossings. Eschbach et al. have shown the vertical segment ordering problem to be NP-hard for hyperedges, even if each hyperedge is constrained to have at most one vertical segment between each pair of consecutive layers [7]. They proposed two heuristics for this problem, one based on greedy assignment and one based on sifting. Sander transformed it to a cycle breaking problem on an auxiliary graph (V^*, E^*) [13]. Each node in V^* corresponds to a hyperedge, and $(h_1, h_2) \in E^*$ if h_1 and h_2 have less crossings with each other if the vertical segment of h_1 is drawn left of that of h_2 than the other way round. For instance, the hyperedge h_1 in Figure 2 has two crossings with h_2 if h_1 is drawn left of h_2, but only one crossing if h_1 is drawn right of h_2. The vertical segments can be ordered by finding a topological order for (V^*, E^*). However, as can be seen in Figure 2, this auxiliary graph may have cycles, which have to be resolved using a heuristic [4]. Note that the order and vertical positions of the nodes are fixed, since they are determined in the preceding phases of the layer-based approach. In our implementations, we use Sander's algorithm

for edge routing.

Chimani et al. replace phases 2 and 3 with *upward-planarization* methods resulting in a decreased number of crossings [3]. To incorporate port constraints and orthogonality they developed new approaches as the original methods were not applicable anymore. Contrary to this, we present small alterations to the original algorithm which can easily be added to existing implementations. It has not been evaluated how well the approach of Chimani et al. works with real-world diagrams.

Wybrow et al. employ a visibility graph to route orthogonal hyperedges between arbitrarily placed nodes [18]. They present an automatic approach that routes all hyperedges in the drawing, and a semi-automatic approach that reroutes hyperedges connected to a node moved by the user. Edge routing algorithms integrated into the layer-based approach constrain the routing of edges to the space between each pair of layers instead of routing them freely around the nodes, but this limitation will usually help improve the performance.

2 Hyperedges in the Layer-Based Approach

The layer-based approach to graph drawing supports regular edges between nodes, but does not support the concept of hyperedges right out of the box. Our general approach for representing a hyperedge thus is to replace it by regular edges, using *ports* to collect the regular edges introduced for each hyperedge incident to a given node. The major benefit of this approach is that it allows us to reuse the standard graph-based data structures and most of the algorithms employed in the layer-based drawing approach. More precisely, the first four phases (cycle elimination, node layering, crossing minimization, and node placement) can be performed with standard algorithms oblivious to hyperedges; however, as shown in Section 3 and 4, the results of the crossing minimization phase can be improved by specializing the algorithms that count crossings.

Let (V, H) be a hypergraph and $h = (S, T) \in H$ be a hyperedge; for each $v \in S$ and each $v' \in T$ we generate an edge (v, v'). We call (v, v') a *representing edge* of h and define E_h to be the set of all representing edges of h. For instance, the hyperedge h_3 in Figure 2 would be represented by three edges $(3, 4)$, $(3, 5)$, and $(3, 9)$. Furthermore, we generate a port p_w for each $w \in S \cup T$ and associate each representing edge (v, v') with the source port p_v and the target port $p_{v'}$. All edges that are connected to the same port of a node are regarded as representing the same hyperedge, which is a sufficient criterion for identifying hyperedges in our representation. The representing edges of a hyperedge may partly overlap each other in the final drawing.

Let $E = \bigcup_{h \in H} E_h$; we create a drawing for (V, H) by applying the layer-based drawing method to the graph (V, E) and then transferring the bend points of the representing edges to their associated hyperedges. Although it is possible to extend the layer-based approach such that the order of ports is respected according to predefined constraints [14], in this report we assume that the ports can be freely positioned during the layout process.

With this approach the number of representing edges of a hyperedge $h = (S, T)$ is $|S| \cdot |T|$. However, the methods we propose do not require that many representing edges. The hyperedge h may be represented by a subset $E'_h \subseteq E_h$ of representing edges. In that case, the graph $(S \cup T, E'_h)$ must be connected for each $h \in H$.

2.1 Merging Dummy Nodes

If no further measures were taken, the approach of replacing hyperedges by normal edges could lead to layouts such as the one shown in Figure 3(a), where the hyperedge

(a) Hyperedge with two dummy nodes **(b)** Hyperedge with one dummy node

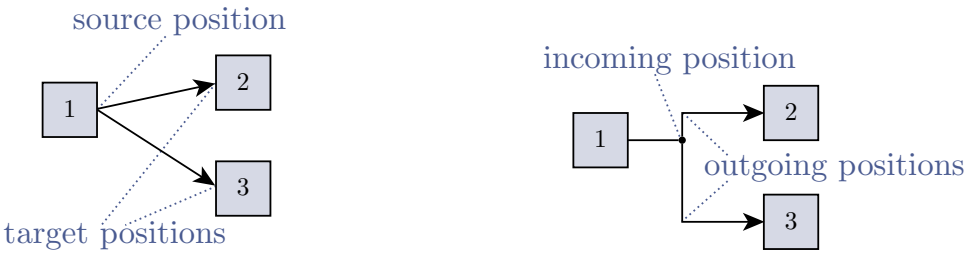
Figure 3. The hyperedge connecting nodes 1, 3, and 4 is represented by the two edges $(1, 3)$ and $(1, 4)$, split by dummy nodes d_1 and d_2. Merging them into one dummy node d' decreases the total edge length and improves readability.

(a) Merged, one crossing **(b)** Unmerged, no crossing

Figure 4. The merging of dummy nodes of long hyperedges has an influence on the number of crossings: (a) merging dummy nodes before crossing minimization leads to an unavoidable crossing, while (b) merging them afterwards can avoid the crossing.

represented by the edges $(1, 3)$ and $(1, 4)$ is assigned two dummy nodes d_1 and d_2 in the second layer in order to obtain a proper layering. As a consequence, the edges that represent the hyperedge are unnecessarily long and arguably reduce the readability of the drawing. This can be improved by merging adjacent dummy nodes that belong to the same hyperedge as shown in Figure 3(b), where the dummy nodes d_1 and d_2 were merged into d'.

It is possible to apply the merging of dummy nodes immediately after they have been created (after the node layering phase), but that prevents the crossing minimization phase from avoiding crossings caused by the hyperedges. Figure 4(a) shows an example where the crossing minimization phase cannot avoid a crossing between the two hyperedges, regardless of whether the dummy node is placed above or below node 2. If the dummy nodes were not merged until after the crossing minimization phase, the crossing could be prevented, as Figure 4(b) shows. Depending on the priority given to the aesthetic criteria of edge lengths and edge crossings, the dummy node merging algorithm should be applied either before crossing minimization (if edge lengths have a higher priority) or after (if edge crossings have a higher priority).



(a) A hyperedge with one source and two targets

(b) Orthogonal drawing with a junction point

Figure 5. Computation of bend points and junction points for orthogonally drawn hyperedges: (a) A hyperedge represented by two normal edges; (b) drawing with two bend points for each representing edge and a junction point at their common incoming position.

2.2 Junction Points

It is important to visualize the junction points of hyperedges, otherwise it can be hard (and sometimes impossible) to distinguish them from edge crossings. The computation of junction point positions can be integrated into the edge routing phase of the layout algorithm. Let e be a regular edge representing the hyperedge h, and x_h be the horizontal position assigned to the vertical line segment of h by the edge routing algorithm (see Section 1.2). Since vertical node and port positions are already fixed when the edge routing is computed, the source position y_s and target position y_t of e are known. If $y_s \neq y_t$, two bend points (x_h, y_s) (the *incoming* position) and (x_h, y_t) (the *outgoing* position) are added to e (see Figure 5). If $y_s = y_t$, the edge e does not require any bend points, but we still assign both an incoming and outgoing position at (x_h, y_s). Both of these positions are potential candidates for junction points. Let $\hat{y}_h$ be the highest and $\check{y}_h$ be the lowest vertical bend point positions of any edge that represents h. Provided that h is represented by more than one edge, we create a junction point (x_h, y) for each $y \in \{y_s, y_t\}$ if $\hat{y}_h < y < \check{y}_h$ or if h contains both an incoming and an outgoing position in y. For instance, the edge $(1, d_1)$ in Figure 3(a) has a junction point at its outgoing position because it lies between the vertical bounds $\check{y}_h$ and $\hat{y}_h$ of its hyperedge h. The edge $(2, 4)$ in Figure 3(b), in contrast, has a junction point at its incoming position because $(2, 3)$, which belongs to the same hyperedge, has an outgoing position with the same value.

Some modeling environments, e. g. Ptolemy [5], have a concept of *hypernodes* (called *relation vertices* in Ptolemy). Hypernodes are hyperedge junction points that are modeled explicitly by the user, in contrast to junction points implicitly computed by the modeling tool as described above. If it is acceptable to have the layout algorithm add or remove hypernodes, a straightforward approach to optimize their number and positions is to remove all hypernodes before layout and then create a new hypernode for each junction point computed by the algorithm. If such a modification of the model is not

(a) Without post-processing **(b)** With post-processing

Figure 6. Treating hypernodes as regular nodes can lead to unpleasant layouts since additional junction points are created. This can be improved by moving the hypernodes to one of the junction points in a post-processing step.

acceptable, the hypernodes can be regarded as regular nodes in the layout algorithm. However, this approach leads to unpleasant layouts, as Figure 6(a) shows: the hypernode v_h requires an additional layer, and for the two edges going to the third layer an additional junction point is added in the edge routing phase.

We propose a post-processing step to improve this situation by moving hypernodes such that they replace junction points that have been computed during edge routing. A hypernode v_h can have both incoming edges $E_\mathrm{i}(v_h)$ from the preceding layer and outgoing edges $E_\mathrm{o}(v_h)$ to the subsequent layer. If $|E_\mathrm{i}(v_h)| \leq 1$ and $|E_\mathrm{o}(v_h)| \leq 1$, there is no junction point to replace, so we leave v_h unchanged. Otherwise we check which side has more edges: if $|E_\mathrm{i}(v_h)| \leq E_\mathrm{o}(v_h)$, we replace the nearest junction point of the outgoing edges by v_h, otherwise we do the same with the nearest junction point of the incoming edges. An example for this procedure is shown in Figure 6(b). The additional junction point between the second and third layer has been replaced by v_h, yielding a more concise layout.

3 Counting Crossings

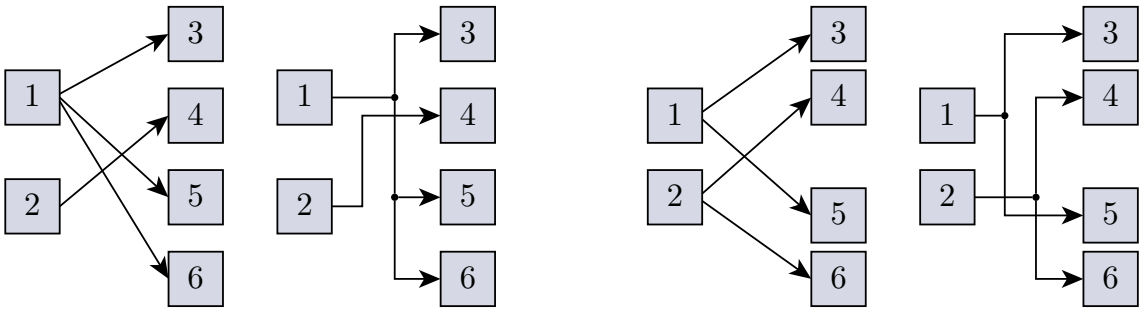
An integral part of the layer sweep heuristic for crossing minimization is an algorithm for counting crossings. Such algorithms usually assume that all edges are drawn as straight lines, which is not the case for orthogonal hyperedges. A fundamental problem with these algorithms is that the actual number of crossings does not depend only on the order of nodes in each layer, but also on the actual routing of the edges between the layers. This routing in turn depends on the concrete positions of the nodes, which is unknown at the time the crossing minimization heuristics are executed. The inevitable consequence is that those heuristics work with unreliable crossing numbers, possibly compromising the quality of their results.

Several authors have addressed the problem of counting straight-line crossings in layered graphs [2, 10, 16]. These methods always produce exact results for normal graphs. Here we call these methods STRAIGHT and denote their result as c_s. As noted by Eschbach et al. [6], there are simple examples where c_s is always different from the actual number of crossings c obtained after applying the usual orthogonal routing methods (see Figure 7). In order to quantify this difference, we measured c and c_s for a number of data flow diagrams from the Ptolemy project (see Section 4). The difference $c - c_s$ averaged -34 with a standard deviation of 190. There are some examples where the difference amounts to extreme values: one diagram with 194 hyperedges reaches $c = 269$ and $c_s = 2216$. As a general observation, the STRAIGHT methods tend to overestimate the crossing number.

For a crossing counting method to be effective it needs to accurately predict the number of crossings a given node order in two layers will produce. That is, the prediction and the result need to be tightly correlated. Standard deviations as large as the ones produced by STRAIGHT methods for orthogonal hyperedges have no tight correlation and are thus not well suited.

Since the STRAIGHT methods all compute the same number of crossings, the results of this report do not depend on which particular straight-line method is used. For our experiments we implemented the method of Barth et al. [2].

For the remainder of this chapter, we concentrate on graphs with only two layers since edges at this point of the algorithm always connect nodes in adjacent layers. It thus suffices for cross counting algorithms to count crossings between pairs of layers—the final crossing number is the sum of the crossing numbers of all pairs of adjacent layers.



(a) $c_\mathrm{s} > c$ **(b)** $c_\mathrm{s} < c$

Figure 7. The number of crossings c_s resulting from a straight-line drawing can be (a) greater or (b) less than the actual number of crossings c resulting from an orthogonal hyperedge routing.

3.1 Lower Bound Method

Since counting straight-line crossings tends to yield rather pessimistic estimates when hyperedges are involved, we assumed that a more accurate approach might be to use a lower bound of the number of crossings.

In the following, let $G = (V, H)$ be a hypergraph with a set $E = \bigcup_{h \in H} E_h$ of representing edges and two layers L_1, L_2, i. e. $V = L_1 \cup L_2$, $L_1 \cap L_2 = \emptyset$, and all $h \in H$ have their sources in L_1 and their targets in L_2. Let $\pi_1 : L_1 \to \{1, \ldots, |L_1|\}$ and $\pi_2 : L_2 \to \{1, \ldots, |L_2|\}$ be the permutations of L_1 and L_2 that result from the layer sweep heuristic for crossing minimization.

We propose an optimistic method MinOpt and denote its result as c_m. This method counts the minimal number of crossings to be expected by evaluating each unordered pair $h_1, h_2 \in H$: if any edge $e_1 \in E_{h_1}$ crosses an edge $e_2 \in E_{h_2}$ if drawn as a straight line, h_1 and h_2 are regarded as crossing each other once, denoted as $h_1 \bowtie h_2$. The result is $c_\mathrm{m} = |\{\{h_1, h_2\} \subseteq H : h_1 \bowtie h_2\}|$.

Observation 1. $c_\mathrm{m} \leq c_\mathrm{s}$.

Proof. Let $e_1, e_2 \in E$ cross each other when drawn as straight lines. There are unique $h_1, h_2 \in H$ such that e_1 represents h_1 and e_2 represents h_2. By definition of the MinOpt method, $h_1 \bowtie h_2$. Hence there is a mapping $\alpha : \{e_1, e_2 \in E : e_1 \bowtie e_2\} \to \{h_1, h_2 \in H : h_1 \bowtie h_2\}$ that is surjective because for each hyperedge crossing there is at least one crossing of representing edges. This implies $c_\mathrm{s} = |\{e_1, e_2 \in E : e_1 \bowtie e_2\}| \geq |\{h_1, h_2 \in H : h_1 \bowtie h_2\}| = c_\mathrm{m}$. □

Observation 2. *Let c be the number of hyperedge crossings in a layer-based drawing D of G. Then $c_\mathrm{m} \leq c$.*

Proof. Let $h = (S, T)$ and $h' = (S', T')$ cross each other as determined by MinOpt. Then there are $v \in S$, $w \in T$, $v' \in S'$, and $w' \in T'$ such that $(v, w), (v', w') \in E$ and (v, w), (v', w') cross each other. Without loss of generality let $\pi_1(v) < \pi_1(v')$ and

$\pi_2(w) > \pi_2(w')$. The representation $D(h)$ of h in the drawing D must connect the representations $D(v)$ and $D(w)$. This connection is not possible without crossing $D(h')$, which must connect $D(v')$ and $D(w')$, since $D(v')$ is below $D(v)$, $D(w')$ is above $D(w)$, and both $D(h)$ and $D(h')$ are inside the area between the two layers. Consequently, each crossing counted by MinOpt implies at least one crossing in D. □

Theorem 1. *Let $q = |H|$ and $H = \{h_1, \ldots, h_q\}$. The time complexity of* MinOpt *is $\mathcal{O}\left(\sum_{i=1}^{q-1} \sum_{j=i+1}^{q} |E_{h_i}| \cdot |E_{h_j}|\right)$. If $|S| = |T| = 1$ for all $(S,T) \in H$, the graph only consists of standard edges and the complexity can be simplified to $\mathcal{O}(|H|^2)$.*

Proof. The result of MinOpt is $|\{\{h_i, h_j\} \subset H : h_i \bowtie h_j\}|$, which requires to check all unordered pairs $U = \{\{h_i, h_j\} \subset H\}$. It is $|U| = |\{(i,j) \in \mathbb{N}^2 : 1 \leq i < q,\ i < j \leq q\}|$, hence $|U| = \sum_{i=1}^{q-1} \sum_{j=i+1}^{q} 1$. Whether $h_i \bowtie h_j$ is determined by comparing all representing edges of h_i with those of h_j, which requires $|E_{h_i}| \cdot |E_{h_j}|$ steps. In total we require $\sum_{i=1}^{q-1} \sum_{j=i+1}^{q} |E_{h_i}| \cdot |E_{h_j}|$ steps. If for all $h = (S,T) \in H$ the constraint $|S| = |T| = 1$ holds, we can imply $|E_h| = 1$. In this case the number of steps is $\sum_{i=1}^{q-1} \sum_{j=i+1}^{q} 1 \leq q^2$, hence the complexity is $\mathcal{O}(q^2) = \mathcal{O}(|H|^2)$. □

3.2 Approximating Method

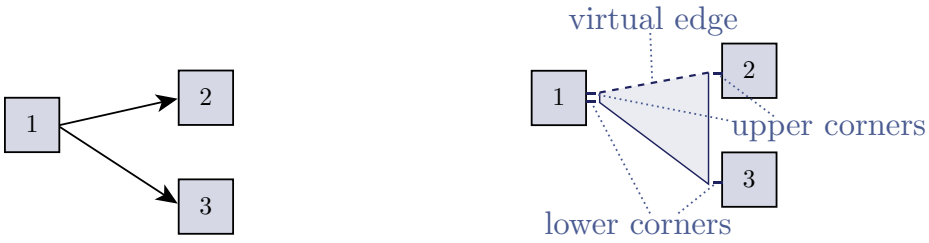
Theorem 1 shows that MinOpt has a time complexity quadratic in the number of hyperedges. In this section we propose a second method with better time complexity, which we call ApproxOpt. The basic idea is to approximate the result of MinOpt by checking three criteria explained below, hoping that at least one of them will be satisfied for a given pair of hyperedges if they cross each other in the final drawing.

Let again $G = (V, H)$ be a hypergraph with layers L_1, L_2 and π_1 and π_2 be the permutations of L_1 and L_2. We denote the result of ApproxOpt as c_a.

The ApproxOpt method is based on the four *corners* of a hyperedge: for each $h = (V_{h,1}, V_{h,2}) \in H$ and $i \in \{1,2\}$, we define the upper corners $\kappa_i^{\uparrow}(h) = \min\{\pi_i(v) : v \in V_{h,i}\}$ and the lower corners $\kappa_i^{\downarrow}(h) = \max\{\pi_i(v) : v \in V_{h,i}\}$ (see Figure 8). We associate each hyperedge with a *virtual edge* between its upper corners, $E^* = \{(\kappa_1^{\uparrow}(h), \kappa_2^{\uparrow}(h)) : h \in H\}$. The method consists of three steps:

1. Compute the number of straight-line crossings caused by virtual edges between the upper corners.
2. Compute the number of overlaps of ranges $[\kappa_1^{\uparrow}(h_1), \kappa_1^{\downarrow}(h_1)]$ and $[\kappa_1^{\uparrow}(h_2), \kappa_1^{\downarrow}(h_2)]$ in the first layer for all $h_1, h_2 \in H$.
3. Compute the number of overlaps of ranges $[\kappa_2^{\uparrow}(h_1), \kappa_2^{\downarrow}(h_1)]$ and $[\kappa_2^{\uparrow}(h_2), \kappa_2^{\downarrow}(h_2)]$ in the second layer for all $h_1, h_2 \in H$.

The result c_a is the sum of the three numbers computed in these steps. A more detailed description is given in Algorithm 1.



(a) A hyperedge represented by two normal edges

(b) Upper and lower corners

Figure 8. Illustration of the four *corners* defined for a hyperedge and the virtual edge between the two upper corners. Here $\kappa_1^{\uparrow}(h) = 1$, $\kappa_1^{\downarrow}(h) = 1$, $\kappa_2^{\uparrow}(h) = 1$, and $\kappa_2^{\downarrow}(h) = 2$ (note that corners refer to node permutations, not to node labels).

Step 1 aims at "normal" crossings of hyperedges such as h_1 and h_2 in Figure 9. The hyperedge corners used in Steps 2 and 3 serve to check for overlapping areas, as shown in Figure 9(c). For instance, the ranges spanned by h_4 and h_5 overlap each other both in the first layer and in the second layer. This is determined using a linear pass over the hyperedge corners, which are sorted by their positions. The sort keys are constructed such that the overlapping of two ranges is counted only if it actually produces a crossing:

- Corners with equal position are locally sorted by their opposite corners (second entry in the sort key). In Figure 9(c), the lower left corner of h_1 and the upper left corner of h_5 both are at position 2. But since their opposite corners are at positions 6 and 8, respectively, the upper corner of h_1 is put before the upper corner of h_5, thereby preventing the ranges of those hyperedges from being regarded as overlapping.

- If two hyperedges have their upper and lower corners all at the same position, e. g. h_2 and h_3 on the left side of Figure 9(c), the third entry $\vartheta(h)$ in the sort key is applied in order to group these corners by their corresponding hyperedge, again preventing an undesired overlapping of ranges.

- The fourth entry in the sort key is -1 for upper corners and 1 for lower corners. This ensures that the upper corner is sorted before the lower corner when they are both at the same position.

The variable d is increased whenever an upper corner is found and decreased whenever a lower corner is found. This variable indicates how many ranges of other hyperedges surround the current corner position, hence its value is added to the approximate number of crossings each time a lower corner is passed.

While MinOpt counts at most one crossing for each pair of hyperedges, ApproxOpt may count up to three crossings, since the hyperedge pairs are considered independently in all three steps. Figure 10(a) shows an example where MinOpt counts a crossing and ApproxOpt counts none, while Figure 10(b) shows an example where ApproxOpt

Algorithm 1: Counting crossings with the ApproxOpt method

Input: L_1, L_2 with permutations π_1, π_2, hyperedges H with arbitrary order ϑ

// Step 1

for each $h \in H$ **do**

  Add $(\kappa_1^{\uparrow}(h), \kappa_2^{\downarrow}(h))$ to E^*

$c_\mathrm{a} \leftarrow$ number of crossings caused by E^*, counted with a straight-line method

// Steps 2 and 3

for $i = 1 \dots 2$ **do**

  for each $h \in H$ **do**

    Add $(\kappa_i^{\uparrow}(h), \kappa_i^{\downarrow}(h), \vartheta(h), -1))$ and $(\kappa_i^{\downarrow}(h), \kappa_i^{\uparrow}(h), \vartheta(h), 1))$ to C_i

  Sort C_i lexicographically

  $d \leftarrow 0$

  for each $(x, x', j, t) \in C_i$ in lexicographical order **do**

    $d \leftarrow d - t$

    if $t = 1$ **then**

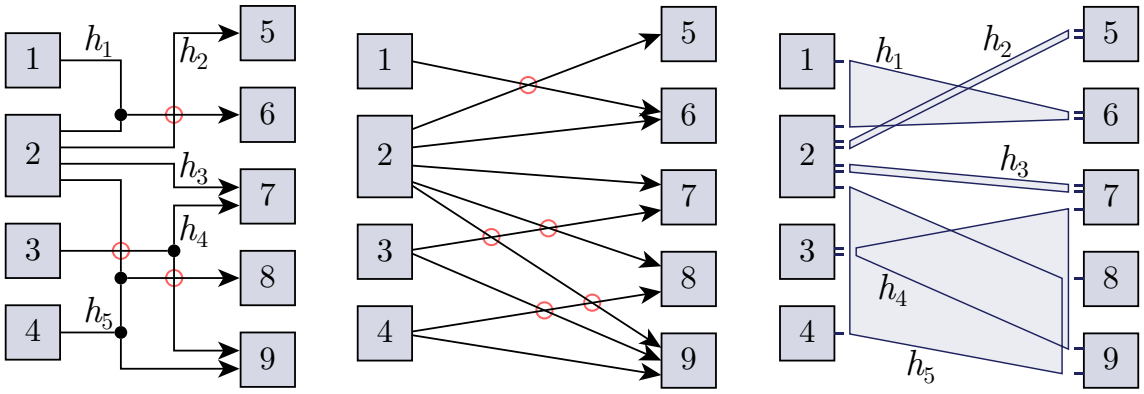
      $c_\mathrm{a} \leftarrow c_\mathrm{a} + d$

return c_a

counts a crossing and MinOpt counts none. Thus neither $c_\mathrm{m} \leq k c_\mathrm{a}$ nor $c_\mathrm{a} \leq k c_\mathrm{m}$ hold in general for any $k \in \mathbb{N}$. However, as shown in Section 4, the difference between c_m and c_a is rather small in practice.

Theorem 2. *Let* $b = \sum_{(S,T)\in H}(|S| + |T|)$. *The time complexity of* ApproxOpt *is* $\mathcal{O}(b + |H|(\log|V| + \log|H|))$.

Proof. In order to determine the corners $\kappa_i^{\uparrow}(h), \kappa_i^{\downarrow}(h)$ for each $h \in H$, $i \in \{1, 2\}$, all source and target nodes are traversed searching for those with minimal and maximal index π_i. This takes $\mathcal{O}\left(\sum_{(S,T)\in H}(|S| + |T|)\right) = \mathcal{O}(b)$ time. The number of virtual edges created for Step 1 is $|E^*| = |H|$. Counting the crossings caused by E^* can be done in $\mathcal{O}(|E^*| \log|V|) = \mathcal{O}(|H| \log|V|)$ time [2]. Steps 2 and 3 require the creation of a list C_i with $2|H|$ elements, namely the lower-index and the upper-index corners of all hyperedges. Sorting this list is done with $\mathcal{O}(|C_i| \log|C_i|) = \mathcal{O}(|H| \log|H|)$ steps. Afterwards, each element in the list is visited once. The total required time is $\mathcal{O}(b + |H| \log|V| + |H| \log|H|) = \mathcal{O}(b + |H|(\log|V| + \log|H|))$. □



(a) A hypergraph **(b)** Representing edges **(c)** Overlapping areas

Figure 9. The hypergraph (a) can be drawn orthogonally with $c = 3$ crossings. The straight-line crossing number (b) is $c_s = 5$, the result of MinOpt is $c_m = 2$, and the result of ApproxOpt is $c_a = 4$. ApproxOpt counts three crossings between h_4 and h_5 (c) because the virtual edges $(2, 8)$ and $(3, 7)$ cross (Step 1 in Algorithm 1) and the ranges spanned by the corners overlap both in the left layer and in the right layer (Steps 2 and 3).

(a) $c_m = 1$, $c_a = 0$ **(b)** $c_m = 0$, $c_a = 1$

Figure 10. Differences between the MinOpt and ApproxOpt methods: (a) $c_m = 1$ due to the crossing of $(1, 4)$ and $(2, 3)$, but $c_a = 0$ since none of the three steps of ApproxOpt is able to detect that. (b) $c_m = 0$ because $(2, 4)$ crosses neither $(1, 4)$ nor $(3, 4)$; $c_a = 1$ because one crossing is detected in Step 2 of Algorithm 1.

4 Experimental Evaluation

Ptolemy diagrams. The Ptolemy open source project [5] contains a large number of models for testing and demonstration in its repository.[1] Ptolemy allows models to be nested using *composite actors* that represent subsystems composed of other actors. Since nested models are usually quite small, the evaluation of the methods presented in the previous sections was performed on a transformed variant of the Ptolemy demonstration models where all composite actors were flattened. This was done by moving their contained actors to the outer hierarchy level and eliminating the composite actors. 171 of the so obtained flattened data flow diagrams were selected for the evaluation. Diagrams unsuitable for evaluations were left out, e. g. those with very few nodes.

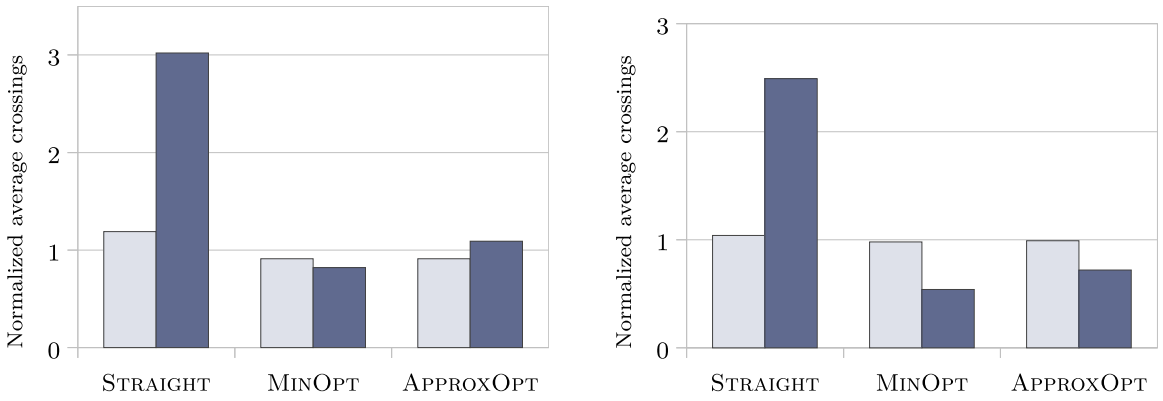
Figure 1 shows two flattened Ptolemy diagrams drawn with the approaches presented in Section 2 for merging dummy nodes and finding junction points. Note that the drawing in Figure 1(a) has been done with port positioning constraints, which require further extensions of the layer-based drawing method [14]. Furthermore, the diagrams contain *multiports*, which are used to form arrays of signals connected through multiple hyperedges. Multiports are drawn as white arrowheads, whereas normal ports are drawn as black arrowheads. However, multiports were treated as normal ports in our experiments, hence all edges connected to the same multiport were regarded as being part of the same hyperedge.

We executed our drawing algorithm once for each crossing counting algorithm on each of the selected Ptolemy diagrams. For each execution, the actual number of crossings in the final diagram as well es the number predicted by the cross counting algorithm were measured. The results can be seen in Figure 11(a). The important observation is that the average number of actual crossings is reduced by 23.6% when using MinOpt and by 23.8% when using ApproxOpt instead of Straight. These differences of mean values are significant: the p-values resulting from a t-test[2] with paired samples are 4.5% for MinOpt and 4.0% for ApproxOpt.

A more detailed view on the experimental results is shown in Table 1. The average results of the three counting methods are given for each of the three executions, even if they have not been used in the layer sweep heuristic for crossing minimization during that execution. The table reveals that the accuracy of the counted number of crossings, $|c - c_\mathrm{m}|$ and $|c - c_\mathrm{a}|$, is consistently better with the two methods proposed here compared to the accuracy $|c - c_\mathrm{s}|$ obtained with the straight-line method. This does not only apply when comparing the mean values of these differences, but also their standard

[1] `http://ptolemy.eecs.berkeley.edu`

[2] In short, a t-test determines the probability p that test results are mere coincidence. Results with $p \leq 5\%$ are generally considered significant.



(a) Ptolemy graphs, $\bar{c} \approx 18.75$ **(b)** Random graphs, $\bar{c} \approx 1628$

Figure 11. Average number of crossings when the crossing minimization phase uses the given crossing counting algorithm (light), and average number of crossings predicted by that algorithm (dark).

deviations: the Straight method leads to more extreme difference values (cf. the observation mentioned in the beginning of Section 3). Furthermore, the difference $|c_\mathrm{m} - c_\mathrm{a}|$ of the results of the MinOpt and ApproxOpt methods is relatively low, averaging about a third of the total crossing number. This confirms that ApproxOpt yields a good approximation of MinOpt.

The layer sweep heuristic for crossing minimization uses the predicted number of crossings only to compare two possible node orderings with each other. Therefore the predicted values as such are not relevant in this context, but rather their comparison: given two node orderings π_1, π_2, corresponding predictions $c_{\mathrm{p},1}, c_{\mathrm{p},2}$, and actual numbers of crossings c_1, c_2, a good prediction must meet

$$\sigma(c_{\mathrm{p},1} - c_{\mathrm{p},2}) = \sigma(c_1 - c_2) \ , \tag{4.1}$$

where $\sigma : \mathbb{R} \to \{0, 1, -1\}$ is the sign operator. With the Straight prediction Equation 4.1 is met in 55% of the cases comparing the values obtained in the three algorithm executions for each graph, which lead to three comparisons per graph (Executions E_s / E_m, E_s / E_a, and E_m / E_a). MinOpt and ApproxOpt performed correctly in 65% and 72% of the comparisons, respectively. These drastic improvements of the ratio of correct comparisons (p-values $< 10^{-6}$ with a t-test) in the context of the layer sweep heuristic explain why the two proposed methods lead to fewer crossings in the actual drawings compared to the straight-line method.

More details on the correctness of comparisons are given in Table 3. It can be seen that the correctness rates are extremely different depending on which execution results are compared and which subset of graphs is considered. Each comparison involves two node orderings π_1, π_2 and actual numbers of crossings c_1, c_2. In general, when constrained to graphs where $c_1 < c_2$, each prediction method M yields high correctness rates if π_1 was determined based on M, but low correctness rates if π_2 was determined based on M. If $c_1 > c_2$ an inverse tendency is observed. For instance, when comparing results made

with the Straight method (Execution E_s) with results made with the MinOpt method (Execution E_m), Straight has a correctness rate of 74% for graphs with $c_1 < c_2$ (i. e., where the Straight method yields better results than the MinOpt method), but only 8% for graphs with $c_1 > c_2$. This difference is not surprising; the better the drawing created with a particular method is, the higher is the probability that the predictions made by that method were correct.

Random graphs. We performed a second experiment with randomly generated bipartite graphs with 5 to 100 nodes and 2 to 319 hyperedges each. The algorithms for counting crossings always operate on pairs of consecutive layers, which are bipartite subgraphs, hence the specialization of the experiment to bipartite graphs is valid. We performed the same measurements as for the Ptolemy diagrams, the results of which are shown in Figure 11(b). They confirm the general observations made before. The average number of actual crossings is reduced by 5.6% when using MinOpt and by 4.6% when using ApproxOpt instead of Straight. Although the relative difference of mean values is lower compared to the Ptolemy diagrams, their significance is much higher: in both cases $p < 10^{-31}$. The observations stated for the results shown in Table 1 also apply to the measurements made with the random graphs, of which mean values and standard deviations are shown in Table 2.

With respect to Equation 4.1, 34% of the comparisons made with the Straight prediction were correct, while MinOpt and ApproxOpt performed correctly in 71% and 68% of the cases, respectively. It is worth noting that the rate of correct comparisons with the straight-line method is very close to the expected result of a function randomly choosing between 0, 1, and -1, which would make a correct decision in 33% of the cases. More details are presented in Table 4.

Execution time. Performance evaluations conducted on a separate set of 100 randomly generated large bipartite graphs (500 nodes, 3 edges per node) confirmed our theoretical results: Straight (mean time 0.3ms) was significantly faster than ApproxOpt (1.7ms), which in turn was significantly faster than MinOpt (24ms).

Table 1. Average values measured for the flattened Ptolemy diagrams, with standard deviations in brackets. All three methods for counting crossings (c_s, c_m, and c_a) were measured in all three executions, but each execution used only one method for minimizing crossings: c_s in Execution E_s, c_m in Execution E_m, and c_a in Execution E_a. The last column shows the average values over all three executions. All values are normalized by the total average number of crossings $\bar{c} \approx 18.75$.

Variable	Execution E_s (using c_s)		Execution E_m (using c_m)		Execution E_a (using c_a)		Total	
c	1.19	[4.77]	0.91	[3.65]	0.91	[3.95]	1.00	[4.14]
c_s	3.02	[13.94]	3.63	[16.48]	3.66	[16.86]	3.44	[15.79]
c_m	1.12	[4.24]	0.82	[3.33]	0.85	[3.62]	0.93	[3.75]
c_a	1.46	[5.52]	1.17	[4.81]	1.09	[4.77]	1.24	[5.04]
$\lvert c - c_\mathrm{s}\rvert$	1.90	[10.13]	2.79	[13.49]	2.78	[13.61]	2.49	[12.50]
$\lvert c - c_\mathrm{m}\rvert$	0.29	[0.69]	0.28	[0.63]	0.26	[0.63]	0.28	[0.65]
$\lvert c - c_\mathrm{a}\rvert$	0.33	[1.01]	0.36	[1.28]	0.31	[0.94]	0.33	[1.09]
$\lvert c_\mathrm{m} - c_\mathrm{a}\rvert$	0.35	[1.47]	0.34	[1.49]	0.26	[1.17]	0.32	[1.38]

Table 2. Average values measured for the random bipartite graphs, with standard deviations in brackets. The table has the same format as Table 1. All values are normalized by the total average number of crossings $\bar{c} \approx 1628$.

Variable	Execution E_s (using c_s)		Execution E_m (using c_m)		Execution E_a (using c_a)		Total	
c	1.04	[2.00]	0.98	[1.93]	0.99	[1.95]	1.00	[1.96]
c_s	2.49	[4.43]	2.67	[4.68]	2.68	[4.69]	2.61	[4.60]
c_m	0.58	[1.28]	0.54	[1.23]	0.55	[1.24]	0.55	[1.25]
c_a	0.79	[1.69]	0.74	[1.63]	0.72	[1.59]	0.75	[1.64]
$\lvert c - c_\mathrm{s}\rvert$	1.46	[3.31]	1.69	[3.65]	1.69	[3.64]	1.61	[3.54]
$\lvert c - c_\mathrm{m}\rvert$	0.46	[0.81]	0.44	[0.78]	0.44	[0.78]	0.45	[0.79]
$\lvert c - c_\mathrm{a}\rvert$	0.24	[0.47]	0.24	[0.44]	0.27	[0.48]	0.25	[0.46]
$\lvert c_\mathrm{m} - c_\mathrm{a}\rvert$	0.22	[0.43]	0.20	[0.41]	0.17	[0.36]	0.19	[0.40]

Table 3. Rate of correctness determined with Equation 4.1 applied to the results of the three executions on Ptolemy diagrams. The execution results are compared in pairs E_s / E_m, E_s / E_a, and E_m / E_a, where Execution E_s used c_s to determine a node order, Execution E_m used c_m, and Execution E_a used c_a. The correctness rates for each prediction method are presented in four rows: three rows constrained to graphs that meet specified criteria and one row showing total average rates for all graphs. The criteria are given in the form $c_1 \sim c_2$, where c_1 and c_2 are the actual numbers of crossing resulting from the first and second compared executions, respectively.

		Exec. E_s / E_m (using c_s/c_m)	Exec. E_s / E_a (using c_s/c_a)	Exec. E_m / E_a (using c_m/c_a)	Total
Straight	$c_1 < c_2$	74 %	82 %	79 %	
	$c_1 > c_2$	8 %	4 %	32 %	
	$c_1 = c_2$	82 %	62 %	73 %	
	Total	64 %	42 %	58 %	55 %
MinOpt	$c_1 < c_2$	46 %	66 %	73 %	
	$c_1 > c_2$	76 %	32 %	28 %	
	$c_1 = c_2$	90 %	79 %	91 %	
	Total	77 %	56 %	63 %	65 %
ApproxOpt	$c_1 < c_2$	67 %	34 %	12 %	
	$c_1 > c_2$	82 %	92 %	91 %	
	$c_1 = c_2$	88 %	58 %	73 %	
	Total	82 %	67 %	68 %	72 %

Table 4. Rate of correctness determined with Equation 4.1 applied to the random bipartite graphs. The table has the same format as Table 3.

		Exec. E_s / E_m (using c_s/c_m)	Exec. E_s / E_a (using c_s/c_a)	Exec. E_m / E_a (using c_m/c_a)	Total
Straight	$c_1 < c_2$	97 %	100 %	59 %	
	$c_1 > c_2$	1 %	0 %	37 %	
	$c_1 = c_2$	66 %	62 %	70 %	
	Total	24 %	28 %	51 %	34 %
MinOpt	$c_1 < c_2$	7 %	31 %	88 %	
	$c_1 > c_2$	98 %	89 %	21 %	
	$c_1 = c_2$	73 %	70 %	81 %	
	Total	79 %	75 %	59 %	71 %
ApproxOpt	$c_1 < c_2$	21 %	5 %	18 %	
	$c_1 > c_2$	93 %	100 %	84 %	
	$c_1 = c_2$	67 %	64 %	71 %	
	Total	78 %	76 %	50 %	68 %

5 Conclusion

In this report, we described how to integrate orthogonal hyperedges into the layer-based approach to graph drawing by replacing hyperedges with representing edges. We proposed a method to handle hyperedges that span multiple layers and a method to compute junction points. We proposed two methods for counting crossings in orthogonal hypergraph drawings more accurately. Our experiments indicate that the algorithms lead to significantly fewer edge crossings both with real-world and with random diagrams.

We see two main areas for future research. First, the number of crossings between orthogonal hyperedges depends not only on the results of the crossing minimization, but also on the exact placement of nodes. However, current node placement algorithms only try to minimize either edge length or the number of bend points. And second, limiting the routing of each hyperedge to one horizontal segment reduces the number of bend points at the expense of edge crossings. Future research could address routing algorithms that reduce the number of edge crossings as well by creating multiple horizontal segments.

Bibliography

[1] Danil E. Baburin. Using graph based representations in reengineering. *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 203–206, 2002.

[2] Wilhelm Barth, Petra Mutzel, and Michael Jünger. Simple and efficient bilayer cross counting. *Journal of Graph Algorithms and Applications*, 8(2):179–194, 2004.

[3] Markus Chimani, Carsten Gutwenger, Petra Mutzel, Miro Spönemann, and Hoi-Ming Wong. Crossing minimization and layouts of directed hypergraphs with port constraints. In *Proceedings of the 18th International Symposium on Graph Drawing (GD'10)*, volume 6502 of *LNCS*, pages 141–152. Springer, 2011.

[4] Peter Eades, Xuemin Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[5] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[6] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Crossing reduction for orthogonal circuit visualization. In *Proceedings of the 2003 International Conference on VLSI*, pages 107–113. CSREA Press, 2003.

[7] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Orthogonal hypergraph drawing for improved visibility. *Journal of Graph Algorithms and Applications*, 10(2):141–157, 2006.

[8] Lars Kristian Klauske. *Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus*. PhD thesis, Technische Universität Berlin, 2012.

[9] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):231–260, 2003.

[10] Hiroshi Nagamochi and Nobuyasu Yamada. Counting edge crossings in a 2-layered drawing. *Information Processing Letters*, 91(5):221–225, 2004.

[11] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 248–261. Springer, 1997.

[12] Georg Sander. A fast heuristic for hierarchical Manhattan layout. In *Proceedings of the Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 447–458. Springer, 1996.

[13] Georg 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.

[14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on on Diagram Aesthetics and Layout*, 25(2):89–106, 2014.

[15] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.

[16] Vance Waddle and Ashok Malhotra. An $E \log E$ line crossing algorithm for levelled graphs. In *Proceedings of the 7th International Symposium on Graph Drawing (GD'99)*, volume 1731 of *LNCS*, pages 59–71. Springer, 1999.

[17] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.

[18] Michael Wybrow, Kim Marriott, and Peter J. Stuckey. Orthogonal hyperedge routing. In *Proceedings of the 7th International Conference on Diagrammatic Representation and Inference (Diagrams'12)*, volume 7352 of *LNAI*, pages 51–64. Springer, 2012.