Hierarchy-Aware Layer Sweep

Alan Schelten

Master's Thesis
2016

Prof. Dr. Reinhard von Hanxleden
Real Time Systems and Embedded Systems
Department of Computer Science
Kiel University

Advised by
M. Sc. Ulf Rüegg

Abstract

Data flow diagrams with modules can be modeled as hierarchical port-based layered graphs. Here, nodes are connected to each other using ports. A module is modeled as a hierarchical node—a node which contains a child graph. Edges incident to a port show the data flowing in and out of the module at this port. Layered graph layout places the nodes on layers and attempts to route as many edges as possible in a left-to-right direction. Minimizing the number of edge crossings is considered an important goal for aesthetic quality and readability of the diagram.

Previous methods of crossing minimization in hierarchical graphs have followed a bottom-up strategy. Here, the layout of each child graph is completed before the layout of the graph containing its parent. This often leads to many unnecessary edge crossings.

Crossing minimization in non-hierarchical layered graphs often uses the two-layer sweep method, which visits each pair of neighboring layers, fixes the order of one layer and permutes the order of the other while minimizing edge crossings. Hierarchy-Aware Layer Sweep (HALS), proposed in this thesis, extends this principle to hierarchical graphs, sweeping in and out of each hierarchical graph. While this principle can decrease the number of crossings in some graphs, in others it leads to an increased number of edge crossings and slow running time. This is improved using a heuristic which switches between the use of bottom-up and HALS for each graph depending on its characteristics.

An evaluation on real-world and random datasets shows that the number of edge crossings is reduced in most graphs up to a median improvement of 63% for one of the datasets, while the running time of the crossing minimization phase worsens up to an average increase in visited nodes of 40%. The amount of improvement and slowdown depends heavily on the characteristics of the graph.

As further enhancements to the area of crossing minimization in layered graphs, this thesis presents a simple algorithm which efficiently counts crossings between in-layer edges (where both ends are in the same layer) and between-layer edges (where the ends are in neighboring layers) and an efficient heuristic for sorting ports on nodes where some of the ports have a fixed port order and others can be sorted freely.

Contents

Chapter 1

Introduction

1.1 Automatic Layout of Hierarchical Graphs

Automatic layout of graphs is used in a wide variety of fields, ranging from VLSI design over visualization of social networks to railway maps and data flow diagrams. We shall concentrate on the latter case, where often the preferred mode of representation uses *layered graphs* which place the nodes on layers and strives to orient most edges left-to-right. For an example of such a graph, see Figure 1.1. Hierarchical graphs (graphs where nodes contain graphs) can be used in any of these applications. In the context of data flow diagrams, they can show modularization, where the data flowing into and out of a module is shown by edges into the hierarchical node and the graph contained within the node shows the detailed implementation of the module. This allows for tools where details can be shown or hidden with a click of the mouse. An example can be seen in Figure 1.2. Automatic layout of hierarchical graphs can lead to a significant increase of complexity compared to dealing with simple graphs.
A prominent method for automatic layout of layered graphs is a framework developed by Sugiyama, Tagawa and Toda [STT81]. Split into different phases, it distributes the nodes to layers and in a crossing minimization phase permutes the order of the nodes to minimize

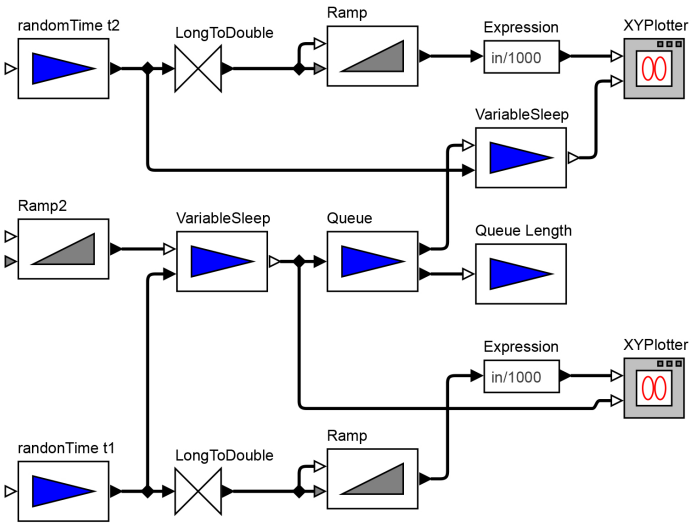


Figure 1.1. Example of a data flow diagram laid out in layers.

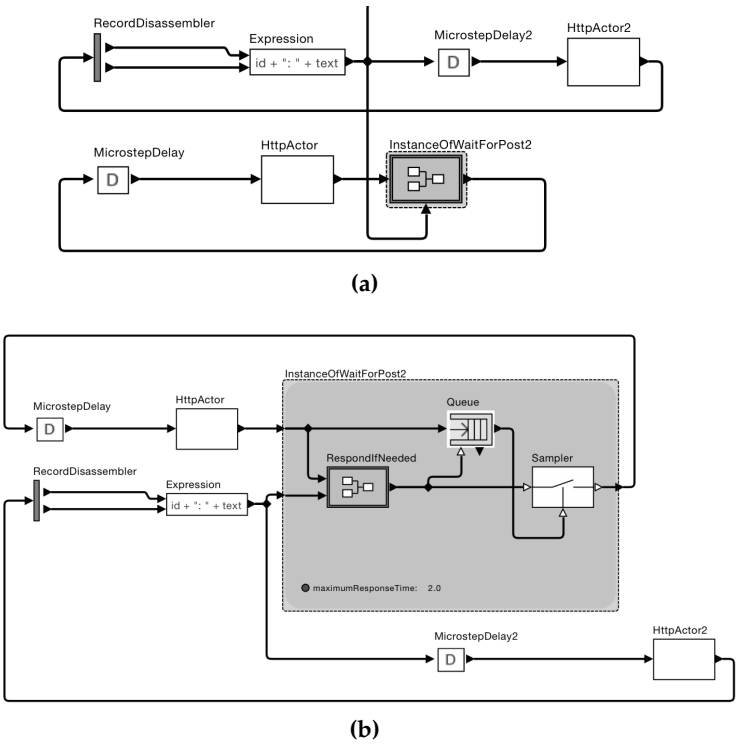


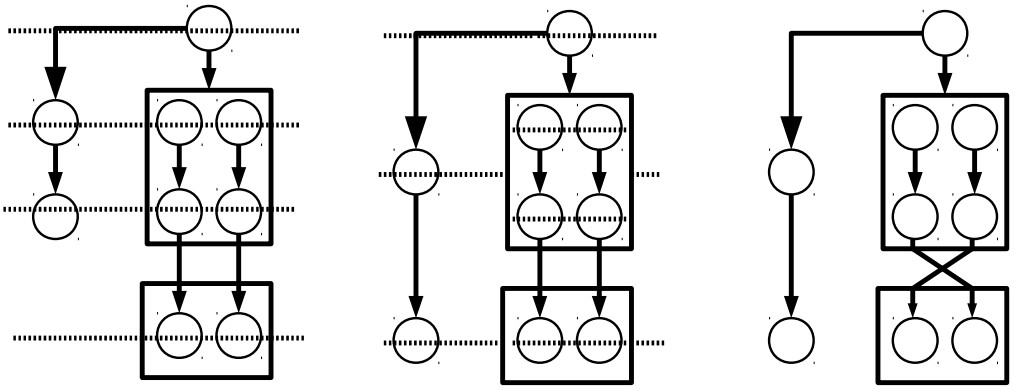
Figure 1.2. (a) shows the hierarchical node `InstanceOfWaitForPost2` collapsed. (b) shows it expanded, revealing the child graph.

edge crossings. When extending this framework to hierarchical graphs, there are in general three different approaches, *Bottom-Up (*BU), *global layering* and *local layering*. For an intuitive understanding of the differences see Figure 1.3.

The simplest method is BU, where the graphs in the innermost hierarchical nodes are laid out first and separately from the rest of the graph. The positions the edges connect to the outside of each hierarchical node, i. e., the positions of what are called the *ports*, are then fixed according to the layout of its inner graph and the hierarchy level above it is laid out next. This method needs almost no change in the Sugiyama framework. However, the layout of the graphs within hierarchical nodes is not influenced by the parent graph in any way. This can lead to many unnecessary edge crossings, causing layouts as in Figure 1.3c.

In the case of global layering, the nodes of the parent and child graphs are assigned to layers together. Here, any hierarchical node can span several of these global layers. During the crossing minimization phase, the child nodes must be constrained to their parent nodes.

In local layering, the layering phase stays generally the same as for simple graphs, placing normal nodes and hierarchical nodes in the same layer and placing the nodes of the child graphs on separate layers. In contrast to BU, the crossing minimization phase can still traverse the hierarchy while taking all edges into account.



(a) Global layering: The dashed lines show the global layers.

(b) Local layering: Each hierarchical node has its own set of layers.

(c) BU layout: Since the subgraphs are laid out before the parent graphs and without taking them into account, crossings such as the one shown can occur.

Figure 1.3. Global layering, local layering and bottom-up

The Eclipse Layout Kernel (ELK)[1] project provides an open source solution for automatic graph layout, and incorporates a wide range of features and possibilities including different types of graph layout algorithms. A popular algorithm of the project is ELK Layered, which is based on the approach originally suggested by Sugiyama et al.

For hierarchical nodes, ELK Layered currently uses BU, resulting in many unnecessary crossings. A successful implementation of global layering was evaluated in the context of ELK Layered (see Fuhrmann [Fuh12]), however this implementation increases computation time and more significantly, leads to an increased maintenance overhead. This is because changes to the layout algorithm for simple graphs must be implemented a second time for the code concerning the hierarchical graphs.

The local approach mainly changes the crossing minimization phase. Minimizing the number of crossings between edges of the graph is generally viewed as an important goal to increase the readability and aesthetic quality of the graph. Unfortunately, this problem has been shown to be NP-complete [EW94]. To minimize crossings many suggested algorithms use the *layer sweep* method: Sweep through the graph and compare neighboring layers, assuming the order of nodes to be fixed in one layer and permuting the order of nodes in the other layer with the goal of minimizing edge crossings. Even when reduced to this task of minimizing crossings by ordering only one part of a bipartite graph, the problem remains NP-complete [EW94]. Most research has concentrated on this simplified case, and a large number of papers have been published, suggesting many heuristics and some optimal algorithms.

[1] `http://www.eclipse.org/elk/`, accessed 2016-08-24

1.2 Contributions

This thesis examines a local layering approach for layout of hierarchical graphs and its implementation into the ELK Layered framework. One of the implementation's main goals is to keep maintenance as easy as possible, ensuring that no other parts of the framework need to be changed except for those explicitly accessing the complete hierarchy. The other parts of the algorithm that previously worked on simple graphs stay the same.

As noted above, a local layering approach must mainly deal with the crossing minimization phase. To be able to apply this method on hierarchical graphs, the layer sweep method is adapted, creating the Hierarchy-Aware Layer Sweep (HALS). Here, the sweep continues into the local layers of each child graph, thereby manipulating the complete hierarchy of the graph.

At first glance, this seems to be a simple and effective solution to the unnecessary edge crossings arising from the use of BU. However, HALS has a disadvantage compared to BU: While BU is a divide-and-conquer approach, HALS is not. Due to the non-determinism in the crossing minimization heuristic used in ELK Layered, this can lead to more crossings and slower run-times.

To alleviate this problem, we propose a heuristic which decides to use BU or HALS for each part of the inclusion tree separately, where the inclusion tree is a tree showing the hierarchy relation of the nodes, i. e., where the children of a node v in the inclusion tree are the nodes in the graph contained within v. To do this, it compares the influence of hierarchical edges and of random placement of nodes. Figure 1.4 shows a number of examples, where HALS leads to improved layouts.

This approach is evaluated using three different datasets: A set of random graphs, a set of graphs from the Ptolemy project[2] and a set of Sequentially Constructive Graph (SCG) with basic blocks [vHDM+14]. The experiments show that there is a default setting for the heuristic, such that the graphs with improved or unchanged crossing number are in the majority and the average number of crossings decreases in all cases. However, the algorithm is much more successful for specific types of graphs, especially those with few simple nodes in the child graphs. The most striking example of these are the SCGs. An example of this can be seen in Figure 1.4c and the improvement in Figure 1.4f.

Since counting crossings is a major part of crossing minimization, we describe a simple and efficient algorithm for counting in-layer edge crossings and between-layer edge crossings at the same time. Here, in-layer edges are edges starting and ending on nodes in the same layer, and between-layer edges are edges starting and ending on nodes in neighboring layers. For the former, to the best of my knowledge, no efficient algorithm for counting crossings has previously been proposed.

Finally, we shortly discuss an algorithm dealing with sorting ports for crossing minimization on nodes where some of the ports have a fixed port order and others can be sorted freely.

[2] See `Ptolemy.eecs.berkeley.edu`, accessed 16/05/27

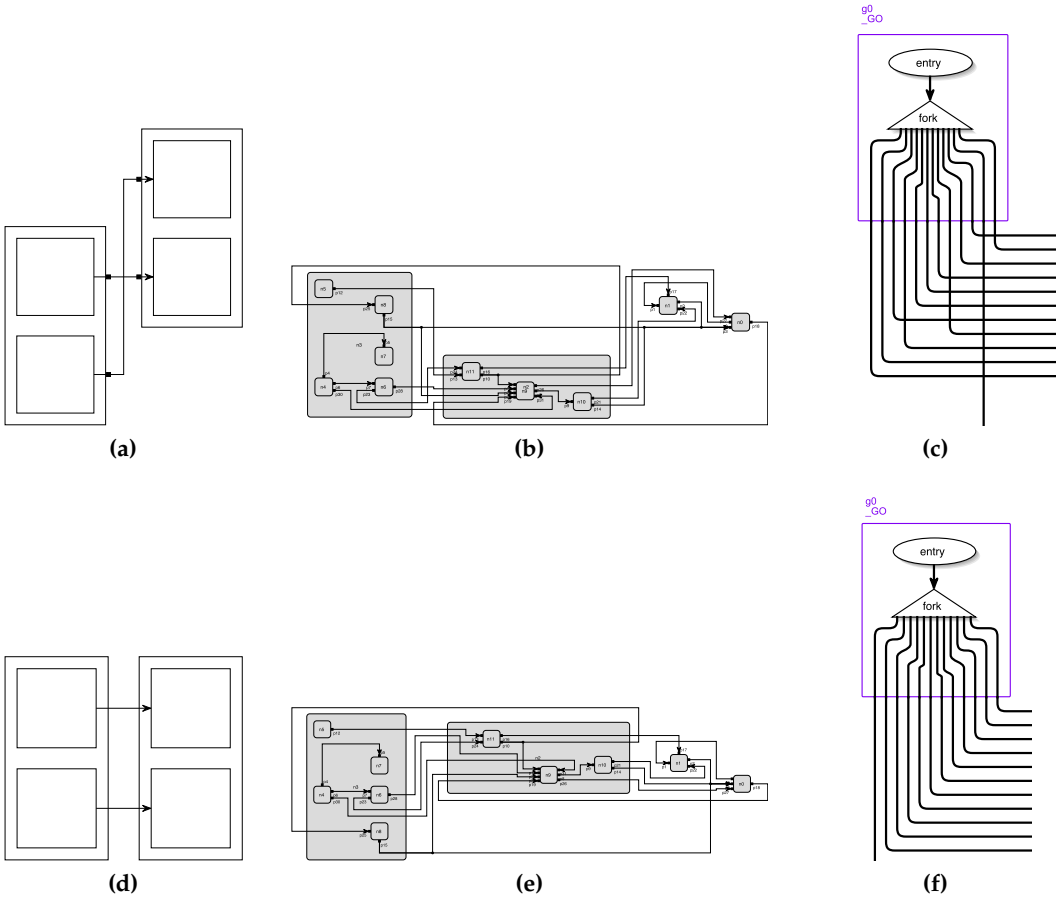


Figure 1.4. Examples of unnecessary crossings when using BU and their resolution with HALS. (a) shows a trivial example. Since the port order is fixed, there is no way to remove the crossing once the layout of the inner graphs is completed. (b) shows an example of a random graph where the same effect can be seen on a less trivial example. (c) shows part of an SCG with basic blocks (see Section 6.1.1) showing the large number of crossings which can happen when there are very few simple nodes with very many hierarchical edges. (d), (e) and (f) show the same examples with the crossings caused by BU removed using HALS.

1.3 Outline

To ensure the understanding of the basic concepts necessary for the rest of the thesis Chapter 2 explains preliminary definitions, the basic principles of the Sugiyama Framework and the current crossing minimization technique.

This information is then used to give a literature overview in Chapter 3. Here, previous work on hierarchical graphs, crossing minimization and port sorting is presented and reviewed.

Chapter 4 then elaborates the main contributions, explaining HALS as well as the challenges which arise from the main idea. The heuristic addressing this issue is introduced. Furthermore, Chapter 4 presents two other contributions: Firstly, a simple and efficient algorithm for counting in-layer and between-layer crossings at the same time. And secondly, a fast method for inserting ports with no port constraints into a list of ports with fixed order.

Chapter 5 describes the main design principles of ELK Layered and the design decisions made for integrating a crossing minimization method which applies to the complete hierarchy.

This implementation is then evaluated on two real-world datasets and different sets of randomly generated graphs in Chapter 6. Here, we examine the success and running time of HALS as well as different graph characteristics influencing the quality of the results.

Finally, Chapter 7 summarizes the results and gives an outlook on further research avenues related to the contributions of this thesis.

Chapter 2

Preliminaries

This chapter addresses the preliminary information necessary for understanding the rest of this thesis. First, we take a look at the terminology and definitions used in the following chapters. The next section introduces the layered graph layout algorithm of which the crossing minimization is one of the main phases. Finally, the current method of dealing with hierarchical graphs is explained.

2.1 Terminology

Definitions and terms used across this thesis are collected here, so should the meaning of a particular word or symbol be forgotten, this is the place to look. The definitions are partly taken from those used in my bachelor's thesis [Sch15] and by Sugiyama et al. [SM91]. An overview over the graph elements is given in Figure 2.1.

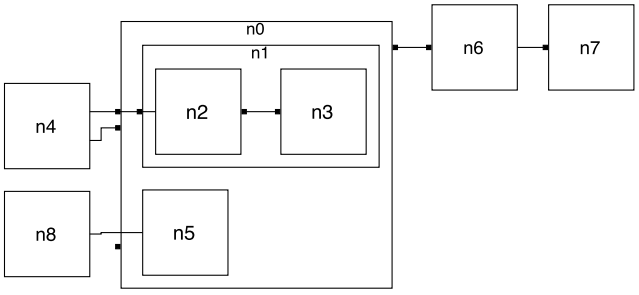
Definition 1 (Simple Graph). A *simple graph* $G_a = (V, E, P, vp)$ contains the following elements: V is the set of *nodes* and P the set of *ports*. Let $D = \{n, e, s, w\}$ (D as in **D**irection) be the set of sides a port can be on, where the letters n, e, s, w in the set D stand for *north, east, south* and *west*. Using these, the function $vp : P \to V \times D$ maps ports to nodes and the side of the node the port is on. To simplify the algorithms in the following chapters, we will be using graphs with undirected *edges* connecting ports $E = \{p_1, p_2\},\ p_1, p_2, \in P$. Even though data flow graphs obviously use directed edges, the direction of the edges is not of importance for this thesis.

Definition 2 (Inclusion Tree). The hierarchy relation of the nodes is described by the *inclusion tree* $T_i = (V, F, r)$. Here, $F = (v_1, v_2),\ v_1, v_2, \in V$ are the directed *inclusion edges* connecting nodes. r is an artificial root node which is not drawn in the final graph. T_i forms a tree rooted in r. In this tree, parent, ancestors, children, and descendants of node v are denoted by $Pa(v)$, $An(v)$, $Ch(v)$ and $De(v)$. Note that then in all cases $An(r) = \varnothing$.

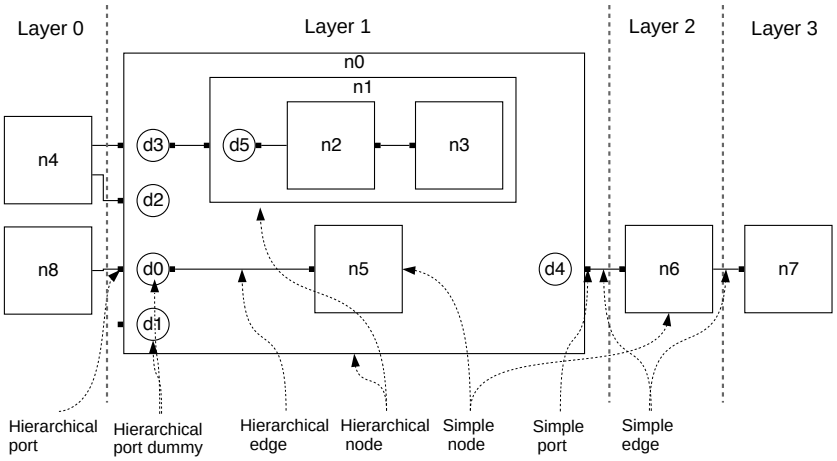
Definition 3 (Hierarchical Graph). A *hierarchical graph* is a tuple $G = (V, E, F, r, P, vp)$. This contains the adjacencies in the simple graph $G_a = (V, E, P, vp)$ and the inclusion tree $T_i = (V, F, r)$.

Definition 4 (Simple Node). A simple node is a node with no children in the inclusion tree, i. e.,

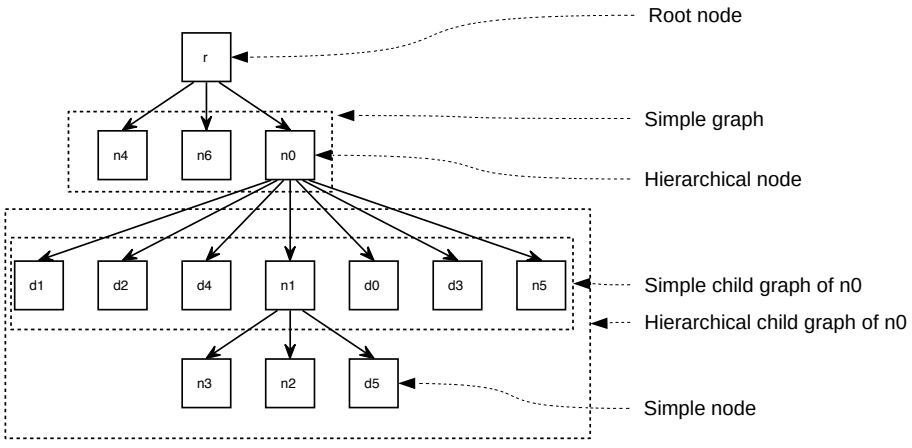
$$v \in V \text{ is simple } \Leftrightarrow Ch(v) = \varnothing.$$



(a) Input graph.



(b) Hierarchical graph.



(c) Inclusion tree.

Figure 2.1. Overview over the graph elements.

Definition 5 (Hierarchical Node). A hierarchical node is a node which is a parent in the inclusion tree, i. e.,

$$v \in V \text{ is hierarchical } \Leftrightarrow Ch(v) \neq \varnothing.$$

Definition 6 (Hierarchical Child Graph and **Simple Child Graph).** A *hierarchical child graph* of a node v_r is a subgraph of $G_o = (V, E, F, r, P, vp)$ where the descendants of v_r in the original inclusion tree $T_o = (V, F, r)$ form a hierarchical graph with v_r as the virtual root node, i. e., using $De_o(v)$ as the descendants function on T_o:

$$G' = (V', E', F', r', P', vp') \text{ is hierarchical child graph of } v_r \in V \Leftrightarrow r' = v_r, V' = De_o(v_r).$$

A *simple child graph* of a node v_r is the simple graph containing only the children of v_r and edges between these children. We denote the simple child graph of a node $v \in V$ as $simple(v)$.

Definition 7 (Layer). In layer-based graph layouts, there are $n \in \mathbb{N}_0$ layers $L_i | i < n$. These partition the set of nodes per simple graph: Each node is contained in exactly one layer. The nodes in a layer form a tuple $L_i = (v_{(i,0)}, \ldots, v_{(i,k_i-1)}), k_i \in \mathbb{N}_0$, where k_i is the number of nodes in L_i. The layered graph shall be denoted by the tuple $\mathbb{L} = (L_0, \ldots, L_{n-1})$ Layers L_i, L_j are *neighbors* or *neighboring* if $|i - j| = 1$.

In contrast to most publications on layered graphs, we assume a left-to-right layout of the layers. This is because ELK Layered is mostly used to lay out data flow diagrams, where the left-right direction is more common.

In most algorithms the problem of minimizing edge crossings is reduced to the subproblem of minimizing crossings between two neighboring layers. The order of nodes in one of these two layers is fixed, while the order of nodes in the second layer is changed. This second layer shall be called the *free layer*, or L_{free}, as opposed to the *fixed layer* or L_{fix}.

Definition 8 (Port Position Function and **Node Position Function).** Each node and also each port has a *position*, i. e., within each layer, ports and nodes are numbered from top to bottom when on the left or right side of a node and left-to-right when on the top or bottom side. The functions $pos_p : P \to \mathbb{N}_0$ and $pos_v : V \to \mathbb{N}_0$ map ports and nodes to their indices.

Definition 9 (Hierarchical Port and **Hierarchical Port Dummy).** To simplify the implementation of crossing minimization algorithms, edges connecting ports on nodes with different parent nodes in the inclusion tree are changed internally in the following manner: For each node v with edges ending in a descendant of v an extra layer at the leftmost (for ports on the west of v) or rightmost (for ports on the east of v) position of the layers is added. Here, for each port on a hierarchical node we add a hierarchical port dummy node in the new layer. This node is treated as a normal node of the child graph with the exception that in a final ordering of the nodes and ports, hierarchical port dummies and hierarchical ports must be beside each other, i. e.,

$$v \in V \text{ is hierarchical port dummy of } p \in P \Rightarrow pos_v(v) = pos_p(p).$$

See Figure 2.2 for an overview over the different types of dummy nodes.

Definition 10 (Simple Port). A *simple port* is a port on a simple node, or a port on a hierarchical node whose corresponding hierarchical port dummy has no outgoing edges.

Definition 11 (Hierarchical Edge). A *hierarchical edge* is an edge connected to a hierarchical port dummy.

Definition 12 (Port Constraints). There are two *port constraints* relevant for our use case (for other constraints used in ELK Layered see Schulze et al. [SFvHM10]): Port ordering is *fixed* when the input graph defines the order of ports for each node. Port ordering is *free* if the layout algorithm can switch the order of ports on the node. In our case a free ordering of ports means that while the order can be switched, the side of the node on which the ports are situated must remain the same.

Definition 13 (Long Edge Dummy, North/South-Port Dummy, In-layer Edge Dummy). To simplify the problem of crossing minimization to two layers, edges traversing more than one layer are replaced by a chain of long-edge dummies, one for each layer.

When an edge is incident to a port on the top or bottom side of a node, a north/south port dummy is added to show where the bend in the edge will be drawn.

Let v and u be nodes with u positioned in the same or a layer to the right of v. Then the nodes are connected by a *feedback edge* $e = \{p_1, p_2\}$, with $vp(p_1) = (v, d)$ and $vp(p_2) = (u, d')$, when $d, d' \in \{w, e\}$ and $d \neq d'$. Feedback edges pass through the same layer as the nodes they connect. When they do so, they are split by *in-layer edge dummies*

See Figure 2.2 for examples of in-layer edge dummies, long edge dummies and north/-south port dummies.

Definition 14 (In-layer and Between-layer Edge). In the context of crossing minimization, edges can be *between-layer* or *in-layer*. An edge $e = \{p_1, p_2\}, p_1, p_2 \in P$ is in-layer when it connects nodes in the same layer $L \in \mathbb{L}$ on the same side $d \in D$:

$$e \text{ is in-layer } \Leftrightarrow vp(p_1) = (v, d), vp(p_2) = (v', d) \wedge v, v' \in L$$

Note that an edge whose ports are on different sides of nodes in the same layer in the input graph is a feedback edge, so an in-layer edge dummy is added. For an example, see Figure 2.2

Between-layer edges $e = \{p_1, p_2\}, p_1, p_2 \in P$ pass between neighboring layers $L_i, L_j \in \mathbb{L}$:

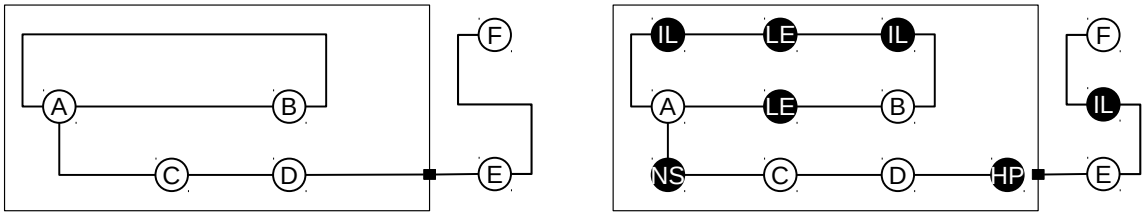
$$e \text{ is between-layer} \Leftrightarrow vp(p_1) = (v_1, d_1), v_1 \in L_i \wedge vp(p_2) = (v_2, d_2), v_2 \in L_j \wedge |i - j| = 1$$

With these definitions all set, we will have a look at the layout method our algorithm will be a part of.

2.2 Layered Graph Layout

The following section will give a short introduction to some basic background information concerning the type of graph layout in question and the basic algorithm framework used for automatic layout, called ELK Layered. This description is based on the one in my bachelor's thesis [Sch15].

The goal of ELK Layered is to construct a graph layout which is suited for graphs which have an inherent edge direction, such as data flow diagrams. Data flow diagrams are used to



(a) Input Graph **(b)** Internal Representation

Figure 2.2. (a) shows the input graph with the nodes A,B,C,D,E. This is transformed to the internal interpretation in (b). The feedback edge from B to A creates two in-layer edge dummies (IL). Another in-layer edge dummy is created when an edge connects ports on different sides of nodes in the same layer, such as the edge between E and F. When traversing more than one layer an edge is split up into portions separated by long-edge dummies (LE). When a node has an edge connected to its north or south side, a north/south edge dummy is created (NS). When an edge connects nodes with different parents in the inclusion tree, a hierarchical port dummy is created (HP).

demonstrate or model the flow of data through components of a given system. As an example consider the one in Figure 1.1.

As the name suggests, ELK Layered uses a *layered* approach to automatic graph drawing. This means that nodes are distributed among a set of layers. To make the graph as readable as possible, the algorithm tries to optimize different criteria, the most important being:

- ▻ Maximize the number of edges directed from left-to-right, following the reading direction.
- ▻ Minimize crossings between edges.
- ▻ Minimize edge bends.

ELK Layered is based on an algorithm developed by Sugiyama, Tagawa and Toda [STT81]. The following section only gives a short and general overview of the Sugiyama framework. For a more detailed description, see the original paper [STT81].

The algorithm is divided into three steps:

1. Distribute nodes to layers and replace long edges with node dummies.
2. Permute order of nodes.
3. Position nodes on the layer.

After the first step, a layering of a graph is then considered to be a *proper* layout when all edges go in a left-right direction and connect nodes from one layer to nodes in a neighboring layer. This why any edges going further than to the neighboring layer are broken by dummy nodes. Node dummies are nodes which at the end of the algorithm are removed.

The second step reduces the number of edges crossings. This is the main concern of this thesis and is more carefully discussed in the following section and the special problems of hierarchical layout in Chapter 4.

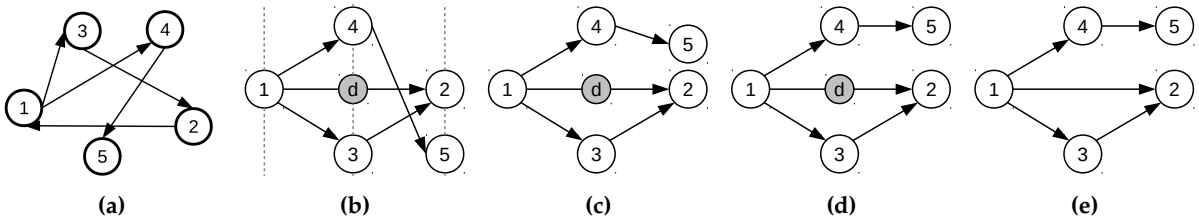


Figure 2.3. Steps in Sugiyama algorithm: (a) Input graph. (b) Adds dummy edges and assigns layers. (c) Reduces edge crossings. (d) Position nodes. (e) Remove dummy nodes.

The third step positions the nodes horizontally in each layer in order to reduce the number of edge bends. Figure 2.3 shows the traversal of all the steps in the algorithm.

The following sections will consider step two more closely: The layer sweep crossing minimization and two heuristics which ELK Layered in conjunction with this method, the barycenter heuristic suggested by Sugiyama et al. [STT81] and the greedy switch heuristic by Eades et al. [EW86].

2.2.1 Layer Sweep Crossing Minimization

As shown by Eades and Wormwald, minimizing the number of edge crossings by permuting nodes in a layered graph is NP-complete [EW94].

The common approach is to reduce the difficulty of the problem by only considering a pair of layers at a time. As determined in Definition 7, one of these is the free layer L_{free}, whose node order is permuted, and the other is the fixed layer L_{fix}, whose node order is fixed. This is a problem which should be easier but unfortunately is also NP-complete [EW94].

To reduce edge crossings in the whole graph, many heuristic algorithms sweep from left-to-right and backward across the layers until no further improvement of the number of crossings is achieved, as shown in the algorithm *layerSweep* (Algorithm 1).

Depending on the heuristic, at least the permutation of the first fixed layer needs to be set before one can sweep forward and backward across the layers. The simplest way to do this is to choose a random order. Obviously, some choices for the order of the first layer may be better than others. To take this into account, one can simply run the algorithm with several randomized values and take the best result. The number of times this is done in ELK Layered is set by the *thoroughness* parameter.

To choose a good permutation with feasible computation time, many different heuristics have been suggested. Sugiyama et al. developed an algorithm which they called the *barycenter heuristic* [STT81]. The *barycenter* (= center of mass) of a node is simply calculated as the average of the positions of its adjacent nodes. The nodes are then sorted by their barycenters. This heuristic is fast in theory and in practice and gives good results, which is why ELK Layered uses this algorithm (for experimental evaluations see the research discussed in Section 3.2).

```
   Input: Layers 𝕃
   Output: Reordered layers 𝕃_bestOrder
 1 currentCrossings = countCrossings(𝕃);
 2 forward = true;
 3 do
 4 |   𝕃_bestOrder = copy of (L_0, ..., L_n), storing the current order of each layer
 5 |   if forward then
 6 |   |   for i = 0 to n − 2 do
 7 |   |   |   L_fix = L_i
 8 |   |   |   L_free = L_{i+1}
 9 |   |   |   L_i = permute(L_fix, L_free)
10 |   |   end
11 |   end
12 |   else
13 |   |   for i = n − 1 to 1 do
14 |   |   |   L_fix = L_i
15 |   |   |   L_free = L_{i−1}
16 |   |   |   L_i = permute(L_fix, L_free)
17 |   |   end
18 |   end
19 |   forward != forward;
20 |   lastCrossings = currentCrossings;
21 |   currentCrossings = countCrossings(𝕃)
22 while lastCrossings > currentCrossings;
```

Algorithm 1: layerSweep

As shown in my bachelor's thesis [Sch15], the barycenter heuristic results in specific types of errors, which can sometimes be an Obviously Non-Optimal Graph (O-NO-graph). For this reason, the *greedy switch* heuristic was implemented as a post-processing step. Here, neighboring nodes are switched, if this reduces the number of crossings.

2.3 Bottom-Up Hierarchical Layout

The current implementation in ELK Layered lays out hierarchical graphs using a simple method, which we shall call BU: Each simple child graph is laid out separately in an order such that each child graph is laid out before the graph containing its parent node is. After finishing the complete layout of the child graph, i. e., including all steps of the layout algorithm, the position and order of the parent node's ports is fixed. Then, from the viewpoint of the layout algorithm processing the parent node's graph, the node is now a simple node. In this way, the layout algorithm itself never processes a hierarchical graph but only simple graphs.

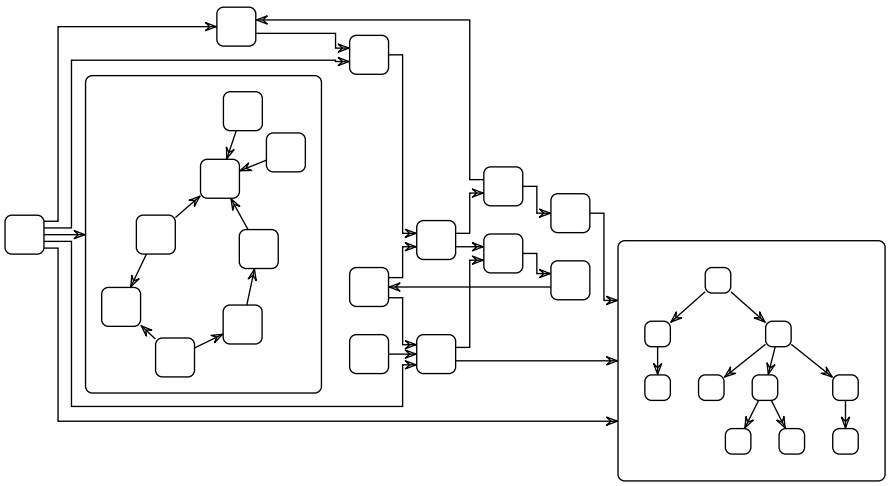


Figure 2.4. Example where three different kinds of layout algorithms are used for different simple graphs of the hierarchical graph: The uppermost graph in the inclusion tree is laid out using ELK Layered, while the simple child graph on the left uses force directed layout and the simple child graph on the right uses tree layout.

This method has a number of advantages: First of all, it simplifies the implementation, since even though the graph in the final layout is hierarchical, the different phases must only deal with simple graphs. Secondly, it enables choosing between different layout algorithms, e. g., non-layered based algorithms such as force-directed methods or tree layout algorithms for each simple graph contained in the hierarchical graph. For an example of such a layout, see Figure 2.4.

The disadvantages of BU are a direct consequence of the necessity to fix the order of the ports after finishing a layout of a child graph. Graphs with fixed port order can lead to many unavoidable crossings. The main contribution of this thesis is a method which can help remove these kinds of edge crossings by taking the complete hierarchical graph into account (see Chapter 4).

Before taking a detailed look at the contributions, we will first examine related work dealing with the automatic layout of hierarchical data flow graphs, crossing minimization and crossing counting as well as port sorting.

Chapter 3

Related Work

3.1 Global and Local Layering

The automatic layout of graphs with hierarchical nodes using the Sugiyama framework is not a very common area of research.

In the context of the ELK Layered algorithm Fuhrmann [Fuh12] implemented a global layering approach. As part of this exploratory thesis, an extensive literature overview is given, not only taking into account publications directly related to the problem, but also examining related areas such as clustered graphs or hierarchical graphs using the force-directed approach. Since a global layering scheme can often lead to reduced graph size, a global layering algorithm was chosen and implemented in ELK Layered. The evaluation shows an increase of computation time, which for large graph instances can be problematic. According to the author, the crossing minimization did not lead to satisfactory results.

As Fuhrmann points out, basically only two publications exist which provide algorithms for dealing with the hierarchical graph problem in the context of layered graph layouts. These are a paper by Sugiyama and Misue [SM91] based on the local layering approach and a paper by Sander [San96] based on the global layering approach.

As described above, the local layering approach mainly differs from the Sugiyama framework for flat graphs with regard to the crossing minimization phase. Sugiyama and Misue's algorithm tries to achieve three goals: Firstly, maximize the closeness of the nodes, meaning the distance between two hierarchical nodes, where one node has children with edges connected to children of the other node. Secondly, they aim to minimize edge crossings and thirdly to minimize the crossings between edges and node borders. In general, for crossing minimization they use a top-down approach. This means that they first do a layer sweep across the simple graph of the virtual root node and then minimize crossings in each child node recursively. The problems arising from this approach would be the same as with BU, however, in contrast to ELK Layered, they do not use ports. In this case, the edges can be routed directly to nodes in other hierarchical simple graphs. Because of this the nodes are placed as close as possible to the side of its parent node where most of the hierarchy traversing edges go to, which the authors call the *splitting* method. To minimize edge crossings, the barycenter algorithm is used and for minimizing crossings between edges and node borders, a barycenter insert method is applied. The splitting method is applied as a preprocessing algorithm to find a starting point for the two barycenter algorithms which are then applied in alternating fashion. Note that in ELK Layered, there can be no unnecessary crossings of edges and node borders. These edges will always be routed around nodes they would otherwise

cross. Furthermore, while the goal of minimizing closeness is an interesting one, it is currently not one of the aims of ELK Layered.

Raitner published a paper expanding on the suggestions proposed by Sugiyama and Misue [Rai05]. The use case driving this publication was the possibility for users to expand and contract hierarchical nodes. Raitner observed that redrawing the complete layout when executing such a command is slow and more importantly, changes the layout of the complete graph. This can cause a conflict with the mental map which users have formed of the graph they are working on. To reduce this problem Raitner suggests a local update scheme. To reach this goal, all old nodes stay in the same layers and only the nodes of the expanded graph are put into new layers. In the case of the crossing minimization phase, the order of all nodes outside of the expanded hierarchical node remain the same. To achieve this, the order of the hierarchical dummy nodes is fixed. Note that in ELK Layered, users can also expand and contract hierarchical nodes. Using a fixed port order on the expanded node, the layout of the rest of the graph could also be kept the same. However, in this case the complete graph layout would still be recalculated.

Sander suggested a global layering method [San96] which aims to be an alternative to the local scheme described earlier, although in its general form it remains similar to the algorithm by Sugiyama and Misue. The main difference is in the use of global layers, meaning that there is only one set of layers for all nodes of the graph across the complete inclusion tree. A hierarchical node can then span several global layers. The algorithm follows the basic pattern of the Sugiyama framework, but with significant differences in many of the phases. Fuhrmann adapted Sanders method to KLayLayered, predecessor of ELK [Fuh12] which works on a flat representation of the hierarchical graph.

In general, the global layering approach often creates more compact drawings at the cost of computation time. Since the existing implementation in ELK Layered also requires an increased amount of maintenance, this thesis explores the possibilities of a simplified local layering scheme.

Currently, the implementation in Eclipse Layout Kernel Layered (ELK Layered) uses Bottom-Up (BU): Here, each child graph is laid out completely before the graph containing it's parent node. ELK Layered shares this approach with two other implementations of the layered layout algorithm based on Sugiyamas work, one from Paulisch and Tichy [PT90] and one from Henry [Hen92].

Henry notes that the user can put the most important modules in the lowest level of the hierarchy. Using BU then lets this module determine the layout of the graph on the outer levels. However, this idea is not suited for the use case of data flow graphs. Furthermore, he states that the most important feature of BU is its simplicity, and also points out the edge crossings which can occur on the boundary of the hierarchical node. In his system, these crossings can be manually removed. This is currently not possible in ELK Layered. He also shortly discusses a top-down approach, and without giving concrete examples touches on the subject of the algorithms communicating in both directions along the hierarchy. For the crossing minimization algorithm, this is what HALS does.

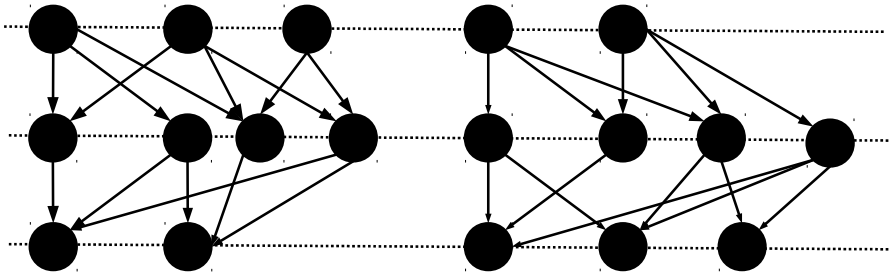


Figure 3.1. Contrived example taken from Bachmeier et al. [BBBH10] a two-layer sweep algorithm cannot lead to a global optimum. Each of the two unconnected components of the graph oscillates between five and six crossings on every sweep. Since one component is just a flipped version of the other, the total number of crossings remains at eleven, although ten would be optimal.

Paulisch and Tichy also point out the unnecessary edge crossings which occur using BU, but note that the layout of the child graph is unaffected by changes to the graph containing the parent node is a benefit of BU.

3.2 Crossing Minimization

The most relevant problem of BU is the high number of unnecessary edge crossings. Minimizing the number of edge crossings is generally seen as an important aspect to increase readability and aesthetic quality of a graph layout. The following will give a short overview of different crossing minimization algorithms in flat layered graphs.

This topic has been researched extensively, with a large number of publications suggesting heuristics and optimal methods for two-layer crossing minimization. These can be used for the general k-level problem by sweeping through the graph and comparing neighboring layers. Even when reduced to this task of minimizing crossings by ordering only one part of a bipartite graph, the problem still remains NP-complete [EW94]. Some of the algorithms suggest to optimize the number of crossings by changing both sides of a two-layer graph. These cannot be used for a layer by layer sweep without some kind of adaptation. The general k-level crossing minimization problem has not been researched as extensively as the two-layer problem. However, as Bachmeier et al. point out [BBBH10], there are cases when even an algorithm which leads to an optimal (i. e., minimal) number of crossings but considers only two layers at a time, cannot lead to an optimal global layout, as can be seen in Figure 3.1.

No current and extensive overview or comparison of the effectiveness of the many approaches for the bipartite graph crossing minimization problem exists. Jünger et al. published a comparative study for heuristics and exact algorithms of two-layer problem in 1997 [JM97] and Martì et al. published a comparison of heuristics for the same problem in 2003 [ML03], however new solutions have been suggested since then.

To the best of my knowledge the only overview of k-level crossing minimization heuristics is by Bachmeier et al. [BBBH10] Here the authors compare the layer-sweep barycenter

approach with a global barycenter heuristic, a layer-sweep sifting solution and two k-level sifting approaches, including one of their own and one by Matuszewski et al. [MSM99]. Sifting is a very simple process: Starting from a predefined order, for every node in a layer each possible position for that node is considered while keeping the relative order of the other nodes fixed. Then the best position is chosen. The global barycenter does not fare well and the two-layer sifting approach results in similar solution quality as the barycenter algorithm, while being significantly slower. The global sifting methods return better results than the common barycenter approach at the cost of increased computation time.

The barycenter heuristic was first suggested in the original paper by Sugiyama et al. [STT81]. Together with its variants described below, it is currently the fastest known algorithm and returns good results. The barycenter of a node in the free layer is calculated as the average of the indices of its respective neighbors in the fixed layer. The vertices in the free layer are then sorted by their barycenters.

Similar algorithms to the barycenter heuristic have been suggested, including the median heuristic which uses the median instead of the average, suggested by Eades and Wormwald [EW94] and the semi-median suggested by Mäkinen et al [Mä90] which includes a greedy tie-breaking strategy.

Eades and Kelly [EW86] introduced the *greedy switch* heuristic, which switches neighboring nodes when this causes a reduction in the crossing number, together with the *greedy insert* heuristic which minimizes the number of crossings to nodes already set until no nodes are left and the *split* heuristic, which imitates a quicksort approach to order nodes. None of these have proven to be successful in practice. However, greedy switch and the split heuristic can be used as post-processing steps, e. g., after applying the barycenter method. ELK Layered currently uses greedy switch as a post-processing step to the barycenter method, an approach which also has been used by Gansner et al. [GNV88] and Nachmanson [NRL08]. In its default setting, this implementation uses the *two-sided greedy switch*, where the crossings to both sides of the free layer are considered. This leads to better performance at the cost of more crossings (see Schelten [Sch15]).

Further algorithms which have proven to be unsuccessful in practical applications are the assignment heuristic suggested by Catarci [Cat95] and the stochastic heuristic suggested by Dresbach [Dre95].

A research group at the University of Valencia including Valls, Martì, Laguna and Lino have published several approaches for minimizing crossings in two-layer graphs using meta-heuristics, such as taboo thresholding [VML96], taboo search [Mar98], GRASP [LM99] for the two-layer case, as well as a taboo search [VML96] algorithm for the k-level problem. These meta-heuristics use combinations of the other known heuristics, including greedy switch, sifting and barycenter.

Another meta-heuristic applied to solve the global crossing minimization problem is presented by Kuntz et al. They use a hybridized genetic algorithm which results in better solution quality than the taboo search method suggested earlier, however once again at the

cost of slow computation time. Unfortunately, they do not compare their heuristic to any of the two layer sweep heuristics.

It is often difficult to judge the practical success of these algorithms. It seems that using barycenter together with the layer sweep approach combined with the greedy switch heuristic remains a robust and fast method. However, it might be interesting to choose and implement one of the several other approaches into ELK Layered to compare them.

3.3 Cross Counting

Since the two-layer sweep only searches for local optima, all two-layer crossing minimization algorithms, including an optimal one, may increase the number of crossings. To check the success of a sweep, we must therefore count the number of crossings each time. This is therefore an operation which must be efficient for any crossing minimization algorithm.

There have been a number of publications dealing with the problem of counting crossings. Due to the complexity of the graphs in use in ELK Layered, there are a number of cases which have to be dealt with. This includes in-layer edges, i. e., edges between nodes in the same layer and edges connected to ports on the north or south port of a node.

Barth et al. [BJM02] suggested an efficient algorithm for counting between-layer crossings. This is the algorithm currently used in ELK Layered. The algorithm by Barth et al. is described in more detail in Section 3.3.

For counting in-layer crossings, an algorithm which counts crossings of in-layer edges was suggested in the context of my bachelor's thesis [Sch15].

The algorithm iterates over all ports and their edges in the graph, saving the start and end position of each in-layer edge and the position of the current port for each between-layer edge in a sorted list. Each time we meet an edge which has been visited, the port positions in between the end and start position of this edge are counted and deleted it from the list. The algorithm is run twice, once for the edges on the eastern side of the layer in question and once for those on the western side.

The implementation uses a binary tree datastructure with the ability to specify a cardinality for its leaves. It is used to store the port positions of each edge. The algorithm runs in $O(|E|(|E_{IL}| + |E_{BL}|)log(|E_{IL}| + |E_{BL}|))$, where $|E_{IL}|$ is the number of in-layer edges and $|E_{BL}|$ the number of between-layer edges [Sch15].

This is obviously a poor running time. To improve this and to reduce the amount of maintenance needed for two different counting algorithms for in-layer and between-layer edges, Section 3.3 presents a simple algorithm for counting both types of crossings at the same time.

For the case of crossings between north and south ports, an algorithm for counting these crossings in linear time can be found in my bachelor's thesis [Sch15], which will shortly explain here.

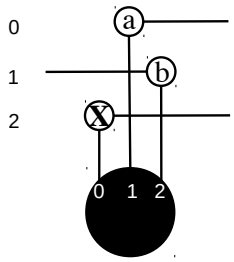


Figure 3.2. The row of numbers in the node show the position of the ports. The numbers on the side show the *nearness* of the north/south dummies to their origin port, i. e., the closer the node, the higher the nearness.

For this, consider Figure 3.2. To simplify, we can draw the situation as a matrix:

$$\begin{bmatrix} - & \boxed{a} & \boxed{-} \\ - & \boxed{-} & \boxed{b} \\ \mathbf{X} & - & - \end{bmatrix}$$

The box shows the nodes on the top right of the node in the bottom left position (written in bold font), which corresponds to the node at position $0, 2$. Comparing Figure 3.2 and the matrix, we can easily see that the number of crossings of a north/south dummy with an eastern edge is equal to the number of nodes in the marked area of the matrix. We use this to formulate the following method of calculating the crossings.

We use the *nearness* of a north/south dummy v_{NS} to the node with the north/south ports v: This measures how close a north/south dummy is in comparison to the number of north/south dummies on the side of v_{NS}.

$$nearness_{v_{NS}} := |conn_d(originNode(v_{NS}))| - |pos_v(v_{NS}) - pos_v(originNode(v_{NS}))|,$$

where $|conn_d(v)|$ is the number of edges incident to node v on side $d \in D$, and $originNode$: $V_{NS} \to V$ maps north/south port dummy to its connected normal node in the same layer.

Then, the number of crossings for a north dummy v_N with a western edge can be calculated as:

$$min(pos_p(originPort(v_N)), nearness(v_N)),$$

where $originPort : V_{NS} \to V$ is the port the dummy node is connected to and pos_p numbers the ports from east to west in ascending order. The number of crossings with an eastern edge can be calculated as:

$$min(card_s(v_N) - pos_p(originPort(v_N)) - 1, nearness(v_N))$$

For north/south dummies on the southern side of a node, the same algorithm is used.

3.4 Port Sorting

A special problem in ELK Layered is the support of *ports*. Sander proposes to handle ports by adding dummy nodes for each port [San94]. Crossing minimization with the use of ports was first suggested by Waddle et al. [Wad01], who uses an adaption of the barycenter heuristic. The implementation and algorithms necessary in ELK Layered are explained in detail in Schulze et al. [Sch11]. The following will give a short overview.

As described in the preliminaries in Chapter 2, each node has ports. The position of each port can be constrained by different port constraints. These can be specified before the algorithm is applied and are also used to separate concerns during execution of the algorithm.

Especially important for the crossing minimization phase are the two port sorting algorithms, presented by Schulze et al. [Sch11]. Both are based on the barycenter heuristic. However it should be easily possible to transfer most of the two-sided crossing minimization heuristics to the port sorting problem.

To calculate the barycenter value of a node with ports, the concept of *port ranks* $r(p)$ for a port p is introduced. For a node v, the barycenter is then redefined using the port ranks to be $\frac{1}{|E(v)|}\sum_{(p_v,p_u)\in E(v)} r(p_u)$, where $p_v, vp(p_v) = v$ is a port on v and $E(v)$ are all edges connected to ports on v. This value is used to sort the nodes just as in the usual barycenter heuristic.

For simplicity we will only describe the case of the forward sweep. To calculate the port ranks, Schulze et al. suggest two similar but slightly different approaches: The first version is called *layer total rank* r_{LT} of v_j, which gives a unique integer value to each port, thereby giving nodes with many ports a greater range of rank. It is defined as $r_{LT}(p) = (\sum_{k<i} range(v_k)) + pos_p(p)$, where $range(v_k) = |P'(v_k)|$ is the range of port ranks occupied by a node v_k in L_a. The other they call *node-relative* and define it as $r_{NR}(p) = pos_v(vp(p)) + \frac{pos_p(p)}{range(v)+1}$. Here, each node is assigned an equal range of one.

When comparing the two different approaches Schulze et al. come to the conclusion that both methods yield different results for different graphs, however on average they are equally effective. For this reason ELK Layered chooses randomly between the two implementations.

Since the port sorting phase will be integrated in the crossing minimization phase as described in Section 4.1.1, sorting north/south ports will also be executed during crossing minimization. Schulze et al. describe the implementation chosen in ELK Layered: hi The north/south port dummies are sorted by the crossing minimization algorithm, permitting long edge dummies to be placed in between the north/south port dummies of a node. To sort north/south ports, each dummy node is assigned a barycenter value depending on its position and whether it has incoming, outgoing or both kinds of edges. These are used to sort the ports on their respective sides.

Chapter 4

Hierarchy-Aware Crossing Minimization

This chapter describes the main algorithm proposed for crossing minimization in hierarchical graphs: The Hierarchy-Aware Layer Sweep (HALS). The first section describes the basic principle. Following that, different issues which can arise from this original idea are shown. First, we examine a problem coming from non-deterministic two-layer sweep algorithms which can cause an increase in the crossing number. The next section then briefly shows how HALS can in some cases increase computation time. Then we examine different solutions for these caveats, using a heuristic which flexibly decides for each hierarchical node whether to process the child graph separately or as part of its parent. Finally, two issues which influence the success of the crossing minimization, which are only indirectly related to the hierarchical layer sweep algorithm are shown: The first is a problem which can arise when having both hierarchical and simple ports on a node laid out using BU. The second is an efficient algorithm for counting both in- and between-layer crossings at the same time.

4.1 Layer Sweep

The following section discusses an adaption of the layer sweep crossing minimization algorithm as described in Chapter 2. To understand the idea of the hierarchy-aware sweep, we will first look at a short intuitive explanation, as well as a pseudo-code representation of the algorithm and a concrete example showing all necessary steps.

The general principle of the normal two-layer sweep will be kept: The order of the nodes in one layer (the fixed layer) is preserved while the order of the nodes in the other (the free layer) is changed. Now, on each hierarchical node we sort the ports after each re-sorting of the free layer. Then, the port dummy nodes on the sweep side of the hierarchical node are sorted according to the order of their respective hierarchical ports. Finally, the algorithm proceeds to sweep across the child graph. If the child graph contains any more hierarchical nodes, we proceed in the same manner. In the case where hierarchical edges exit the other side of the hierarchical node currently being processed, the last layer in the child graph will be the layer containing the hierarchical port dummies. Now, the ports on the this side of the hierarchical node are sorted according to the order of the port dummies. Once all hierarchical nodes in the layer have been processed, the sweep continues. In this way the entire hierarchy is visited in one sweep.

The pseudo-code for this procedure can be found in Algorithm 2 (*layerSweep*), which uses Algorithm 3 (*sweepForward*) and an equivalently defined variant of the latter: *sweepBackward*.

```
   Input: Layers 𝕃
   Output: Reordered layers 𝕃_bestOrder
 1 lastCrossings = ∞
 2 currentCrossings = countCrossings(𝕃)
 3 forward = true
 4 while lastCrossings > currentCrossings do
 5 |   𝕃_bestOrder = copy of (L_0, ..., L_n), storing the current order of each layer
 6 |   if forward then
 7 |   |   sweepForward(𝕃)
 8 |   end
 9 |   else
10 |   |   sweepBackward(𝕃)
11 |   end
12 |   forward = !forward
13 |   lastCrossings = currentCrossings
14 |   currentCrossings = countCrossings(𝕃)
15 end
```

Algorithm 2: layerSweep

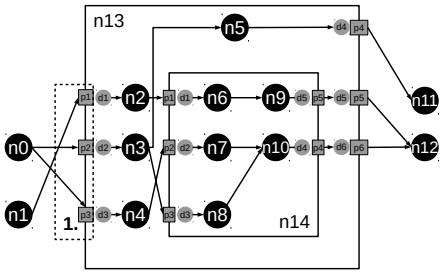
Before starting to sweep across the graph, Algorithm 2 counts the number of crossings using the order given in the input. The sweeping stops as soon as the number of crossings does not shrink anymore. We are able to use any two-layer *permute(fixed layer L_{fix}, free layer L_{free})* algorithm for the one-sided crossing minimization problem, and any *sortPorts(node n, sweep side s)* algorithm to sort ports on a node in such a way as to minimize crossings. Note the condition in *sweepForward*, where a compound node can be marked so as to allow traversing downward in the inclusion tree or not. The reason for this option and its use is explained in Section 4.2.

Finally, Figure 4.1 gives an example with a step by step visualization of the approach.

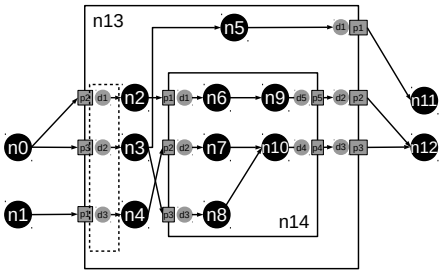
4.1.1 Sorting Ports During Sweep

In ELK Layered, ports are currently sorted in a separate step after the crossing minimization phase. For this, the crossing counting algorithm does not count all crossings of edges to ports on nodes with free port order. However, this separation can lead to crossings which could easily have been avoided in the crossing minimization phase, as can be seen in Figure 4.2.

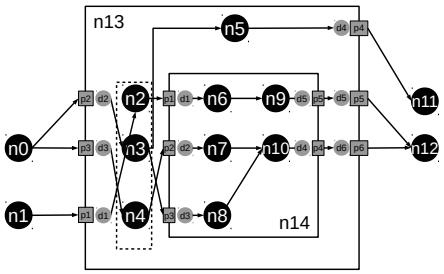
As described above, the hierarchy-aware sweep must sort ports during crossing minimization at least for every hierarchical node. Since cases such as in Figure 4.2 can happen, this is changed so that sorting ports is done during sweep for all nodes: All port orders are assumed to be fixed in the crossing minimization phase and ports are sorted after each reordering of the free layer. The port sorting algorithms already implemented in ELK Layered as described in Schulze et al. [Sch11] can be used for this. This leads to fewer crossings by removing such



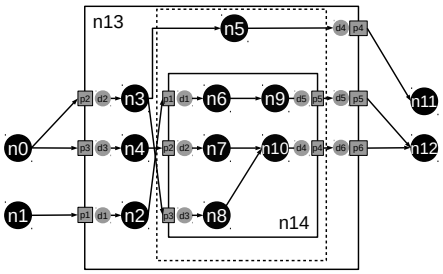
(1.) The ports of `n13` are sorted.



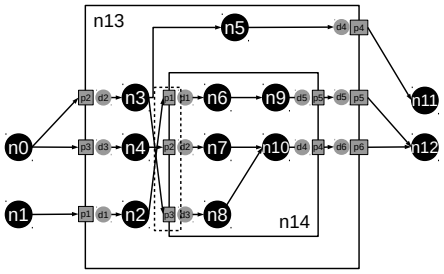
(2.) The hierarchical dummy nodes of `n13` are sorted by their hierarchical ports, according to the order determined in (1.).



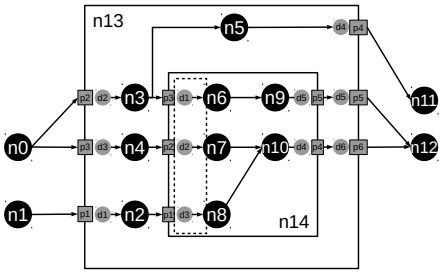
(3.) Now the layer with the dummy nodes is the fixed layer, and the layer marked with the dotted box is the free layer whose nodes are are sorted with respect to the order of the fixed layer.



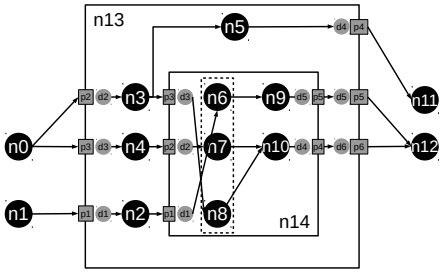
(4.) The nodes in the next layer (`(n5,n14)`) are sorted.



(5.) The ports on `n14` are sorted.



(6.) The hierarchical dummy nodes of `n14` are sorted by their hierarchical ports.



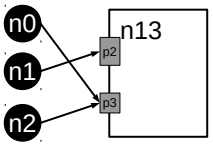
(7.) The layer sweep is continued within the child graph of `n14`.

Figure 4.1. Example for the steps of HALS.

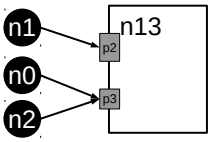
Input: Layers $\mathbb{L} = (L_0, L_1, \ldots, L_{n-1})$
Output: Reordered layers $\mathbb{L}$

```
1  for i = 0 to n − 2 do
2      L_fix = L_i
3      L_free = L_{i+1}
4      L_i = permute(L_fix, L_free)
5      for v_i ∈ L_i | v_i is compound do
6          if v_i marked for hierarchical sweep then
7              sort eastern ports of v_i and dummy nodes
8              sweepForward(childGraph(v_i))
9              sort western ports of v_i by their corresponding dummies
10         end
11     end
12 end
```

Algorithm 3: sweepForward. sweepBackward is defined equivalently



(a) If the ports of `n13` have free port order, the current policy considered all ports of `n13` to be at the same position, assuming that all remaining crossings can be resolved in a separate port sorting phase. However, this is impossible in this case.



(b) The crossings could easily be removed by the crossing minimization algorithm.

Figure 4.2. Reason for sorting ports during the layer sweep.

crossings as in Figure 4.2, but might also lead to a slower run-time, because the ports are re-sorted in each sweep.

Using the barycenter port sorting algorithms excludes the use of the two-sided greedy switch algorithm as previously described in my bachelor's thesis [Sch15]. The two-sided greedy switch exchanges the position of neighboring nodes if the number of crossings on both sides of the node would be reduced. Because then the number of crossings can never increase, this enables the algorithm to skip counting crossings, which improves the running time. However, the port sorting algorithm in ELK Layered uses barycenter values and therefore does not offer this guarantee: Using barycenter can in some cases increase crossings. Furthermore, the barycenters are calculated with respect to the edges going to the fixed layer only. To continue using the two-sided greedy switch algorithm, the two-sided greedy switch approach must be applied to port sorting. This includes counting crossings caused on the inside of a hierarchical node.

While the basic idea of this algorithm is simple, there are a few limitations which become apparent when taking a closer look. The following sections do exactly that.

4.2 Limitations

4.2.1 Solution Quality

When using a non-deterministic two-layer sweep algorithm, the hierarchical layer sweep can in some cases cause significantly worse layouts then BU. As an abstract explanation, this is because BU is a divide and conquer approach. Therefore, it must make fewer random decisions per sweep, because it sweeps on smaller graphs. This way, it has a higher probability of making a combination of random decisions which leads to a better layout. To understand this in more detail, let us first take another look at the main crossing minimization algorithm: the barycenter heuristic.

The barycenter algorithm (see Section 2.2.1) depends on a number of random decisions. Barycenter values for two nodes are varied by small random values to decide on a specific order when the barycenter calculation resulted in equal values. Furthermore, each node with no connection to the fixed layer is given a random barycenter value which then defines its position in the layer. In ELK Layered, this is only the case in the first sweep of the algorithm. The subsequent sweeps are implemented as follows: For a node n_u with unknown barycenter value, this value is set by taking the middle value between the known barycenters of the two nodes neighboring n_u in the order calculated in the previous sweep. However, since the direction of the first sweep is set randomly in ELK Layered, both nodes with no eastern and no western connections can have completely random barycenter values.

This randomization process is de-facto part of all two-layer crossing minimization algorithms. If there are no connections to the fixed layer, there is no way to determine the correct position of a node. In the case of the greedy switch algorithms implemented in ELK Layered, the one-sided greedy switch does not randomly re-sort such nodes. This is because it is applied as a post-processing step after the barycenter algorithm, the assumption being that the order found already contains only few crossings. If greedy switch were the only crossing minimization algorithm, one would have to consider comparing different random initial layouts.

For the barycenter algorithm, multiple layouts with different random decisions are compared and the one with the fewest crossings is chosen. The number of performed layout runs is defined by a user parameter which we call *thoroughness* or t.

As an example, consider a hierarchical graph G. To prove our point and for the sake of simplicity, the graph shall contain no hierarchical edges. Furthermore, let its inclusion tree contain two levels and $\mathbb{G}_2$ be the set of simple graphs in the second level of the inclusion tree. When layouting G using BU, each graph in $\mathbb{G}_2$ is considered separately. Let the probability of an optimal layout for a graph $g_{2_i} \in \mathbb{G}_2$ be p_i. To simplify the following calculation, we choose the graph in such a way so that the crossing number of the parent graph is always optimal and can therefore be ignored. See Figure 4.3 for an example of such a graph.

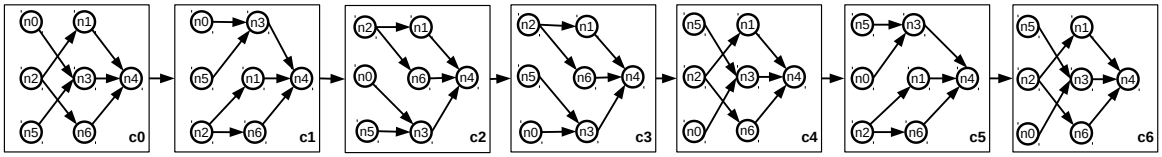


Figure 4.3. Example hierarchical graph with five hierarchical nodes containing equal graphs (on level 2 of the inclusion tree). When sweeping backward, the barycenters for `n1`, `n3` and `n6` are equal. Therefore the order of the second layer is completely random. Two out of the six possible orders for the second layer lead to crossings as in `c0`. Therefore the overall probability of a layout with two crossings is $\frac{1}{3}$.

The probability p_{opt_i} of achieving an optimal layout for g_{2_i} once in t runs, is the inverse of the probability of achieving a bad layout in all runs or: $p_{opt_i} = 1 - (\overline{p_i})^t = 1 - (1 - p_i)^t$. Using BU, the layout of the child graphs on the same level of the inclusion tree are always independent of each other. Therefore the overall probability $p_{opt_{bottom\text{-}up}}$ of layouting the complete graph optimally with BU strategy is:

$$p_{opt_{bottom\text{-}up}} = \prod_{i \in |\mathbb{G}_2|} (1 - \overline{p}_i^t)$$

In the example in Figure 4.3 all subgraphs are equal, so using a thoroughness $t = 10$ the probability of a successful layout would be equal to $(1 - \overline{p}_i^t)^{|\mathbb{G}_2|} = (1 - \frac{1}{3}^{10})^7 \sim 100\%$.

Now let us look at HALS. Since there are no cross hierarchy edges, the layout of the subgraphs is again independent of each other. The probability of layouting all graphs correctly in a single run is $p_{run} = \prod_{i \in |\mathbb{G}_2|} p_i$ and the probability of making a single mistake is the inverse $\overline{p}_{run}$. Therefore, the probability of achieving only non-optimal layouts in t runs is $\overline{p}_{run}^t$. Putting it all together we can calculate the probability of an optimal layout by:

$$p_{opt_{hierarchical}} = 1 - \overline{p}_{run}^t = 1 - (1 - \prod_{i \in |\mathbb{G}_2|} p_i)^t.$$

For the example in Figure 4.3 we once again use a thoroughness of $t = 10$ and the fact that all subgraphs are equal with $p_i = \frac{2}{3}$. Then the probability of a successful layout using the hierarchical sweep algorithm would be $1 - (1 - p_i^{|\mathbb{G}_2|})^t = 1 - (1 - \frac{2}{3}^7)^{10} \sim 43\%$.

4.2.2 Speed

We have seen that the solution quality can in some cases suffer when using HALS. What about speed? Does the run time of HALS differ from BU?

Assume a single run of the algorithm, i. e., with thoroughness equal to one. When using any two-layer sweep algorithm, another sweep is executed as long as the number of crossings has decreased in the last sweep (see Algorithm 2). Furthermore assume once again a hierarchical graph $G = (V, E, r, F, P, vp)$, where each simple graph is independent of another and we have no hierarchical edges. Furthermore let there be n simple child graphs in G,

where each simple graph G_i, $i < n$, needs a number s_i of sweeps when laid out separately. How do we find the number of visited nodes in a sweep?

When using BU, we simply sum the sweeps times the number of nodes in the each simple graph, $visited = \sum_{i<n} s_i \cdot |V_i|$, where V_i is the set of nodes in the simple graph G_i.

When using HALS, the number of sweeps is the maximum number of sweeps needed in the simple child graphs, or $s_m = \max_{i<n} s_i$. This is because once again the layout of the simple child graphs is independent of one another, since as before there are no hierarchical edges. Then the number of visited nodes in a sweep is all the nodes in the graph times the maximum number of sweeps, $|V| \cdot s_m$. Since in most cases not all s_i will be equal, this results in more work to be done. Note that not only does the sweep visit more nodes, the crossing counting algorithm must also visit more nodes in each sweep.

In a more general view, BU is a divide-and-conquer approach, where the hierarchical graph is divided up into its simple graphs. This has consequences both for speed and solution quality. For example, when only viewing simple graphs, there are much fewer random decisions to make in each sweep. Therefore it is more likely to find a better solution. Furthermore, the simple graphs are smaller, so cases where we need to sweep across the graph often before reaching a local minimum do not lead to a strong runtime increase. While this does not sound good for the hierarchical sweep approach, bear in mind that we excluded the reason why the hierarchical sweep was considered in the first case, namely hierarchical edges.

These observations suggest that using HALS for all graphs in the inclusion tree is not a good idea. In the following we therefore discuss different solution proposals.

4.3 Solution Proposals

The following sections first show why the intuitive solution of increasing the thoroughness value to improve solution quality is not practical. Then a variant of the hierarchy-aware sweep is developed which combines BU and HALS and switches between them based on a heuristic comparing the influences of randomly positioned nodes and hierarchical edges.

4.3.1 Thoroughness Value

An obvious approach to increase solution quality is to simply increase the thoroughness value. Here, the drawback is just as obvious: The running time increases linearly with the thoroughness value. The success of this approach once again depends on the form of the graph in question. Let us consider a worst case example such as the one shown in Figure 4.3. As shown above, the probability of an optimal layout for this graph is $1 - (1 - \frac{2}{3}^7)^t$. How high must the thoroughness value be to reach a probability equal to the bottom up solution above? If we simply set $(1 - \frac{2}{3}^7)^t = (1 - \frac{1}{3}^{10})^7$, we can solve for t, resulting in $t \sim 150$. So to reach the same probability in the example from above, the thoroughness would have to be 15 times higher, thereby increasing the running time by a factor of at least 15. *At least*, because the

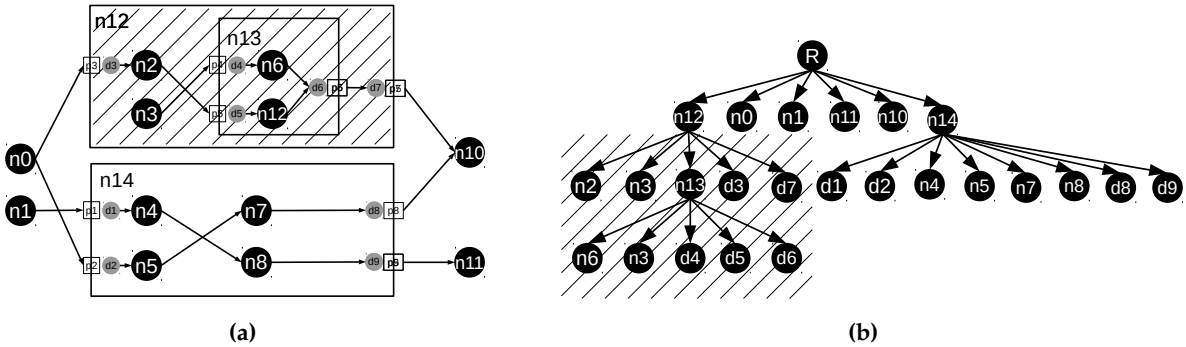


Figure 4.4. (a) shows a hierarchical graph. (b) shows its inclusion tree. There is no reason to sweep into the child graph of `n12`, G_{n12} (hatched), because there is only one hierarchical edge on either side of `n12`. For this reason the list of processed graphs would be (G_{n12}, G_r). After processing G_{n12} separately, `n12` is seen as an atomic node from the perspective of G_r. Note that G_{n12} is once again laid out using HALS.

hierarchical layer sweep will often be slower in any case, due to the considerations described in Section 4.2.2. Once again, this type of graph will probably be quite rare, but still it would be preferable to have a method which scales better.

4.3.2 Choosing Approach per Subgraph

A different idea is to evaluate for each child graph some characteristic to decide whether to layout it together with its parent graph or whether to use a BU approach. To be able to flexibly choose between a BU and hierarchical approach, we need to change the layer sweep algorithm slightly. Before processing the node order, we traverse each simple graph in the inclusion tree in level order, i. e., as returned by a breadth-first-search. For each subgraph we decide whether to allow a hierarchical sweep using the methods described below. In the case where the decision is to layout the simple graph separately from its parent, the graph is pushed onto a stack of graphs. This stack is then processed and *layerSweep* (Algorithm 2) is applied on each of its graphs. As described above, the *sweepForward* (Algorithm 3) and *sweepBackward* algorithms then check for each child graph whether to sweep into it or not. After processing a graph, the parent node's port constraints are set to fixed order. This can then lead to combinations of both approaches as shown in the example in Figure 4.4. The following paragraphs describe an algorithm to decide for each hierarchical node n whether to process its child graph together with the graph containing n.

In a first step, we see that we can exclude the use of the hierarchy-aware sweep in some simple cases. If there are no hierarchical edges (as with all hierarchical nodes in Figure 4.3) or only one hierarchical edge, using BU can not be worse than using the hierarchy-aware sweep. When we have no hierarchical edge to a hierarchical node, we can ignore the child graph and sort its ports without considering hierarchy. When we have up to one hierarchical edge on

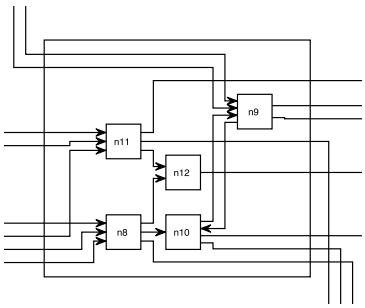


Figure 4.5. Example of a hierarchical node where all nodes in the child graph have only hierarchical influence, i. e., there are no paths to nodes without edges on the eastern or western side of the node.

each side of a node, the conclusion is the same. Note that in ELK Layered using BU, the port order is always set to fixed after each layout of a simple graph. For these cases, the port order must still be changeable.

In a second step, we develop a heuristic which uses characteristics of each child graph to decide whether to sweep into it or process it using BU. The problem elaborated in Section 4.2 only exists in subgraphs where the crossing minimization algorithm must make random decisions. As already noted, the example in Figure 4.3 is one extreme case because it has no hierarchical edges. In this case there is no reason not to layout each child graph separately. Conversely, the hierarchical sweep is maximally beneficial when the order of all nodes in a subgraph is only dependent on the order of the nodes of the parent graph. The example in Figure 4.5 shows such an extreme example, where the child graph is completely dependent on its parent.

The general idea of the heuristic is to judge how strongly the order of the nodes depends on hierarchical edges and how strongly it depends on random decisions.

Heuristic For this we suggest comparing two numbers: Firstly, the number of paths beginning at (or ending in) hierarchical dummies to (or from) all other nodes, which we shall denote as $p_H(G)$ for a simple child graph G. Secondly, the number of paths beginning at (or ending in) nodes with no edges on one side of the node, which we shall denote as $p_R(G)$. In the latter case, the position of the nodes cannot be defined when the sweep comes from the direction of the side with no edges. It is therefore subject to random influences, explicitly in the barycenter algorithm by randomly setting barycenter values and implicitly in the greedy switch algorithm, by using the order previously defined. Figure 4.6 shows examples of paths from a hierarchical port dummy and a node with random influence.

We then normalize and compare the numbers to calculate a measure showing the relation between random and hierarchical influence in the following manner:

$$b(G) = \frac{p_R(G) - p_H(G)}{p_R(G) + p_H(G)} \tag{4.1}$$

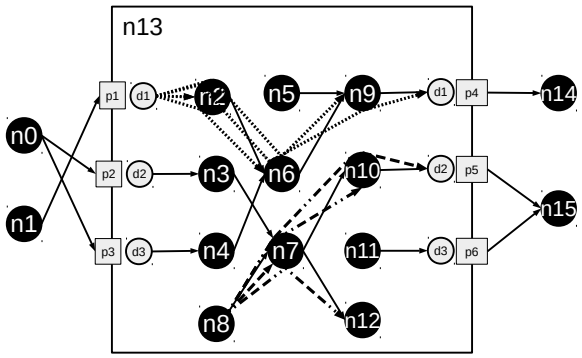


Figure 4.6. Influence of two nodes: `d1` is hierarchical dummy node. It is the start of four paths to other nodes in the graph. These are shown as finely dashed arrows. `n8` has no edges on its west side. In a left-right sweep, its position is therefore dependent on a random barycenter value. It also is the start of four paths to other nodes in the graph. These are shown in dashed and dotted arrows. In total there are 25 paths to and from hierarchical dummy nodes and 11 paths to and from nodes with random initial positioning.

This definition ensures that $-1 \leqslant b(G) \leqslant 1$. The closer the value is to -1, the higher the influence of hierarchical edges. And the closer the value is to 1, the higher the influence of random edges. Then, we can define a threshold value th, where once again $-1 \leqslant th \leqslant 1$. This value can be set by the user or validated experimentally, To be able to choose how likely it is to use a hierarchical sweep, using:

$$hierarchical(G) = \begin{cases} \text{true} & \text{if } b(G) < th \\ \text{false} & \text{otherwise} \end{cases} \tag{4.2}$$

Therefore, if we set $th = -1$, we will always use BU. Conversely, if we set $th = 1$, we will always sweep into the child graphs, except in those cases excluded by the first step, i. e., where there are fewer than two hierarchical edges on each side.

Note that this method only takes into account random decisions which define positions of nodes without connections to a neighboring layer. While this kind of random placement implicitly or explicitly is part of any two-layer crossing minimization algorithm, the barycenter algorithm implemented in ELK Layered adds some random fluctuations to each barycenter, to increase diversity of solutions and to decide on a position when two nodes have the same barycenter value.

Path Counting The problem which remains is how to find the number of paths from and to random nodes and cross-hierarchical dummies. Since at phase three of the Sugiyama Framework (see Section 2.2) we know that the graph is a directed, acyclic and layered graph, this can be done efficiently in a single sweep across the graph. Using these characteristics, we will describe an algorithm to count these paths which runs in $O(|V| + |E|)$. To simplify the description, we only show how to count the number of paths to hierarchical dummies.

Input: V ordered by layers left-to-right
Output: Number of paths p_H coming from or going to hierarchical port dummies

```
1  p_H = 0 for v_s in V do
2      if v_s is western hierarchical dummy then
3          v_s.H = 1
4      end
5      if v_s is eastern hierarchical dummy then
6          p_H += v_s.all
7      end
8      for (v_s, v_t) in E do
9          p_H += |v_s.E_out| · v_s.H
10         v_t.all += v_s.all + 1
11         v_t.H += v_s.H
12     end
13 end
```

Algorithm 4: influenceCount

For each node v we store extra information: The number of paths from a western hierarchical dummy to this node, $v.H$, and the number of all paths ending in this node, $v.all$. We iterate the nodes $v_s \in V$ ordered by layer left-to-right. If a node is a western hierarchical dummy, we set $v_s.H$ to one. If it is an eastern hierarchical dummy, all paths ending in this node are influenced by this dummy, so we add $v_s.all$ to the number of paths to hierarchical nodes, which we denote as p_H. Let $\{(v_s, v_t) \in E\}$ be the outgoing edges of v_s. We increment p_H by the number of outgoing edges multiplied by the number of hierarchical paths to v_s, since each edge is the end of another path from a hierarchical dummy to a normal node. For each edge (v_s, v_t), we then transfer the information of the current node v_s to the target node v_t, by incrementing $v_t.all$ by $v_s.all + 1$ ($+1$ for the current edge) and $v_t.H$ by $v_s.H$.

Algorithm 4 shows this method in pseudocode and Figure 4.6 shows a step by step example.

Counting the number of paths to and from random nodes is the same problem as for cross-hierarchical dummies. The algorithm above can therefore easily be expanded to find both numbers at once, by simply storing for each node the number of paths from nodes with random influence.

A special problem arises from north/south dummy nodes. Algorithm 4 relies on the fact that every node in the set of outgoing edges has not been visited yet. However a northern north/south dummy node will already have been seen when its origin is visited. Therefore we have to deal with a number of special cases: When the target of an edge is a north/south dummy, we execute the transfer step between the dummy and its origin node immediately. If we visit a node with outgoing north/south edges we deal with them in the same manner as other outgoing edges. When we encounter a north/south dummy, we save them for later

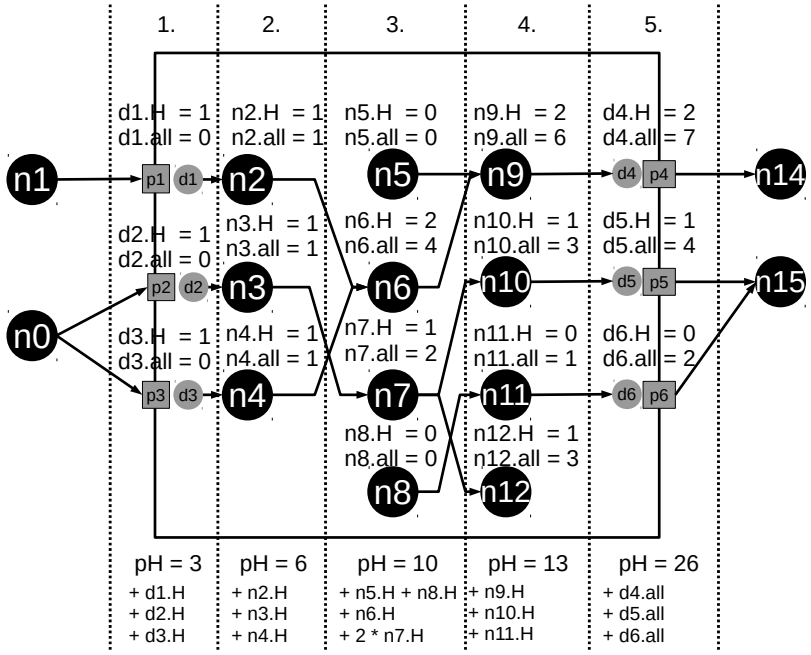


Figure 4.7. Counting the number of paths from (and to) hierarchical dummy nodes to (and from) other nodes. This is the same graph as in Figure 4.6. For a node n, $n.H$ denotes the number of hierarchical dummy nodes which have a path to this node. $n.all$ denotes for a node n the number of all reachable nodes when traversing edges backward from this node. pH denotes the current count of paths to and from hierarchical dummy nodes. The sum underneath this value shows how pH is calculated. The numbering over the graph shows the sequence of layers in the order they are traversed.

processing. Finally, after visiting each node in a layer, we process the north/south dummies in this list as we would a normal node. Before visiting the next layer, we clear the list.

The previous sections explained the hierarchy-aware sweep, its issues and proposals to deal with these problems. The following two sections deal with problems that only indirectly influence the success of HALS. The first is a problem which can occur when we have simple and hierarchical ports on the same node and layout its child graph with BU. The second discusses an efficient algorithm for counting both in-layer and between-layer crossings.

4.4 Further Enhancements

The following two sections deal with two problems which are important for crossing minimization, but not directly part of HALS. However, both topics are interesting enhancements with direct impact on the quality of the algorithm.

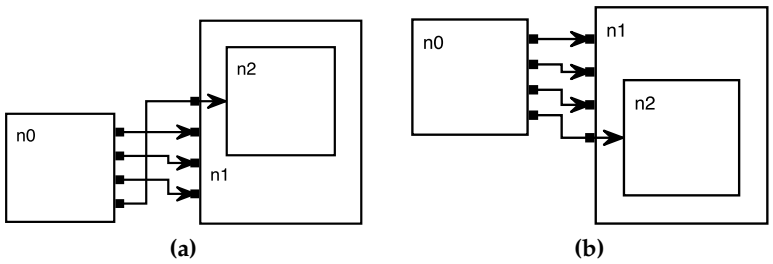


Figure 4.8. (a) shows an example of an Obviously Non-Optimal Graph (O-NO-graph) with crossings caused by fixing the port order on a hierarchical node when using BU. Here, `n0` has fixed port order, while `n1` originally does not, but the port constraints are set to fixed after layouting the child graph separately. (b) shows an obviously better layout.

4.4.1 Sorting Simple and Hierarchical Ports on Same Node

The first is a problem which can arise in BU: Here, the order of the ports for each hierarchical node is fixed and cannot be changed. This makes sense for all edges traversing up the hierarchy, because the child graph has already been laid out. However, there can also be simple edges going into a hierarchical node. In this case, fixing the order of all ports can lead to crossings which could be avoided. This case will probably not occur too often, however when it does, it will certainly be an O-NO-graph, because in most cases removing the crossing can be done by moving the port only by a small distance (Consider the example in Figure 4.8). Therefore it would be helpful if the port sorting algorithm would be able to keep a specific set of ports with the same relative position to each other (the hierarchical ports) and distribute another set of ports among these (the simple ports).

Let us look at the two current crossing minimization algorithms implemented in ELK and see how these could be adapted for this situation.

Greedy Switch Using greedy switch is simple. For two neighboring ports, these are switched as soon as at least one of them is a simple port and the number of crossings decreases. Here, the usual problems for greedy switch apply: It leads to orders which are less optimal than barycenter and has a quadratic complexity.

Barycenter The barycenter port sorting algorithms as presented by Schulze et al. [Sch11] sort the ports by their barycenter values. Adapting these is more difficult. In our case we must keep the relative order of the hierarchical ports, effectively preventing us from sorting on barycenter values. We therefore want to insert the simple ports into the hierarchical ports, keeping the order of the hierarchical ports and the resulting order as *sorted as possible* according to the barycenter values. The degree of *sortedness* of a list can be evaluated using the number of inversions, i. e., pairs of unsorted numbers. This problem can therefore be reduced to the following:

Input: Original array $M = (m_0, \ldots, m_{n-1})$, Element to insert k
Output: Index i_{min}

```
1  inversions = array of length n + 1
2  curr = 0
3  for i = 1 ... n do
4  |  if k < M[i − 1] then
5  |  |  curr = curr + 1
6  |  end
7  |  inversions[i] = curr
8  end
9  curr = 0
10 for i = n − 2 ... 0 do
11 |  if k > M[i + 1] then
12 |  |  curr = curr + 1
13 |  end
14 |  inversions[i]+ = inversions[i]
15 end
16 i_min = arg min_{i⩽n+1} inversions[i]
17 return i_min
```

Algorithm 5: insert

Problem Statement Given are two lists of values $K = (k_0, k_1, \ldots, k_{k-1})$, and $M = (m_0, m_1, \ldots, m_{n-1})$. Insert the elements of K into M without changing the relative order of the elements in M, in such a way as to minimize the number of inversions. A pair (m_i, m_j) from L is an inversion iff $i < j$ and $m_i > m_j$. The number of inversions $inv(L)$ is the number of such pairs.

We now present an $O((n+k)\log k)$ algorithm for this problem.

Algorithm Inserting a single element element k from the second list K is simple: We first try each position in M from left-to-right, accumulating the number of inversions caused at this position and storing the value in an array of length $n+1$, starting at index 1. Remember that n is the number of elements in M. The size of the array must be $n+1$ because the new element can be inserted at $n+1$ positions (including before and after all elements of M). The left-right traversal finds for each possible position the number of times k is smaller than an element to its left. Therefore, when comparing the element k with each element m of M, the accumulated number of inversions increases if $k < m$. We then iterate the list from right to left starting at $m = M[n-2]$, once again accumulating the number of inversions and updating the array. This right-left traversal finds for each possible position the number of times k is larger than an element to its right. Therefore, when comparing our element k with each element m of M, the accumulated number of inversions increases if $k > m$. Using this information we find the index with minimal inversion number. Consider Algorithm 5 for a pseudo-code implementation for inserting a single element.

This algorithm obviously runs in $O(n)$. Now we extend the problem to inserting all elements of K into M.

We denote a *position* x_l to be an element $x \in K$ inserted into M at index l and $L[i]$ the element in L at index i. The number of inversions added by inserting x at l shall be $inv(x_l)$. A *solution* is the tuple of positions for all elements in K, sorted by their index.

To find an efficient algorithm we use the following observation:

Lemma 4.1. *In a solution with minimal number of inversions, the elements of K will always be sorted ascendingly.*

Proof. To prove our observation we show the following: When two elements $x, y \in K$ with $x < y$ are positioned as x_j and y_i in inverted order such that $i < j$, the number of inversions will always be reduced when switching the positions to be x_i and y_j, i. e.,

$$\forall x, y \in K, x < y,\ i, j < |M|, i < j :\ inv(x_j) + inv(y_i) > inv(x_i) + inv(y_j)$$

As a first step, observe that for all elements to the left of i and to the right of j the number of inversions caused with x, y at both positions will not change, because:

With $x^+_{[i,j)} = |\{k \mid i \leqslant k < j, M[k] > x\}|$ as the number of elements in M between the indices i and j larger than x and $x^-_{[i,j)} = |\{k \mid i \leqslant k < j, M[k] < x\}|$ as the number of elements in the same range smaller than x, we see that $inv(x_i) = x^+_{[0,i)} + x^-_{[i,j)} + x^-_{[j,n)}$ and $inv(x_j) = x^+_{[0,i)} + x^+_{[i,j)} + x^-_{[j,n)}$. Remember that $n = |M|$. Here, $x^+_{[0,i)}$ are the elements to the left of index i causing inversions with x and $x^-_{[j,n)}$ are the elements to the right of index j causing inversions with x. Both are contained in $inv(x_i)$ as well as in $inv(x_j)$. The same holds equivalently for y.

We therefore only need to look at the indices between i and j for both x and y. Here, consider the two cases for which the number of inversions could have increased when switching x_j and y_i:

First, an inversion to y_j could be added with an element $L[k]$ at index k with $i < k < j$ and $L[k] > y$. However, because $L[k] > y > x$, $L[k]$ also caused an inversion with x_j, which is removed when moving x to position i. Therefore, no new inversion is introduced.

Second, an inversion to x_i could be added with an element $L[l]$ positioned at index l with $i < l < j$ and $L[l] < x$. However, because $x < L[l] < y$, $L[l]$ also caused an inversion with y_i, which is removed when moving y to position j. Therefore, no new inversion is introduced.

Finally, the positioning x_j and y_i causes an inversion because $x < y$ and $j > i$. This inversion is removed when switching to x_i and y_j so it holds that $inv(x_j) + inv(y_i) < inv(x_i) + inv(y_j)$. □

We use this observation to construct an efficient algorithm traversing the sorted array K as a balanced binary tree:

We sort K before the first call. Using Algorithm 5 we calculate the index i_{minv} with the minimum inversion number for the middle element m_k of K. Using our observation, we now know that all elements to the left of m_k in K must be distributed among the elements to the

left of i_{minv} and all elements to the right of m_k in K must be distributed among the elements to the right of i_{minv}. We use this for our recursive call and place $K[m_k]$ in between the results. To end our recursion we have two cases: If M is empty then return the remaining elements of K and if K is empty we return the remaining elements of M. Algorithm 6 shows a pseudo-code implementation.

Given a sorted K, imagine the calls to the insertion algorithm (Algorithm 5) in the form of binary tree of the sorted array K. Then all elements of L are distributed with no overlapping between all nodes at each level of the tree of K. Since Algorithm 5 runs in $O(n)$, and the height of the tree is $\log k$ the algorithm runs in $O(n \log k)$. See Figure 4.9 for an example. Since sorting K is in $O(k \log k)$, we have a total of $O((n+k)\log k)$.

We now turn to a topic which is not related to hierarchy, but plays an important role in crossing minimization: Counting the number of crossings.

4.4.2 Efficient In-Layer and Between Layer Crossings Counting

As described in Section 3.3, the current in-layer crossing counter has worse than quadratic runtime. Aside from the run-time considerations, splitting up in-layer and between-layer crossings into separate steps leads to an unwieldy implementation, with a greater amount of code to maintain. Since counting crossings is a large part of the expense of a sweep, it is worthwhile to make this as efficient as possible. Hence, this section shows a fast and simple algorithm for counting both in- and between-layer crossings at the same time.

Before describing our new algorithm, we will first take a closer look at the between-layer edge crossings counting algorithm as suggested by Barth et al. [BJM02] and show why we cannot use the same method for counting in-layer edge crossings. The following description is based on the explanation in my bachelor's thesis [Sch15].

Input: $M = (m_0, \ldots, m_{n-1})$, List of elements to insert K
1 **if** *M is empty* **then**
2 | **return** K
3 **end**
4 **if** *K is empty* **then**
5 | **return** L
6 **end**
7 $mid = |M| \div 2$
8 $k = M[mid]$
9 $m_k = minIndex(M, k)$
10 $left = insert((m_0, \ldots, m_{m_k}), (k_0, \ldots, k_{mid-1}))$
11 $right = insert((m_{m_k+1}, \ldots, m_{n-1}), (k_{mid+1}, \ldots, k_n))$
12 **return** $left$ ++ (k) ++ $right$

Algorithm 6: merge

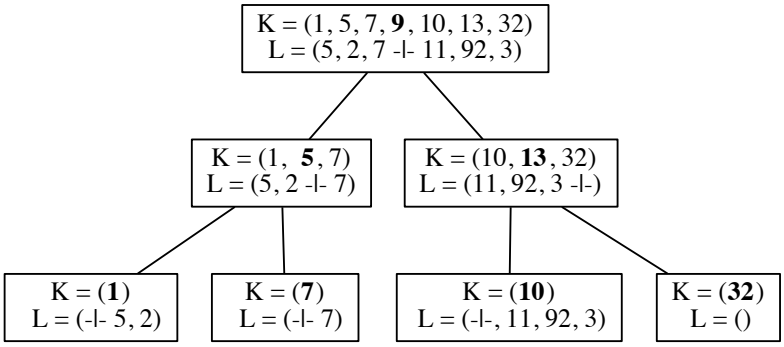


Figure 4.9. Example showing the arguments of the recursive calls to Algorithm 5. The calls are shown as a binary search tree on the list K. The number in bold font is the middle element, and $-|-$ shows the index where it will be inserted. Note how the elements of L are distributed among each level of the tree. The final answer will be: $(1, 5, 2, 5, 7, 7, 9, 10, 11, 92, 3, 13, 32)$. There are other orders with minimal inversion number. The solution returned by Algorithm 5 depends on the implementation of arg min. For the answer above, we take the rightmost index with minimal inversion number when inserting a single element.

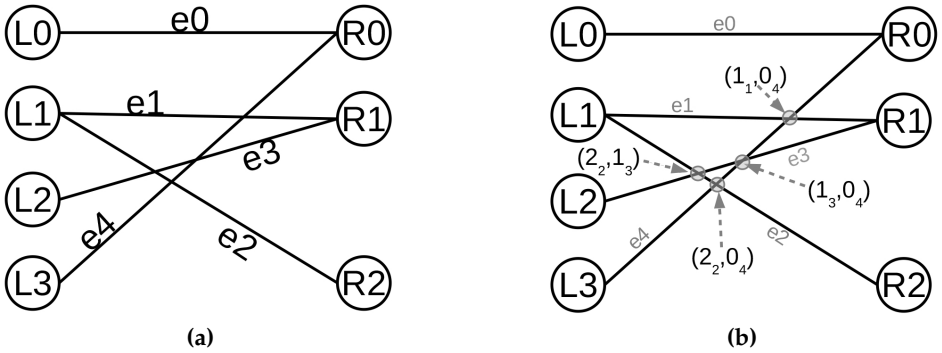


Figure 4.10. (a) shows the lexicographical sort order of the edges. (b) shows the correspondence of edge crossings to the inversions.

Original Between-Layer Edge Crossings Counter Between-layer edges in a two-layer graph (L_l, L_r) consisting of a left layer L_l and a right layer L_r can be sorted lexicographically in such a way that in $\pi_E = (e_0, \ldots, e_{|E|-1})$ for each pair of edges $e_i, e_j \in \pi_E$, it holds that:

$$\begin{aligned} e_i = \{l_i, r_i\} < \{l_j, r_j\} = e_j \text{ iff} \\ pos_v(l_i) < pos_v(l_j) \text{ or } pos_v(l_i) = pos_v(l_j) \text{ and } pos_v(r_i) < pos_v(r_j). \end{aligned} \tag{4.3}$$

We now take a look at a sequence π consisting of the position values of the nodes in the right layer as they occur in π_E. Note that each position value can occur multiple times, because there can be multiple edges incident to the same node. The number of between-layer crossings then corresponds to the number of *inversions* in π. Remember the definition of inversions from above: A pair (p_i, p_j) from π is an inversion iff $i < j$ and $p_i > p_j$.

As an example, let us take a look at the sequence of the position values of the nodes in the right layer sorted by their occurrence in π_E in Figure 4.10. Here, we have $\pi = (0_0, 1_1, 2_2, 1_3, 0_4)$ using the form $rightNodePosition_{edgeNumber}$. The inversions and their corresponding crossings are marked in Figure 4.10b.

Counting can be simply and efficiently done using a modified merge or other comparison based sort algorithm, a Fenwick tree, or as chosen by Barth et al. an *accumulation tree*. All of these choices result in an $O(|E| \log |V_{small}|)$ running time, where V_{small} is the number of nodes in the layer with fewer nodes. Note that the edges need to be sorted in a preliminary step. For this step to not dominate the complexity, Barth et al. suggest the use of radix sort. A comparison based sort would lead to an $O(|E| \log |E|)$ complexity. Since collecting the edges will follow the order of the nodes, in many cases the edges will already be sorted by the first condition in Equation 4.3: $pos_v(l_i) < pos_v(l_j)$. Therefore, in practice, using an adaptive comparison based sorting algorithm will often be faster. This is the manner in which it is implemented in ELK.

This algorithm is explicitly designed for a two-layer graph. To the best of our knowledge, it cannot be adapted and used for counting in-layer edge crossing. This can be seen by examining the differences between in-layer and between-layer crossings:

Definition 4.2 (In-layer edge crossings)**.** Given in-layer edges as pairs of node positions (i, j) with $i, j \in |L|$, two edges $e_0 = (i, j)$ $e_1 = (k, l)$ can be chosen such that without loss of generality $i < k$. Then there exists a crossing between e_0 and e_1 iff $k < j \wedge l > j$. Figure 4.11 shows the different possibilities for e_0 and e_1.

Definition 4.3 (Between-layer edge crossings)**.** Given between-layer edges as pairs of node positions, two edges $e_0 = (i, j)$ and $e_1 = (k, l)$ with $i \neq k$ can be chosen such that without loss of generality $i < k$. Then there exists a crossing between e_0 and e_1 iff $j > l$. Note that if $i = k$ or $j = l$, there is no crossing.

As we can see, even when we sort in-layer edges by their source positions, crossings are dependent on both the position of the source and the target of an edge, e. g., for the nomenclature used in Definition 4.2 both k and l. On the other hand, after sorting the edges as described above, between-layer crossings are only dependent on the position of the target nodes, i. e., j and l as used in Definition 4.3.

Therefore we will now introduce an efficient algorithm for counting in-layer crossings, and then show how the same algorithm can be used to count between-layer crossings at the same time.

Preliminaries For the first step, we are given a graph containing only in-layer edges. Furthermore, we will not consider a port-based graph, but instead assume that all edges directly connect two nodes.

Ignoring ports simplifies our examples and is sufficient for our purposes: Since the ports are sorted during the layer sweep (see Section 4.1.1), we can assume the port order to be fixed. In this case the iteration over the nodes can simply be replaced by an iteration over ports.

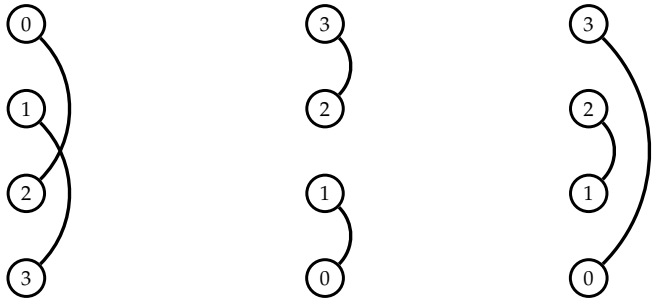


Figure 4.11. Different possibilities for two in-layer edges with no two edges incident to the same node.

Lastly, we will assume that the graph has no self-loops, i. e., edges whose target and source are the same port. Note that in the case of the reduced port-less graph for this algorithm, self-loops are edges connecting a single node with itself. In an implementation, these can simply be skipped, since with proper edge routing, these edges will not cause any crossings.

Note that this type of graph is equivalent to a type of graph drawing called *one-page book*, *circular*, *outerplanar* or *convex*. See He and Sýkora [HS04]. Here, as in many other publications, algorithms for crossing minimization in these types of graphs are proposed, however to the best of our knowledge, no efficient algorithm for counting crossings has been suggested.

Counting In-Layer Crossings Counting crossings then works as follows: We traverse the nodes and store the end position of each edge sorted by position in a data-structure T. Since multiple edges can end in the same node, the same position can be entered in T multiple times. Each time we visit a new node n we remove all entries equal to the position of n. Then, for each new end position $p_{e_{end}}$ of an edge e, e will cross as many edges as the index of $p_{e_{end}}$ in T. As an intuition, the entries in T are the edges which have started before n and will end after the current node position. The index of the $p_{e_{end}}$ shows the number of edges which will end before $p_{e_{end}}$ and therefore will cross the current edge. Algorithm 7 shows pseudo-code for this procedure.

As an example, which is illustrated in Figure 4.12, consider the following edges: $(0,3)$, $(1,2)$, $(1,4)$, $(2,5)$. Assume that we have already visited v_0, then we have currently stored only the end of the first edge, so $T = (3)$. We now visit v_1. There are no entries in T equal to 1, so we delete nothing from T. Then, for each edge of v_1, we check at which index in the list we would add the end node position. The edge $(1,2)$ ends at 2. If we would add this to the list, it would be at index 0, adding no crossings. The edge $(1,4)$ ends at 4. If we would add this to the list, it would be at index 1, so we add one crossing. Next, we add all edge ends to T resulting in $T = (2,3,4)$. Note that in case we would have already added them before updating the crossing number, 4 would have been at index 2, and we would have counted an extra crossing. We continue with v_2. We remove its position 2 from T, resulting in $T = (3,4)$ The only edge $(2,5)$ from this node ends in 5, which would now be at index 2, so we add 2 more crossings,

Input: Nodes $V = (m_0, \ldots, v_{k-1})$
Output: Number of crossings c

```
1  c ← 0
2  T ← binary indexed tree with size k
3  for v_i in V do
4  |   T.remove(i)
5  |   for e = (v_i, v_j) in E_d(v_i) = {e|e ∈ E(v_i) ∧ j > i ∧ j ≠ i} do
6  |   |   c ← c + T.sum(j)
7  |   end
8  |   for e = (v_i, v_j) in E_d(v_i) do
9  |   |   T.increment(j)
10 |   end
11 end
```

Algorithm 7: countInLayerCrossings

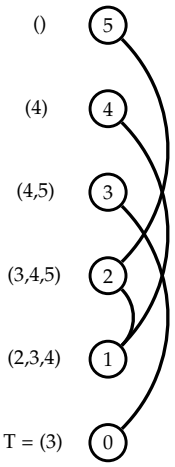
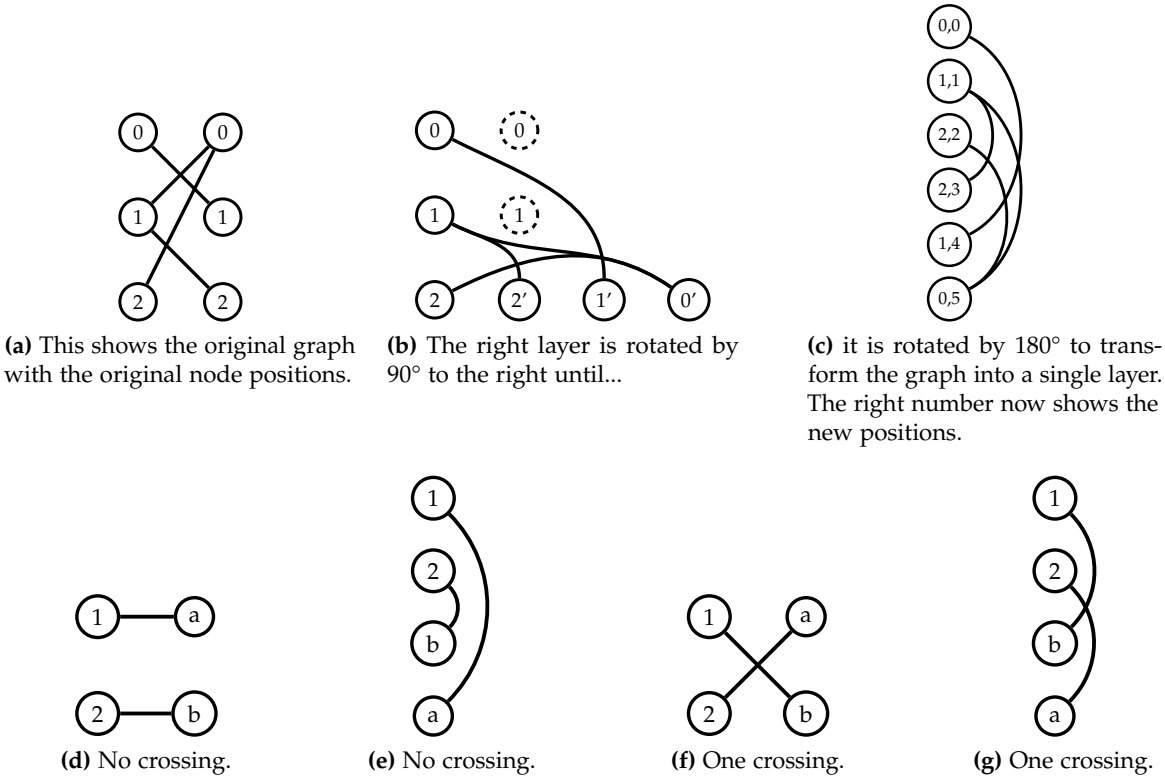


Figure 4.12. Example for in-layer crossings with edges $(0,3), (1,2), (1,4), (2,5)$. The annotations beside the nodes show the content of T after the node beside each annotation has been visited.

resulting in the expected total of 3 crossings. The visits at all remaining nodes only delete the nodes as they have no more edges pointing toward nodes with higher positions.

Why is this algorithm efficient? In a preprocessing step, we collect the position of each node. This takes $O(|V|)$ time. We then traverse all edges and nodes, conducting insertion, looking for the index of a value and removing all instances of a value. The main open question is therefore the choice of data-structure for T. To be faster than the existing in-layer cross counting algorithm, all of these operations must run in at most $O(\log n)$ time.

There are different possible choices for this data-structure. A possible choice would be to use an *order statistic tree*. This is a binary search tree augmented by the operations $rank(x)$ which finds the index in the sorted list of elements of the tree and usually $select(i)$ which



(a) This shows the original graph with the original node positions.

(b) The right layer is rotated by 90° to the right until...

(c) it is rotated by 180° to transform the graph into a single layer. The right number now shows the new positions.

(d) No crossing.

(e) No crossing.

(f) One crossing.

(g) One crossing.

Figure 4.13. Intuition behind counting between-layer edge crossings in the same way as in-layer edge-crossings. (a), (b) and (c) show the rotation of a complete graph. The lower four graphs show two transformations possible.

finds the i-th smallest element stored in the tree. This could be backed with any kind of tree such as a perfect binary tree or a self-balancing tree such as a Red-Black tree.

Instead we chose a *Binary Indexed Tree (*BIT) or *Fenwick Tree* [Fen94]. Its most attractive feature is its extraordinarily simple implementation. In its original proposal, it supports two operations: $GetCumul(i)$, which returns the sum of the entries below and including the index i and $putValue(i,k)$, which adds to the entry in index i the value k. Since we always only increment a single value, this can be simplified to an $increment(i)$ operation, incrementing the entry at index i. Furthermore, we add a $remove(i)$ method which removes all entries at one index i. To do this, we must save the exact values for each entry in addition to the data needed for the Fenwick Tree. Then we lookup the value x at i and perform $putValue(i,-x)$ and set the exact value to 0.

All of the methods of BITs require $O(\log n)$ time, where n is the maximum number of elements which can be added to the BIT. In our case, n is the number of the nodes in the layer.

In this way, counting in-layer crossings has a running-time of $O(|V| + |E| \log |V|)$.

Counting Between-Layer Crossings We do not only want to count in-layer crossings, but crossings involving between-layer edges as well. Using a simple trick, we can convert all edges between layers to a problem which can be solved using the algorithm described above. Let us consider a two-level graph with between-layer edges as in Figure 4.13a.

Let (i, j) and (k, l) be two between-layer edges, with $i \neq k$ and $i < k$. Then we have a crossing when $j > l$, as defined in Definition 4.3. We transform these crossings to in-layer crossings by numbering the nodes in a different manner: We keep the numbering the nodes of the left layer: Numbered top-down with 0 to $|L_0| - 1$ and change the numbering of the nodes of the right layer by labeling them bottom-up with $|L_0|$ to $|L_0| + |L_1| - 1$. As before, it holds that $i < k$. Since the indices in the right layer are all larger than in the left layer, it now holds that $k < j$. Furthermore, due to the new numbering we now have $j < l$. This fulfills the condition for in-layer crossings as in Definition 4.2.

Now, we traverse the nodes in the order of this new numbering: Down the left layer, and up the right layer, executing the same algorithm as before.

Intuitively, the right layer is rotated around its lowest node and added to the bottom of the left layer, essentially transforming between-layer crossings to in-layer crossings. See Figure 4.13 for an illustration which also shows the two possible transformations of a pair of edges, once with a crossing, and once without a crossing.

We can now use this algorithm for counting both in- and between-layer crossings in the same traversal of both layers. This is because the areas of the in-layer crossings for both layers still do not overlap. Therefore they remain unchanged from the perspective of the counting algorithm when the numbering of the nodes is updated.

Concluding Remarks This chapter showed a number of different contributions to the topic of minimizing crossings in hierarchical graphs. It described HALS which was extended by a heuristic to flexibly switch between separating simple graphs and sweeping across them together with their parents. This was done because depending on the characteristics of the graph, in some cases solution quality and running time can worsen. We then examined further issues related to crossing minimization, namely sorting ports on nodes with only partly fixed port orders and efficiently counting crossings of in-layer and between-layer edges.

We now turn to some practical considerations and describe the most important features and ideas behind the implementation into ELK Layered.

Chapter 5

Integration into ELK Layered

A major goal for the hierarchical crossing minimization phase is to be as easy to maintain as possible. To reach this goal, we will first examine the current architecture and define what makes easy maintainability in our case. Then the architecture chosen for hierarchical crossing minimization is described. The description of the current architecture in ELK Layered is partly based on the one of my bachelor's thesis [Sch15].

5.1 ELK Layered

ELK Layered principally follows the framework suggested by Sugiyama et al. However, it is designed to be able to deal with more types of graphs than described in the original paper. ELK Layered does not assume that all graphs which are given to the algorithm are acyclic. Therefore another phase is added at the beginning of the algorithm to remove the cycles. The removed edges are then re-added at the end of the algorithm. Another issue dealt with by ELK Layered in a separate phase is edge routing, resulting in a total number of five phases.

To name just a few of the cases considered during the layout not covered by the Sugiyama algorithm, the graphs processed by ELK are port-based graphs and contain labels, hyperedges (edges connecting more than two nodes) and comment nodes.

To keep complexity under control and to have sufficient flexibility in the implementation, all cases that the five main phases were not specifically designed for are kept separate in so-called *Intermediate Processors* [Sch11], which are executed in between the phases. This modular structure of the algorithm enhances the freedom to adapt and extend the algorithm. For a simple overview, see Figure 5.1.

For the algorithm to know when to execute the intermediate processors, each phase must specify their dependencies. This defines in between which phases which intermediate processor must be run. The execution order of processors which are in between the same phases must be manually defined by the programmer.

The list of processors $proc(G)$ for a simple graph G depends on the specifics of each graph. For example, special processors are activated when north/south ports are used. Others depend on layout options chosen by the user, such as whether edges should be drawn orthogonally or with splines. This is also true for each simple graph in the inclusion tree, so while one graph might have one list of processors, the graphs containing its nodes' ancestors or descendants might well have others.

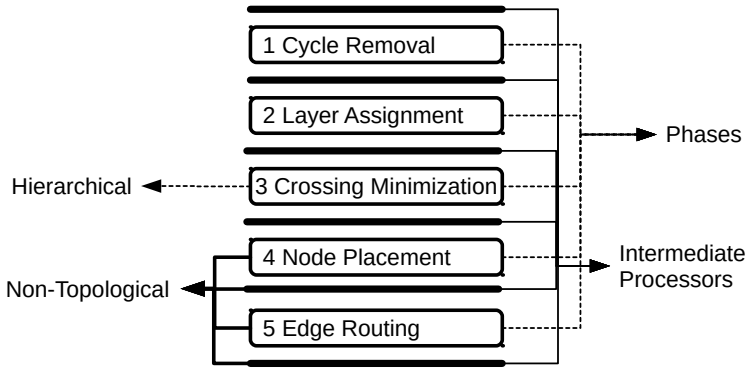


Figure 5.1. Overview of ELK Layered's architecture.

Phases four and five, the intermediate processors between and after these phases and some of the intermediate processors between phases three and four change the exact coordinates of the drawing, while phases one through three do not. These processors are marked as *non-topological* in Figure 5.1.

The algorithm is controlled by the `ELKLayered` class which steps through the list of processors and executes one after another. As described above, the BU method for laying out hierarchical graphs ensured that all descendants of each graph were completely processed before starting the algorithm.

To be able to implement processors which operate on the complete hierarchy, the algorithm for controlling the processors must be changed. The following section describes the aim and details of the solution chosen here.

5.2 Design of Hierarchy Aware Layer Sweep

5.2.1 Maintainable Processor Control

A major goal of the implementation of HALS is to keep the code as maintainable as possible. This leads to a few important consequences:

A We split the processors into hierarchical and non-hierarchical processors. As stated, using BU reduces the complexity by enabling the programmer to only deal with simple graphs when working on a processor. Therefore this advantage must be kept on all processors that do not have to be hierarchical.

B There must be no processors which need two separate implementations, i. e., once for simple and once for hierarchical graphs. In the case of hierarchical processors, the developer has to ensure that the processor can deal with both simple and hierarchical graphs.

C The controlling `ELKLayered` class should not need to differentiate between hierarchical and simple graphs, but should be able to execute the same method for both.

```
Input: Graph G = (V, E, F, r, P, vp) with inclusion tree T_i = (V, F, r)
Output: laid out Graph G
1  simpleGraphs = breadthFirstSearch(T_i)
2  simpleGraphs = reverse(simpleGraphs)
3  while proc(simple(r)) is not empty do
4      for G' in simpleGraphs do
5          while proc(G') is not empty do
6              p = remove head from proc(G')
7              if (p is not hierarchical) then
8                  run p
9              end
10             else if G' == simple(r) then
11                 run p
12                 break
13             end
14             else
15                 break
16             end
17         end
18     end
19 end
```

Algorithm 8: layout

The *layout(Graph)* algorithm (Algorithm 8) shows the way these requirements were fulfilled. In a first step the graphs in the inclusion tree are collected using a breadth-first search. This list is then reversed. In this manner each graph in the list is before the graph containing its parent node. If we were to execute all processors on each graph in the order given by this list, it would behave the same as BU. The algorithm ends as soon as the last processor of the root graph is executed. Since an executed processor is always removed from the processor list, this is the case when $proc(G)$ is empty, G being the root graph. We then step through the list of graphs and execute a processor normally if it is not marked as hierarchical. If it is marked as hierarchical, only the root graph may execute it. If the current graph is not the root graph, the execution of the processors on this level is stopped and the current processor removed from the processor list of this graph. In this way, hierarchical processors are never executed on any graph that is not the root graph. When the hierarchical processor has been executed by the root graph, the execution of the processors is continued with the graphs in the lowest hierarchy level.

For this thesis, only processors dealing with crossing minimization were implemented to be hierarchical. To be able to judge whether other phases could also work on the complete hierarchy, let us consider Figure 5.1 once again. The processors marked as *non-topological* change the exact coordinates of the graph, e. g., the size of a parent node or the exact position of external ports. For this reason, these processors must be executed the same way as with BU.

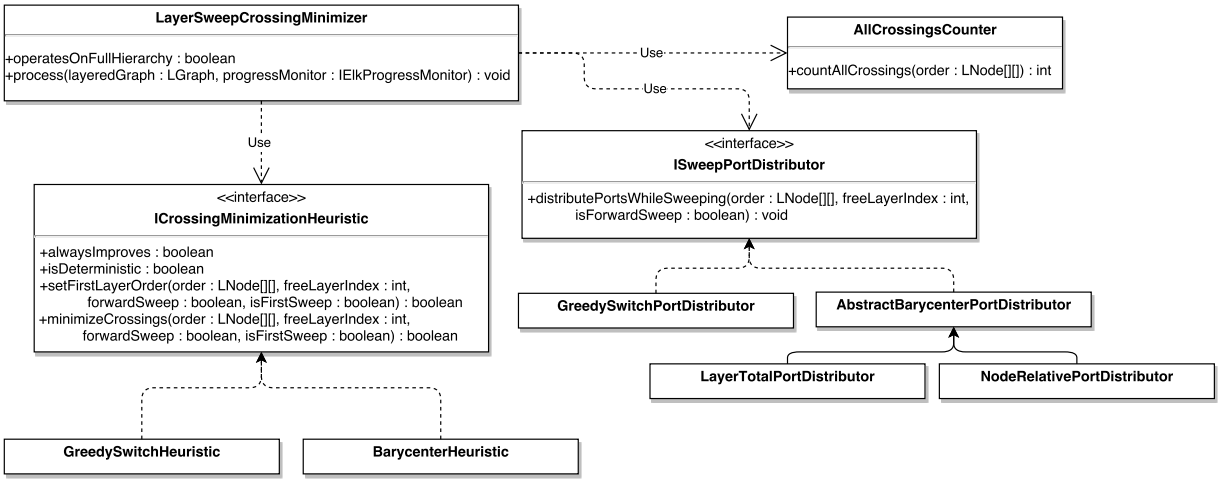


Figure 5.2. Overview of the classes to be changed for further modifications.

Otherwise it is impossible to know the exact sizes of the hierarchical nodes or to calculate the positions of their ports. Therefore, it is not allowed to add a hierarchical processor in this area. This is enforced in the implementation by not permitting processors marked as hierarchical to be placed after the start of phase four. When implementing other processors, developers must take care never to change exact coordinates before any hierarchical processors. Furthermore, they must not work across hierarchies in processors which are not explicitly hierarchical, i. e., they must not access child or parent graphs.

5.2.2 Extending Crossing Minimization

The implementation of the layer sweep algorithm was chosen in such a manner as to enable a flexible development and exchange of some parts of the algorithm. This includes the two-layer crossing minimization heuristic, the crossings counter algorithm and the port sorting heuristic. For an overview, see Figure 5.2.

To integrate a new two-layer crossing minimization heuristic, the interface `ICrossingMinimizationHeuristic` must be implemented. It requires four methods:

The method `alwaysImproves` determines whether the heuristic can only improve the number of crossings. This is for example the case when using two-sided greedy switch, as described in my bachelor's thesis [Sch15]. This is necessary because the layer sweep class can refrain from counting crossings and only needs to continue sweeping when the order of the nodes has changed.

The method `isDeterministic` defines whether an algorithm, given a specific order in the input, will always return the same order or not. If this method returns false, the layer sweep is repeated as many times as defined by the *thoroughness* parameter and the order with the lowest number of crossings is taken.

`setFirstLayerOrder` is needed because the order of the first layer is often dealt with in a specific manner. For example, while barycenter sets a random order to achieve a higher diversity in the solutions, greedy switch when used as a post-processor wants to build on the results set by the barycenter processor and therefore does not change the order.

Finally, `minimizeCrossings` minimizes crossings in the layer indicated by `freeLayerIndex`. We pass the order of all the nodes in the graph, because in cases such as the two-sided greedy switch, we take the layers on both sides of the free layer into account. The barycenter heuristic as implemented in ELK Layered assigns random barycenters to nodes with no connection to the sweep direction only in the first sweep across the graph. In all following sweeps the barycenter value is calculated from the order created by the last sweep. For this, a flag `isFirstSweep` is needed.

A new crossing minimization heuristic will almost always make it necessary to adapt the same heuristic to sort ports. For example the two-sided greedy switch heuristic could lead to endless loops, if the corresponding port sorting heuristic cannot guarantee that the number of crossings is always reduced. For this we use the `SweepPortDistributor` interface which defines the method `distributePortsWhileSweeping`.

Currently, three crossing counting algorithms exist, for which we can define no joint interface because they solve different problems: While the `NorthSouthEdgeCrossingsCounter` only counts crossings of north/south ports in a single layer, the `HyperedgeCrossingsCounter` and the `CrossingsCounter` count between-layer crossings. However, there is currently no way to count in-layer edge hyperedges. Instead crossings of in-layer edges which will be drawn as hyperedges are counted as normal in-layer edge crossings by the `CrossingsCounter`. To have a joint place to deal with counting all edge crossings, a utility class `AllCrossingsCounter` collects all methods to have a central place to call `countAllCrossings`.

Finally, it might be an interesting alternative to implement a global crossing minimization processor. However, this would have to be done in a completely new crossing minimization processor.

Chapter 6

Experimental Evaluation

This chapter presents the results of the experimental evaluation of HALS. The first section will shortly introduce the datasets the experiments were performed on. The second section discusses the solution quality and performance of applying the new algorithm in relation to the heuristic threshold value. Finally, in the third section we try to find graph characteristics which influence the success of HALS for random graphs with different properties.

6.1 Datasets

6.1.1 SCGs with Basic Blocks

A Sequentially Constructive Graph (SCG) with basic blocks is a graphical representation of a synchronous programming language for safety critical applications [vHDM+14]. Basic blocks are parts of the program which can be executed monolithically. They are part of the compilation process. In the context of the development tools for SCG, this compilation process can be visualized step by step. Showing basic blocks is part of this visualization.

Figure 6.1 shows an example of an SCG with basic blocks. The basic blocks are shown by the light gray unfilled boxes around at most two nodes and for easy processing by the layout algorithm are modeled as hierarchical nodes with the parent node modeling the basic block.

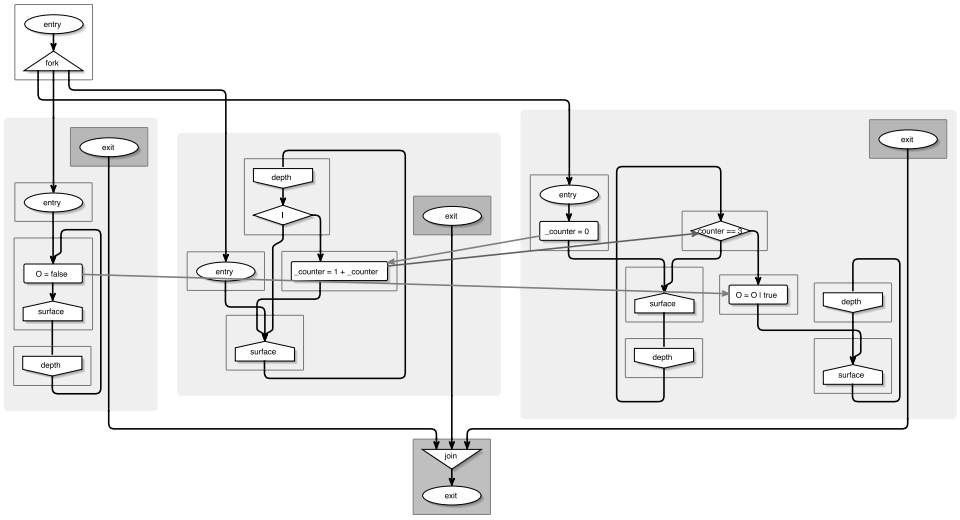
The dataset consists of 106 graphs none of which fail to meet the criteria described previously (hierarchical nodes, more than two hierarchical edges, at least one edge crossing).

The largest difference to the other datasets is the number of simple child nodes contained within a single hierarchical node. Here the average mean number of such nodes is only 1.1 with an average standard deviation of 0.01.

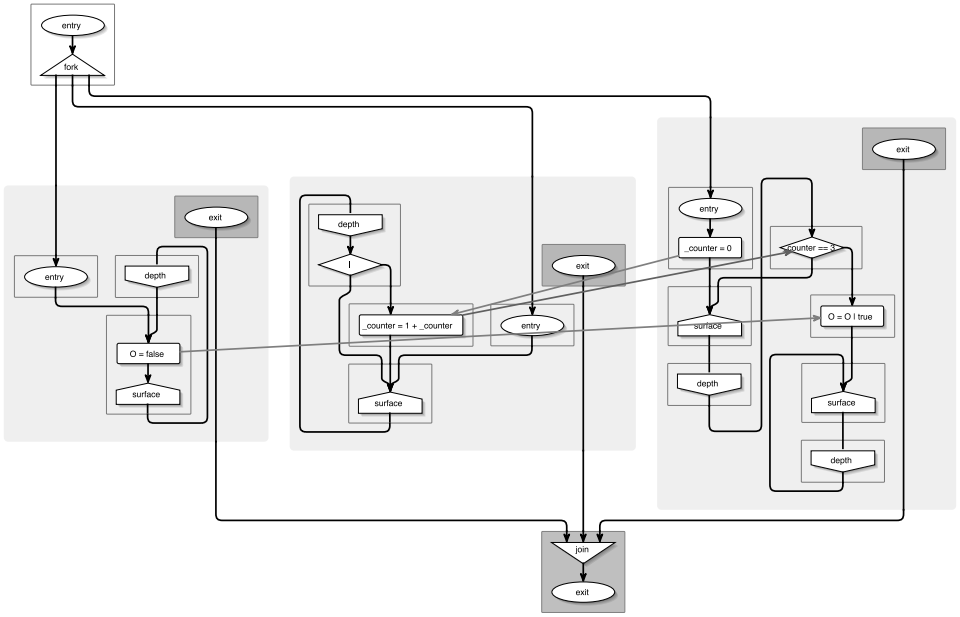
6.1.2 Ptolemy Graphs

Ptolemy is a project studying modeling, simulation and design of concurrent, real-time and embedded systems.[1] ELK Layered is integrated into the system and can be used to lay out the models. The dataset originally contained 194 graphs. To only use graphs relevant for reducing crossings in hierarchical graphs, we filtered out graphs containing no hierarchical nodes and fewer than two edges. Furthermore, we also excluded graphs which had no edge crossings. This reduced the dataset to 139 graphs.

[1] See `Ptolemy.eecs.berkeley.edu`, accessed 16/05/27



(a) BU: Six edge crossings.



(b) HALS: One edge crossing.

Figure 6.1. Comparing BU and HALS on SCG. The straight edges across the graph show dependencies in the compilation process and are not laid out using layered layout.

The dataset still remains heterogeneous. The size of the graphs vary widely, with a mean of 103.0 nodes per graph and standard deviation of 134.1. The hierarchical edges almost always connect simple nodes and only rarely end on the outside of a hierarchical node. The largest difference to the SCG is the number of simple nodes in a hierarchical node. Here we

```
generate 500 graphs {
  nodes = 10 +/- 6 { // normally distributed number of simple nodes per
      hierarchical node (incl. root).
    remove isolated // no nodes without incoming edges
    ports {
      re-use = 0.1 // More than one edge leaving a port leads to hyperedges
    }
  }
  edges relative = 0.7 to 1.2 // Equally distributed number of edges relative to
      the number of simple nodes within a hierarchical node
  hierarchy {
    nodes = 3 +/- 1.5 // Number of hierarchical nodes in each hierarchical node
    edges relative = 0.025 to 0.3 // Total number of hierarchical edges relative to
        the number of nodes.
    levels = 3 // Maximum depth of inclusion tree.
  }
}
```

Listing 6.1. Specification in the random graph creation DSL used to generate 500 strongly varying random graphs. The comments show the effect of each statement.

have an average mean of 7.7 nodes (calculated across all hierarchical nodes of a graph) with an average standard deviation of 3.7. Figure 6.2 shows an example of a Ptolemy graph.

6.1.3 Random Graphs

Since the number of graphs in the other datasets described below is quite small for an experimental evaluation, a DSL for the flexible generation of random graphs with specific characteristics was implemented. Listing 6.1 shows the specification used to generate 500 random graphs with strongly varying characteristics. Here, as in the following sections, the exact semantics of the DSL will be explained in the comments of each listing.

6.2 Quality and Speed

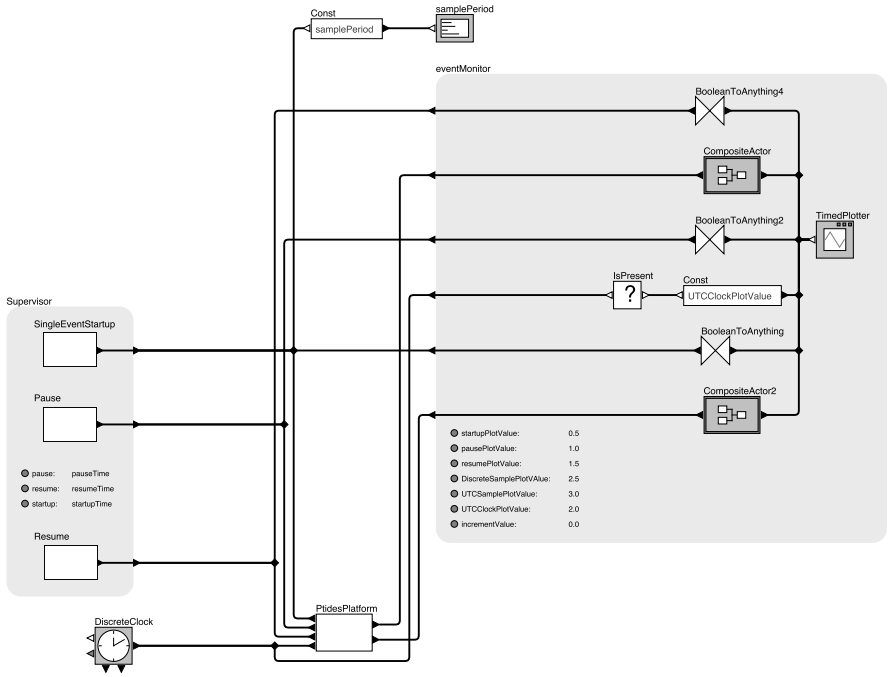
This section answers a number of questions:

A) How does HALS perform when not using the heuristic?

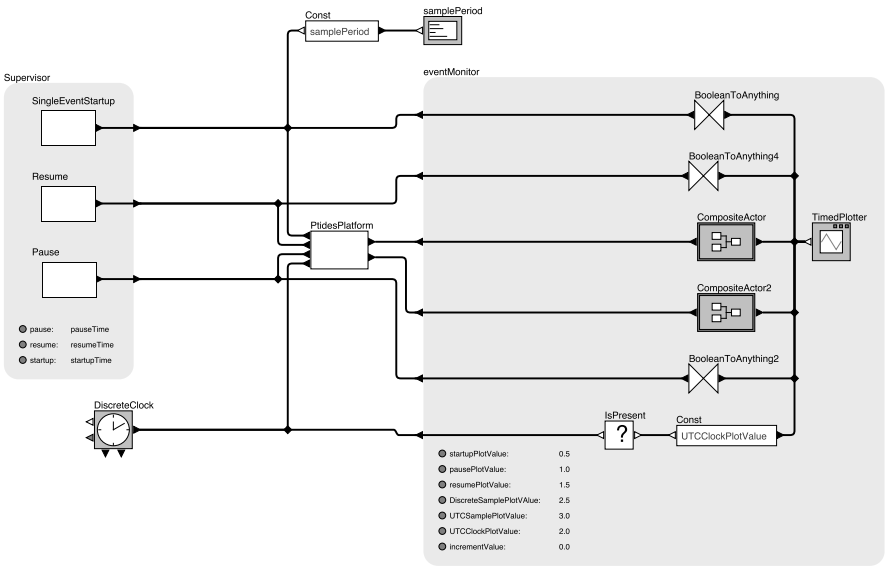
B) Does the heuristic perform better than a random choice?

C) Is there some boundary value which can be set as a sensible default value?

D) How does the algorithm perform on different datasets?



(a) BU: Seven edge crossings.



(b) HALS: Two edge crossings.

Figure 6.2. Comparing BU and HALS on Ptolemy graphs. The shadowed boxes are hierarchical nodes. The nodes such as `CompositeActor` with the symbol of three connected nodes can be expanded by the user to hierarchical nodes with a double click to show the containing child graph.

As described in Section 4.3.2, the decision heuristic compares the number of paths to nodes with random influence to the number of paths to nodes with hierarchical influence. This is normalized to return values between -1 and $+1$, where -1 is the same as always using the BU approach and $+1$ is equal to always sweeping into the child graph except for those cases where no improvement is possible by doing so.

6.2.1 Setup

Percent Change We use percent change instead of absolute values in order be able to compare crossing numbers and running time independent of the size of the graph. Percent change is calculated by $\frac{new-old}{old}$ or in the case of the crossing number $\frac{c_i-c_{BU}}{c_{BU}}$ where c_i is the number of crossings for boundary value i and c_{BU} is the number of crossings when using BU. This is equivalent to setting i to -1, i. e., $c_{-1} = c_{BU}$. As described before, all graphs with no crossings in the BU setting were removed from the datasets, thereby preventing divisions by zero. Note that a negative percent change indicates a reduction in the number of crossings compared to using the BU strategy, while a positive percent change shows an increase of crossings. While a negative percent change can at most be -100%, i. e., in the case when no crossings remain, there is no limit to an increase of crossings.

Crossing Numbers For showing the change in crossing number we use box plots. As an example see Figure 6.3. Box plots show the distribution of the data: The top and bottom edges of the boxes show the first and third quartiles and the fliers are positioned at a distance of $1.5 \cdot IQR$, where IQR is the inter-quartile range or the distance between the first and third quartile. All points outside of the fliers are considered to be outliers (shown as $+$-signs in the plot). Note that in some settings, there is a large partition of graphs with no change, resulting in a lot of the data being centered at zero. In the case where 50% or more of the data is at the same point, the fliers disappear due to the way they are calculated. In this case, all other values are shown as outliers. The median of the data is shown as a bold line within the box. In the case where the median lies at the same point as one of the box edges, 25% of the data lies on the median. The mean of the data is shown as a small black point. In many cases, the median stays quite stable, while the mean often changes more strongly toward improved values. This shows that while the number of graphs with improved crossings stays equal, the amount of improvement in graphs with improved crossing number increases. We show the change in crossing number by using one box plot per threshold value.

Running Times As described in Section 4.2.2, the more often the algorithm sweeps into a child graph, the more nodes and edges are visited per sweep and the higher the probability of sweeping more often. To measure this, we need to take a look at the speed of the algorithm. Instead of measuring actual running time, we count the number of edges and nodes visited during an execution of the layer sweep algorithm using the barycenter heuristic for node and port sorting. Remember that the run-time of the barycenter algorithm is in $O(|E| + |V| \log |V|)$ for each layer. We use this method instead of actual running time to have a more stable

measure not prone to random fluctuations caused by things such as Java JIT compiler, caching effects or other processes competing for system resources. Once again we compute the percent change in number of visited edges and nodes compared to only using the BU algorithm. For each threshold value we take the average value across all graphs. To find possible trade-offs between running time and solution quality, we plot the change in crossing number and the average increase of visited edges and nodes directly above each other using the same x-axis.

Threshold Values For each dataset we tried all boundary values from -1 to $+1$ with a resolution of 0.1. For each graph and each boundary value, we count the number of crossings and then calculate the percent change to the number of crossings when using the BU strategy, i. e., when the boundary is set to -1. Remember that the heuristic is used to increase the success of the method without changing the thoroughness value as described in Section 4.3.2.

Random and Always on We plot two other settings in the same manner: As described in Section 4.3.2, even when setting the threshold value to the maximum value of 1.0, the algorithm does not sweep into a child graph in the cases where no improvement can be made. To examine the effect of this decision, we show the results for the original idea of always sweeping into every child graph. The other part of the plot shows a setting where the decision whether to sweep into the child graph is simply randomized. We use this as a baseline to check whether the heuristic makes reasonable decisions. Both settings are compared to a threshold value of 0.1.

We shall now analyze and interpret the results for each datasets, starting with the random dataset, followed by the Ptolemy graphs and finally the SCGs. In each case, we split up the plot into regions of similar behavior. For quickly finding each region, they are shown explicitly in each plot, annotated in the description and the plot with capital letters.

6.2.2 Random Graphs

We first examine the results on the random graphs described in Section 6.1.3. Figure 6.3 shows the plots in the form described above We can roughly divide the results into three regions:

The first region (A) has threshold values of -0.9 to -0.5. Here, the change between the values is small. At least 50% of the graphs have equal or only slightly improved crossing numbers and almost all outliers lie within $\pm$ 50% difference. The average number of visited nodes and edges also stays mostly the same.

The second region (B) spans threshold values of -0.5 to 0.5. Here, the amount of improvement increases steadily. In the same period the amount of graphs with worsened layouts stays roughly the same. The number of visited nodes and edges quickly increases, up to about 35%.

Starting at a threshold value of about 0.5 and upwards (region C), there are more graphs whose crossing numbers increase. After a threshold of 0.7, the layout of most graphs stays the same, which can be seen in the rare changes in crossing number as well as in the only very slight increase of visited nodes and edges.

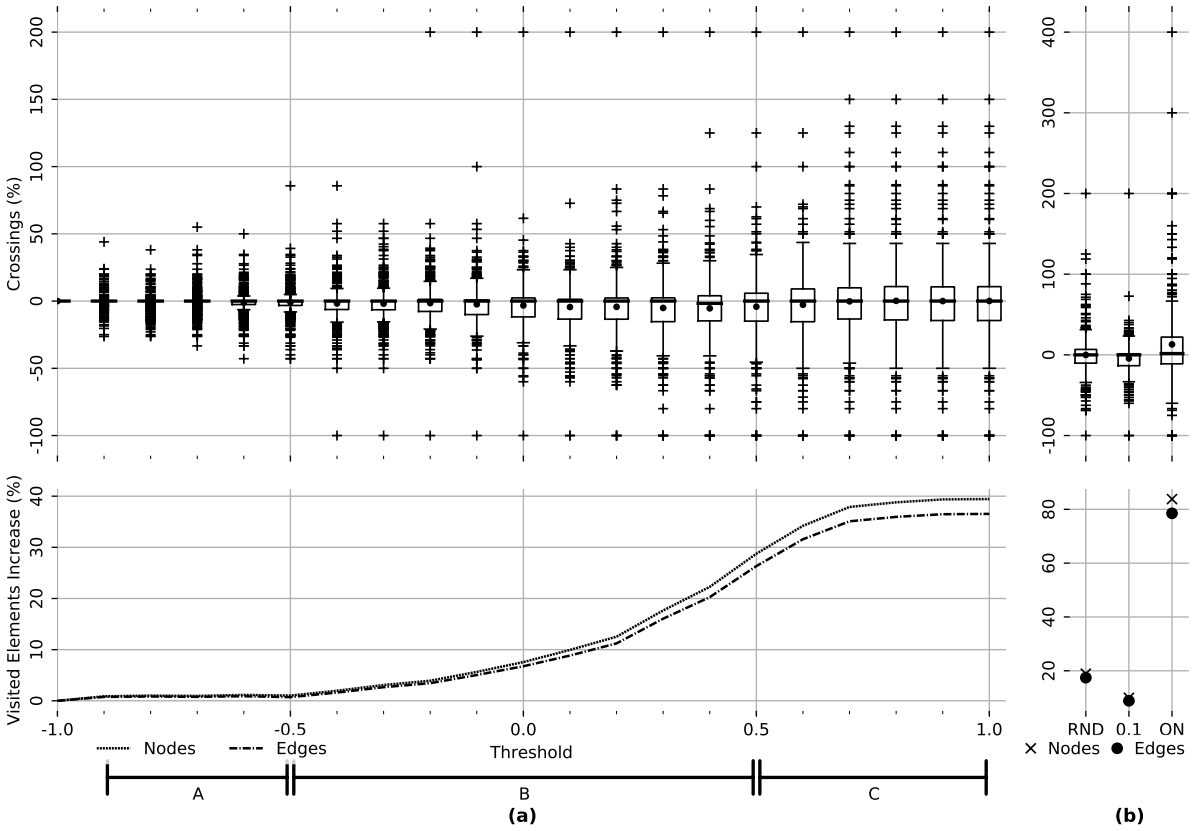


Figure 6.3. Random graphs: Plot (a) shows success and runtime for the against different thresholds for the heuristic. Plot (b) compares random choice (RND), a threshold of 0.1 and always sweeping into child graphs (ON).

In general, only at a threshold value of 0.4 does a majority of graphs have improved crossing numbers. In all other cases, the median percent change is exactly at 0. Note however that up to a threshold of 0.3, 25% or more graphs have unchanged crossing numbers. The spread of outliers increases with a higher threshold value, including one graph with all crossings removed and one graph where the number of crossings has tripled (i. e., a 200% change).

The right hand plot in Figure 6.3 shows the same type of plot for a random choice (RND), for setting the threshold to 0.1 and for always sweeping into the graph (ON). It can be seen that using the heuristic with a setting of 0.1 is more successful and faster compared to the base case of a random setting. Always sweeping into the graph results in significantly worse performance. In the latter case we can observe an increase in the number of crossings in a majority of cases including outliers with a very high increase of crossing numbers. The number of visited nodes also increases by 83%.

The measured data shows the evenly distributed randomness of this dataset. At both ends of the spectrum, the crossing numbers and visited elements only change slowly. In the

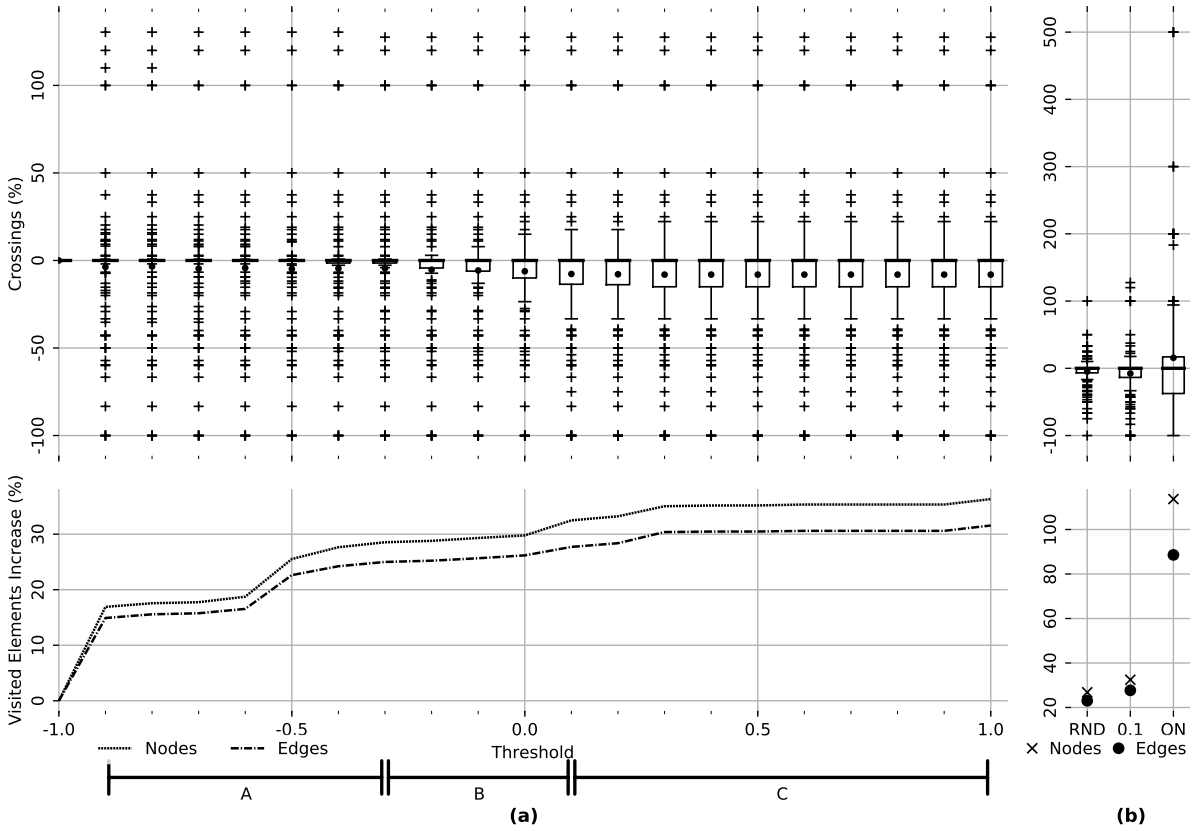


Figure 6.4. Ptolemy graphs: Plot (a) shows success and runtime plotted against different thresholds for the heuristic. Plot (b) compares random choice (RND), a threshold of 0.1 and always sweeping into child graphs (ON).

region of -1.0 to -0.5, the algorithm is mostly identical to the BU setting. In the upper region on the other hand, the heuristic almost always chooses to sweep into the graph. Noticeable improvement only happens in the region of -0.4 to 0.5. Choosing a value in this region can then be traded against running time: While the only value with a majority of improved graphs is at 0.4, the difference to 0.1 is small, but the percent change in number of visited elements compared to BU has increased by 10%.

6.2.3 Ptolemy Graphs

Next, we examine the results on the Ptolemy graphs described in Section 6.1.2. Figure 6.4 shows the first of the two plots. Once again, we can divide the plot into three regions:

In the region from -0.9 to -0.3 (region A), the results stay roughly equal, with 50% of the graphs remaining unchanged (up to -0.5) or with only slight improvement (-0.4 and -0.3). After a jump of 27% increase in the number of nodes visited after activation of the algorithm with a threshold value of -0.9, the number of visited elements rises to a 38% increase in

number of nodes visited. The amount of improvement increases in the range between -0.2 and 0.1 (region B). Starting at the value 0.2 and above (region C), both the crossing numbers and the number of visited nodes and edges show no further change.

Compared to the random graphs examined previously, the spread of the outliers stays constantly large. In all cases, the number of improved graphs are not in the majority. However, also in all cases, at least 25% of all crossing numbers remain unchanged.

The right hand plot in Figure 6.4 once again compares random choice, always-on and a threshold value of 0.1. Using the heuristic returns better results but slower performance than a random choice. Once again, always sweeping into each child graph leads to a very strong increase in visited nodes and edges, with an average of 114% increase. The crossing number change shows a much larger variation up to an extreme outlier with 6 times the number of crossings compared to BU.

The measurements of this real world dataset show less smooth characteristics than the random dataset. The amount of improvement compared to BU is strongest above a value of 0.1 after which the numbers change only a little. Since the number of visited elements continues to increase up to a threshold value of 0.3, there is no reason to choose any value above 0.1. Since the increase in the number of visited elements between -0.4 and 0.1 is small, once again, 0.1 seems to be a good default threshold setting for the heuristic.

6.2.4 SCGs with Basic Blocks

Next, we examine the results on the SCGs described in Section 6.1.1. Figure 6.5 shows the first of the two plots. This dataset shows the strongest decrease of crossing numbers and the strongest increase in visited elements.

Following a jump after the activation of the algorithm the graphs stay the same up to a threshold value of -0.3 (region A). Here, the median improvement of crossing numbers is at -37 % and the number of visited nodes increases quite strongly by 40 %. The large difference between the increase in visited edges and nodes could be explained by the form of the graph: Many simple graphs in the input graph have only one node. For any hierarchical edge incident to that node, an extra hierarchical dummy node is created. This can lead to cases where a single node in the input graph with a single incident hierarchical node leads to two nodes (the node and the hierarchical dummy) and only one edge being visited in the crossing minimization phase.

Region B shows a short range of decrease in crossing number up to an improvement of -63% at a threshold value of -0.1, after which neither the crossing numbers nor the number of visited elements change (region C). Interestingly, the increase of visited elements is lessened at the same time. The average increase in the number of visited nodes is only 34% in region C. This is contrary to the expectations formulated in Section 4.3.2, and presumably is due to the characteristics of this type of graph. In cases where the number of crossings is zero after the first sweep, the algorithm stops sweeping immediately.

The right hand plot in Figure 6.5 compares a random choice with the use of the heuristic and always sweeping into every graph. Randomly deciding still improves the crossing number

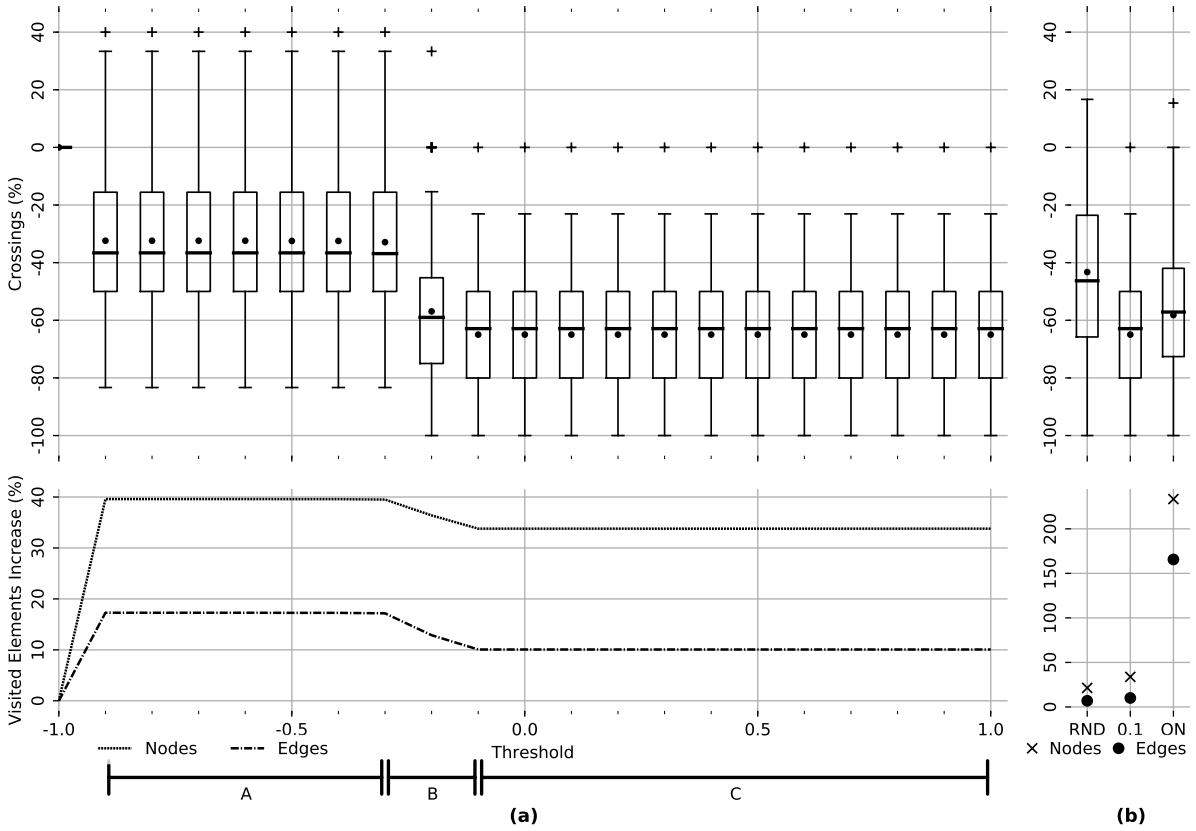


Figure 6.5. SCGs with basic blocks: Plot (a) shows success and runtime plotted against different thresholds for the heuristic. Plot (b) compares random choice (RND), a threshold of 0.1 and always sweeping into child graphs (ON).

compared to BU, but much less than at a threshold of 0.1. Always sweeping into the graph returns worse results than when using the heuristic, with a median improvement of 57% and a high increase of visited nodes to 233% compared to BU.

Even though looking at the number of visited nodes and edges has the benefit of being an exact measurement, we take a short look at measured computation time for the SCGs dataset. We do this to because it simplifies the interpretation of the percentage increase of visited elements. We take the minimum of five measured values in an attempt to compensate for the problems of measuring running time.

The graph in the SCG dataset with the slowest layout time needs 40 ms for crossing minimization using BU. The same graph needed 61 ms when setting the threshold to 1.0 which in the case of the SCGs is the same running time as when using a threshold such as 0.1, as can be expected from the measurements shown in Figure 6.5. Crossing Minimization took 167 ms when always sweeping into the graph. The average crossing minimization time for an SCG using BU was 4 ms, 6 ms with 1.0 and 16 ms when always sweeping into the graph. To set these times into perspective, the complete layout including all phases needed on average

across all graphs in the dataset 40 ms when using BU. For the layout of the largest graph as above it needed 144 ms with BU.

Since the number of simple nodes in most hierarchical graphs in the SCG is one, fixing the port orders of the parent nodes leads to a large number of nodes with fixed port order. This effectively prevents the success of the crossing minimization, stopping the algorithm very early. Also, since most child graphs have many paths with hierarchical influence, HALS mostly sweeps across the complete hierarchy, increasing the number of visited nodes and edges. While the median crossing number is best starting at a threshold value of -0.5, a good trade-off between solution quality and performance could already be at a threshold value of -0.9.

6.2.5 General Interpretation

We now answer the questions posed at the beginning of this section.

A) How does the algorithm perform on different datasets?
The success of the algorithm is very dependent on the type of graph. The number of graphs with equal or improved crossing numbers is greater than the number of graphs with worsened crossing number in all datasets. However, the SCG dataset shows a much stronger improvement in crossing numbers than all other datasets. The running time is always slower, with up to 40 % increase in the number of visited nodes.

B) How does HALS perform when not using the heuristic?
In all cases, using the heuristic is much better and faster than sweeping into every graph.

C) Does the heuristic perform better than a random choice?
For all datasets, using a threshold value of 0.1 returns superior values than random choice.

D) Is there some boundary value which can be set as a sensible default value?
In general, the best threshold value strongly depends upon the dataset in question. In all cases, a value of 0.4 is one of the threshold settings that leads to the highest improvement, however when trading off runtime and quality, the data for random and Ptolemy datasets seem to suggest a value more like 0.1. Choosing between these values then also depends on whether slower run-time turns out to be an actual problem in practical applications. Note that choosing a value above zero also removes crossings in trivial examples where the number of paths with hierarchical influence is equal to the number of paths with random influence as shown in Figure 6.6.

6.3 Graph Characteristics Influencing Layout Quality

The previous results suggest that the success of HALS strongly depends on the characteristics of the graphs. This section examines a number of different graph types using the random

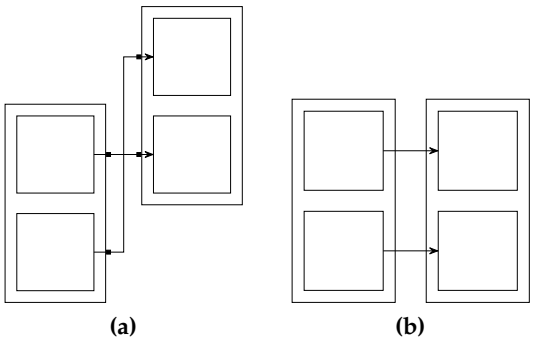


Figure 6.6. Example of a simple graph where the number of paths with hierarchical influence is equal to the number of paths with random influence. The value calculated according to Equation 4.1 is zero. The graph is swept into only if this value is strictly smaller than the threshold as defined in Equation 4.2. Therefore with the threshold value set to zero, we still keep the O-NO layout of (a). Setting the threshold to any value greater than zero returns the layout in (b).

graph DSL described earlier. To examine the effect of one specific characteristic, such as the number of hierarchical edges, we keep the specification of the rest of the graph equal and only change one element. This can be done using a range declaration in the DSL, as can be seen in Listing 6.2.

6.3.1 Increasing Hierarchical Edges

The specification shown in Listing 6.2 increases the absolute number of hierarchical edges from 0 to 50 on graphs containing six simple nodes and three hierarchical nodes per simple graph with an inclusion tree depth of two and 1.1 simple edges per simple node. For each setting, two graphs are created. Figure 6.7a shows the average percent change in crossing numbers comparing BU to hierarchy-aware sweep, setting the heuristic threshold to 1.0.

In most graphs, above a certain number of hierarchical edges we can see a slight improvement in crossing numbers. In general, however, there is no clear correlation between the absolute number of hierarchical edges and the relative improvement

A similar result can be seen in Figure 6.7b. Here we created graphs with a similar specification as above except that we set the number of simple edges to zero, creating graphs whose edges are all hierarchical. As expected, many crossings are removed using HALS for this setting. However, the improvement is less for graphs with more hierarchical edges. Presumably, this is because the complexity of the graphs increases with the number of edges, and there are simply fewer crossings which can be removed by the algorithm in any case.

6.3.2 Increasing Number of Simple Child Nodes

The predominant feature of the SCG graphs with basic blocks is the small size of the simple child graphs. To verify the assumption that this is a reason for the large improvements

```
generate per configuration 2 graphs { // Generate two graphs per value in range
    declaration.
  hierarchy {
    nodes = 3 // Fixed number of hierarchical nodes in each hierarchical node
    edges total = range 0 : 1 : 50 // Range declaration: From 0 to 50, with a
        stepping size of one.
    levels = 2 // Maximum depth of inclusion tree.
  }
  nodes = 6 { // number of simple nodes per hierarchical node (incl. root).
    remove isolated // no nodes without incoming edges
  }
  edges relative = 1.1 // Fixed number of edges relative to the number of simple
      nodes within a hierarchical node
}
```

Listing 6.2. Random graph DSL specification for increasing number of hierarchical edges using the `range` statement

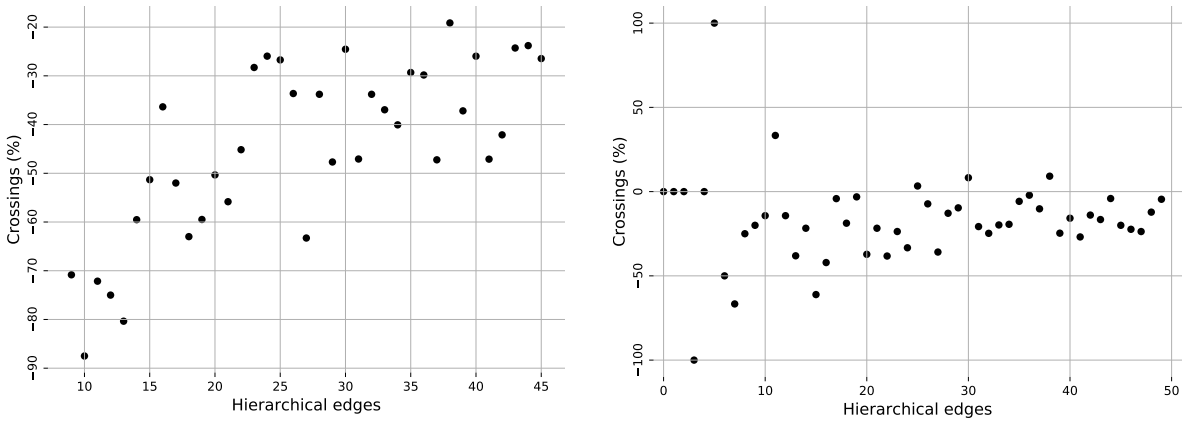


Figure 6.7. (a) shows average crossing numbers while increasing hierarchical edges in graphs containing simple edges. (b) shows the same with no simple edges except between dummy nodes.

using the hierarchy aware sweep, we compare the percent change for random graphs when increasing the number of nodes in simple child graphs. For this we increased the number of simple nodes from one to ten, while keeping the other specifications constant. In this case we used 1.1 edges per simple node, four hierarchical nodes per hierarchical graph with an inclusion tree of depth 3 and 0.5 hierarchical edges per node.

Figure 6.8 shows the results of this experiment. As expected, the hierarchy aware sweep is especially effective for graphs with few simple nodes and less effective, the larger the simple graphs are.

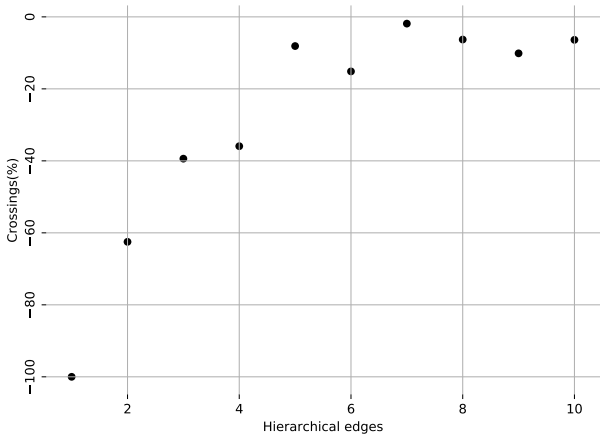


Figure 6.8. Increase number of simple nodes.

Conclusion In this chapter we evaluated the success of HALS using three datasets: Random graphs, Ptolemy graphs and SCG with basic blocks. The evaluation showed that the success of the algorithm varies greatly between different types of graphs, however in all cases using HALS resulted in a majority of graphs with improved or equal crossing numbers. The findings suggested a reasonable default value for the heuristic threshold value to be 0.1. We furthermore found that the algorithm is especially effective in graphs with few simple child nodes.

The following chapter summarizes the results and contributions of this thesis and describes open questions as well as further interesting avenues of research.

Chapter 7

Conclusions

In this thesis we examined an extension of the crossing minimization step in hierarchical layered graphs which we call Hierarchy-Aware Layer Sweep (HALS). It was developed and implemented as a part of ELK Layered, an algorithm for automatic layout of layered graphs. The main goal was to improve the crossing minimization of hierarchical graphs, while at the same time not placing a burden on the maintenance of the code-base.

This extension is based on the local layering scheme, where each child graph has its own separate set of layers. Previously, ELK Layered used the Bottom-Up (BU) approach, where the layout of each child graph and its parent node was completed before the layout of the graph containing its parent node. For each simple graph, the layer sweep method was applied: The layers are visited two at a time, keeping the order of one layer fixed while permuting the order of the other. The two-layer crossing minimization problem can be solved by using one of many heuristics, where currently the barycenter and greedy switch heuristics are implemented in ELK Layered. Using HALS, on each hierarchical node we sort the ports after each re-sorting of the free layer. Then, the port dummy nodes on the sweep side of the hierarchical node are sorted according to the order of their respective hierarchical ports. Finally, the algorithm proceeds to sweep across the child graph.

We showed that there are many cases where this intuitive idea for HALS leads to worse layouts with slower running times. This is due to the fact that BU is a divide-and-conquer approach, where the algorithm is executed separately on much smaller graphs. Since the two-layer crossing minimization heuristics are non-deterministic, the chance is higher to make a better combination of the right random decisions. Furthermore, the algorithm must sweep across the complete hierarchical graph and not only on the smaller child graphs, increasing computation time. To alleviate these caveats, we developed and implemented a heuristic which decides for each child graph whether or not to sweep into it. To do this, the heuristic compares the influence of hierarchical edges to the influence of nodes whose position is determined randomly.

Furthermore, we presented two other contributions. Firstly, an efficient algorithm for counting both in-layer and between-layer edge crossings. Counting edge crossings is an important part of crossing minimization. Previous efforts have only enabled efficient counting of between-layer edge crossings and an inefficient algorithm for counting in-layer crossings. Secondly, we show an efficient port-sorting heuristic using barycenters for the case where some of the ports have a fixed order but others can be freely inserted with the goal of minimizing edge crossings.

All of the algorithms except for the port insertion were implemented in ELK Layered. HALS was evaluated using two real-world datasets and a set of random graphs. The results show that when using the heuristic, the number of crossings in most hierarchical graphs is reduced. However, the effectiveness of the algorithm strongly depends on the characteristics of the graph. The improvement in crossing number comes at the price of an increase in running time, with an up to 40% higher number of nodes visited.

7.1 Future Work

To conclude this thesis, we examine areas of future research related to crossing minimization, hierarchical graphs and ELK Layered.

When dealing with crossing minimization, there is a very large number of heuristics which have been developed. When using a two-layer heuristic, the changes in ELK Layered which have occurred in the context of this thesis enable future programmers to implement other heuristics for crossing minimization and port sorting with no change to the rest of the algorithm. While all other heuristics are slower than barycenter and its variants (see the comparisons by Jünger et al. [JM97] and Martì et al. [ML03]), the size of the current graph or child graph could be used to switch between barycenter and slower but more effective algorithms. Experience shows that O-NO-graphs (for an example of an O-NO-graph, see Figure 4.8) are more common in cases when the graph is small, because in large graphs it is often more difficult to see how to remove edge crossings. Whether or not this is correct could be examined in a user study. If the assumption proves to be correct, this might make such an approach interesting.

While two-layer crossing minimization algorithms can now easily be integrated into ELK Layered, the crossing minimization processor would have to be rewritten for the use of a global crossing minimizer. In some cases, the method for adapting a global crossing minimizer to a locally layered hierarchical graph is easily apparent. Sifting, for example, is a quadratic algorithm which has been adapted to be used as a global crossing minimizer by Bachmeier et al. [BBBH10] and Matuszewski et al. [MSM99]. In sifting, starting from a predefined order, for each node every possible position is considered while keeping the relative order of the other nodes fixed. To use this for a locally layered hierarchical graph, these algorithms would only have to be changed in two manners: They must be able to chose any node in any child graph and when changing the position of a port dummy, its corresponding port must be moved at the same time.

As we showed for crossing counting in Section 3.3, a two-layered graph can be transformed into a one-page book drawing, while keeping the same crossings as before. There is a large amount of literature on crossing minimization in one-page book drawings, which could conceivably be adapted for the two-layer case, with the restriction that nodes must be restricted to stay on their layers.

Hyperedges are edges connecting a tuple $H = (S, T)$ of source ports S and target ports T. Crossings of hyperedges not only depend on the ordering of the nodes in the layers,

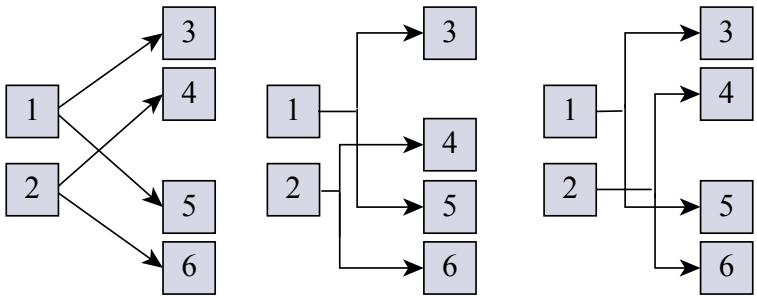


Figure 7.1. Crossings of hyperedges depend on the actual position of the nodes.

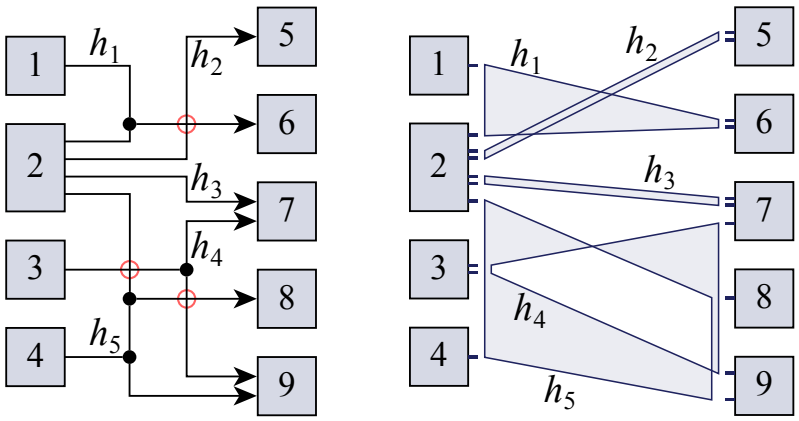
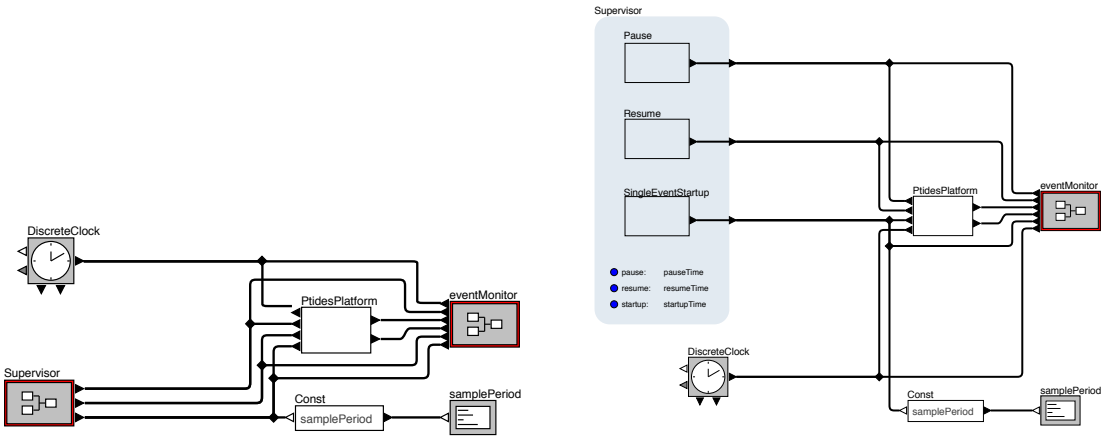


Figure 7.2. The crossing estimation algorithm counts four crossings: Three crossings between h4 and h5 because the virtual edges (2, 8) and (3, 7) cross and the ranges spanned by the corners overlap in both layers. Example taken from Spönemann et al. [SSRvH14].

but also on the positioning of the nodes and the arrangement of vertical line segments. See Figure 1.1 for an example. In layer based graph layouts the crossing minimization knows nothing about hyperedges and minimizes crossings between normal edges. These are then converted into hyperedges in a later stage of the algorithm, where a hyperedge combines every edge going into the same port. As an extension for improving the effectiveness of crossing minimization, Spönemann et al. [SSRvH14] developed an algorithm to estimate the number of hyperedge crossings the resulting node order will create. For this it transforms a hyperedge into polygons with four corners: The uppermost and lowermost ports of the hyperedge on both layers. It then counts the number of crossings between virtual edges formed by the upper corners of each polygon formed by a hyperedge and adds the number of overlapping areas of the hyperedges on both layers. See Figure 7.2 for an example of this method. This algorithm currently cannot estimate the number of in-layer hyperedge crossings. Using the idea for counting normal edge crossings elaborated in Section 3.3, it might be possible to adapt the hyperedge crossing counter to also take these into account. The only



(a) All hierarchical nodes contracted. **(b)** `Supervisor` node unfolded.

Figure 7.3. Example of a Ptolemy graph where the expanding of a hierarchical node changes the order of the nodes and the drawing of the hyperedges in the graph containing the hierarchical node, making it harder for the user to understand the data flow graph.

differences to the algorithm would be three things. First, we create the virtual edges using in-layer edges as well as between-layer edges. Secondly, we can then use the new crossing counting algorithm to count the crossings between the top virtual edges. Thirdly, we must adapt the range detection to work on the transformed port indices.

The algorithm suggested in Section 4.4.1 for inserting ports with no fixed order into a list of ports whose order cannot be changed is currently not implemented in ELK Layered. Implementing this, evaluating it and considering other algorithms is a very interesting possible future research topic. Note that it might not only be interesting for the case of BU layout of hierarchical graphs with edges that are not all hierarchical. It could also be a feature interesting to users, where they would be able to specify a fixed order for some ports, while leaving the order of the other ports to the algorithm.

As Raitner pointed out [Rai05], a feature enabling the user to contract and expand hierarchical nodes can lead to changes of the order of the nodes in the parent graph. These changes interfere with what is called the *mental map* of the users, since now the placement of the nodes has changed from what they are used to. As an example see Figure 7.3. Since the ease of understanding the content of the graph is important in the layout of data flow diagrams, this is an issue which could be addressed. The problem could obviously be solved by using a top-down approach, laying out the parent graph first and fixing the port orders for the child graphs. This way, when a hierarchical node is expanded, the order of the ports and therefore the nodes around the node will stay the same. However, this will obviously lead to the same problems which we tried to solve in this thesis. An alternative would be to layout the complete graph in the background while only showing the user the reduced graph with contracted nodes. The drawback of this would be that in the case of large graphs,

contracting nodes would not lead to faster computation of the graph layout. When using contractable nodes as modules however when developing in languages such as SCGs, the size of the complete expanded graph can grow exponentially with the use of more and more abstract modules. It is obvious that there are many trains of thought to follow for this topic.

The implementation of HALS as described in Chapter 5 made it necessary to create processors which can be hierarchical. The algorithm enables the creation of multiple hierarchical processors as long as they are not executed after any non-topological processor, i. e., a processor which changes exact coordinates. Examples could be processors which change the layer assignment of nodes for example with the goal of reducing whitespace across the complete graph, or perhaps for certain aspect ratios for parent node or root graph dimensions.

Acronyms

ELK	Eclipse Layout Kernel
ELK Layered	Eclipse Layout Kernel Layered
O-NO-graph	Obviously Non-Optimal Graph
BIT	Binary Indexed Tree
HALS	Hierarchy-Aware Layer Sweep
BU	Bottom-Up
SCG	Sequentially Constructive Graph

Detailed Contents

List of Figures

Bibliography

[BBBH10] Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Ferdinand Hübner. A global k-level crossing reduction algorithm. In Md. Saidur Rahman and Satoshi Fujita, editors, *WALCOM: Algorithms and Computation*, volume 5942 of *Lecture Notes in Computer Science*, pages 70–81. Springer Berlin Heidelberg, 2010.

[BJM02] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 331–360. Springer, 2002.

[Cat95] Tiziana Catarci. The assignment heuristic for crossing reduction. *Systems, Man and Cybernetics, IEEE Transactions on*, 25(3):515–521, 1995.

[Dre95] Stefan Dresbach. A new heuristic layout algorithm for dags. In *Operations Research Proceedings 1994*, pages 121–126. Springer, 1995.

[EW86] Peter Eades and Nicholas C. Wormald. The median heuristic for drawing 2-layered networks. Technical Report 69, University of Queensland, Department of Computer Science, 1986.

[EW94] Peter Eades and Nicholas C Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[Fen94] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[Fuh12] Insa Fuhrmann. Layout of compound graphs. Diploma thesis, Kiel University, Department of Computer Science, February 2012.

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. Dag—a program that draws directed graphs. *Software: Practice and Experience*, 18(11):1047–1062, 1988.

[Hen92] Tyson Rombauer Henry. Interactive graph layout: The exploration of large graphs, 1992.

[HS04] Hongmei He and Ondrej Sýkora. New circular drawing algorithms. In *Proceedings of the Workshop on Information Technologies - Applications and Theory (ITAT)*, Slovakia, 2004.

[JM97] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:1–25, 1997.

[LM99] M. Laguna and R. Martí. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

[Mar98] Rafael Martí. A tabu search algorithm for the bipartite drawing problem. *European Journal of Operational Research*, 106(2–3):558 – 569, 1998.

[ML03] Rafael Martì and Manuel Laguna. Heuristics and meta-heuristics for 2-layer straight line crossing minimization. *Discrete Applied Mathematics*, 127(3):665 – 678, 2003.

[MSM99] Christian Matuszewski, Robby Schönfeld, and Paul Molitor. Using sifting for *k*-layer straightline crossing minimization. In *Proceedings of the 7th International Symposium on Graph Drawing (GD'99)*, volume 1731 of *LNCS*, pages 217–224. Springer, 1999. doi:10.1007/3-540-46648-7.

[Mä90] Erkki Mäkinen. Experiments on drawing 2-level hierarchical graphs. *International Journal of Computer Mathematics*, 37(3-4):129–135, 1990.

[NRL08] Lev Nachmanson, George Robertson, and Bongshin Lee. Drawing graphs with GLEE. In Seok-Hee Hong, Takao Nishizeki, and Wu Quan, editors, *Graph Drawing*, volume 4875 of *LNCS*, pages 389–394. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-77537-9_38.

[PT90] F.N. Paulisch and W.F. Tichy. EDGE: An extendible graph editor. *Software: Practice and Experience*, 20(S1):S63–S88, 1990.

[Rai05] Marcus Raitner. Visual navigation of compound graphs. In János Pach, editor, *Graph Drawing*, volume 3383 of *LNCS*, pages 403–413. Springer Berlin / Heidelberg, 2005. doi:10.1007/978-3-540-31843-9.

[San94] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, October 1994.

[San96] Georg Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, June 1996.

[Sch11] Christoph Daniel Schulze. Optimizing automatic layout for data flow diagrams. Diploma thesis, Kiel University, Department of Computer Science, July 2011.

[Sch15] Alan Schelten. On the greedy reduction of edge crossings. Bachelor thesis, Kiel University, Department of Computer Science, March 2015. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/alan-bt.pdf.

[SFvHM10] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In David Eppstein and EmdenR. Gansner, editors, *Graph Drawing*, volume 5849

of *Lecture Notes in Computer Science*, pages 135–146. Springer Berlin Heidelberg, 2010.

[SM91] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–892, Jul/Aug 1991. doi:10.1109/21.108304.

[SSRvH14] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. Counting crossings for layered hypergraphs. In Tim Dwyer, Helen Purchase, and Aidan Delaney, editors, *Diagrammatic Representation and Inference*, volume 8578 of *Lecture Notes in Computer Science*, pages 9–15. Springer Berlin Heidelberg, 2014.

[STT81] 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.

[vHDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM. Long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274.

[VML96] Vicente Valls, Rafael Martí, and Pilar Lino. A tabu thresholding algorithm for arc crossing minimization in bipartite graphs. *Annals of Operations Research*, 63(2):233–251, 1996.

[Wad01] Vance Waddle. Graph layout for displaying data structures. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of *LNCS*, pages 98–103. Springer, 2001.

Acknowledgments

I would like to thank my thesis advisor Ulf Rüegg for his time, patience and for his numerous smart ideas and suggestions. Further thanks go to Christoph Daniel Schulze for always being the next in line for me to fire my various nasty questions at and for the original idea for counting in-layer crossings. I would also like to thank Prof. Dr. Reinhard von Hanxleden for his assistance and encouragement. Furthermore I would like to thank Dr. Florin Manea for his help on my one and only proof.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,