# Graph Layout of Connected Components

Michael Cyruk

Master's Thesis
2017

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

Advised by
M. Sc. Ulf Rüegg

**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,

# Abstract

Graphical models are a useful tool for communicating and understanding abstract concepts. Many of them use graph-like notations. Even though reading these models might come quite naturally to humans, creating them can be tedious work with a bulk of the time spent adjusting the layout of the model instead of its semantics. Automatic graph drawing and adjustment algorithms try to mitigate this downside to graphical models.

Most automatic layout approaches only work on connected graphs where all elements, called nodes, are (at least transitively) connected to each other via an edge. They can work on an unconnected graph by processing each connected component separately. A second algorithm then has to lay out the connected components.

This thesis presents such an algorithm called Disconnected Components Compactor (DisCo) which is implemented as a part of the Eclipse Layout Kernel (ELK). Instead of approximating connected components with their rectangular bounding boxes a low resolution representation of the components is used to achieve drawings of minimum area. The approach is designed for laying out simple graphs at first and then extended to work on hierarchical graphs, i.e. graphs whose nodes can contain other graphs.

The resulting drawings have been tested against two other approaches already implemented in ELK with results indicating an advantage of the new implementation in terms of minimizing the total area of the layout, adhering to a specified aspect ratio and scaling a drawing to fit on a canvas of fixed proportions.

# Contents

Chapter 1

# Introduction

People use graphical notations to model, communicate, and understand abstract concepts. A lot of these notations can be represented by a class of mathematical structures known as *graphs*. Graph-like notations are used in a plethora of different domains, such as software engineering (e.g. Unified Modeling Language (UML)[1]) or the modeling of reactive systems (e.g. Sequentially Constructive Statecharts (SCCharts)[2], Ptolemy[3]), just to name a few examples. Even though the fields of application for these diagrams differ, they share some common features, notably two-dimensional shapes connected by lines. Hereafter, the shapes will be called *nodes* and the lines *edges*. See Figure 1.1 for comparison.

One of the problems with these graphical approaches lies in the construction of the models: drawing them by hand using an editor based on drag and drop can force users to spend most of their time on formatting the graphical representation instead of focusing on the semantics of their model. This is where algorithms for laying out graphs automatically come in handy. The Eclipse Layout Kernel (ELK)[4] features many different algorithms and options for this purpose.

Many established automatic layout algorithms assume a connected graph, i.e. each node is connected to another node by a sequence of incident edges. If this is not the case, one can partition the graph into more than one connected subgraph, and each of these graphs is called a *connected component* of the original one. ELK supports many of these algorithms by applying them to each connected component of a graph instead of working on the whole structure. This makes a post-processing step necessary to place the components relative to each other without permitting them to overlap. One goal of this procedure is to achieve the most compact layout having the smallest overall area.

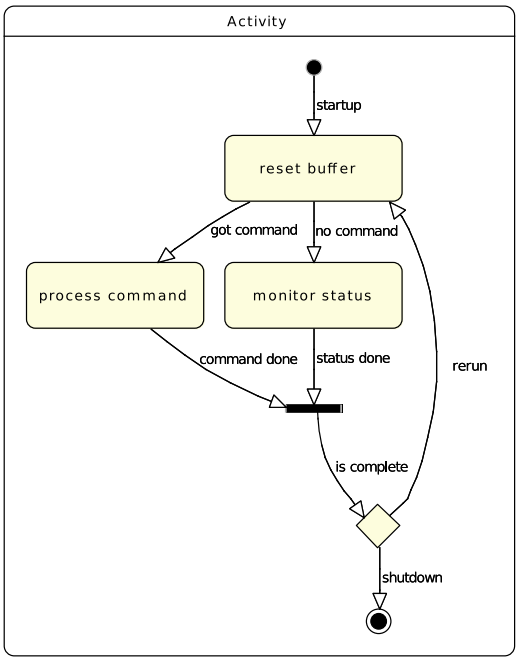
The approach currently implemented in ELK and described in Algorithm 1 is similar to *strip packing*, a technique Section 3.2 describes in more detail. Each connected component gets approximated by a rectangular bounding box. After that the components are presorted according to a configurable priority, e.g. by decreasing number of nodes per component as an estimation of its size or even a constant priority. Components of equal

[1] `http://www.uml.org/` (visited 2017-04-23)

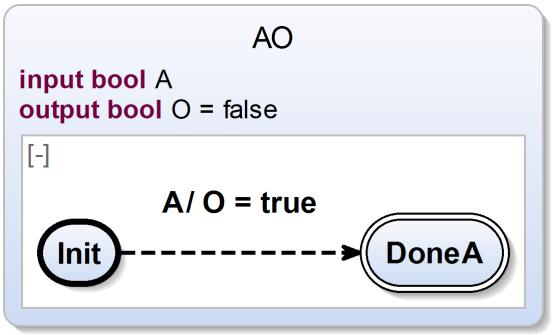
[2] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/SCCharts` (visited 2017-04-23)

[3] `http://ptolemy.eecs.berkeley.edu/` (visited 2017-04-23)

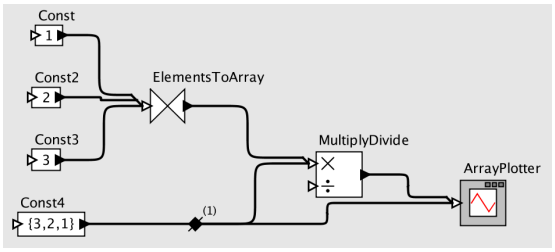
[4] `https://www.eclipse.org/elk/` (visited 2016-10-31)




**(a)** An UML activity diagram taken from Fuhrmann et al. [FSM+10]




**(b)** A SCChart taken from Rybicki et al. [RSM+16]




**(c)** A Ptolemy diagram taken from Spönemann et al. [SFH09]

**Figure 1.1.** Three different applications for graph-like notations looking similar despite semantic differences: nodes can be drawn rectangular (a, c), oval (b) or any other shape (a, c), whereas edges are represented by (dashed) lines consisting of straight segments (b, c) or curves (a).

priority are sorted by decreasing size of the area of their bounding boxes. Finally, the components are placed on a canvas according to this sorting row by row on a strip of indefinite height. The width of the strip is determined by the maximum of the width of the broadest bounding box of a component and the square root of the sum of the areas of all bounding boxes to achieve a square looking aspect ratio of 1 for the final layout. Moreover, the algorithm accepts a desired *aspect ratio* as a parameter to adjust the width of the strip even further. This is done by multiplication with the strip width, as the aspect ratio is simply defined as the width of the desired drawing by its height. The amount of empty space between each bounding box can be configured, too. Figure 1.2 shows an example layout.

This simple layout algorithm for packing connected components onto a canvas is obviously non-optimal regarding the minimization of the overall area of the drawing. The approximation of the shape of the components by rectangular bounding boxes is easy to implement but rather imprecise, for instance. This thesis will present a different approach which relies on other structures than bounding boxes to represent a component to better approximate their shape, which is a first step to achieve drawings with a smaller area.

**Algorithm 1:** Simple Row Placement for Components

**input :** $n$ components $C_i$ with width $w_i$, height $h_i$, priority $p_i$ , $1 \leqslant i \leqslant n$
spacing $s$ between each component
desired aspect ratio $a_{ratio}$ of the drawing

sort $C_i$, $1 \leqslant i \leqslant n$ by decreasing priority $p_i$. If there are components of equal priority, order them by decreasing size of the area of their bounding boxes $w_i * h_i$.
/* determine the maximum width for all rows */
// set width to the maximum width among the components
$m_{row} \leftarrow max(\{w_i : 1 \leqslant i \leqslant n\})$
// or use the square root of the total area of all components if greater
$m_{row} \leftarrow max(m_{row}, \sqrt{\sum_{i=1}^{n} w_i * h_i})$
// adhere to the given aspect ratio
$m_{row} \leftarrow m_{row} \cdot a_{ratio}$
/* place components iteratively into rows */
$x \leftarrow 0$ // current x-position on the canvas
$y \leftarrow 0$ // current y-position on the canvas
$h_{box} \leftarrow 0$ // height of the tallest component in current row
**foreach** component $C_i$ **do**
    **if** $x + w_i > m_{row}$ **then**
        // switch to next row
        $x \leftarrow 0$
        $y \leftarrow y + h_{box} + s$
        $h_{box} \leftarrow 0$
    place $C_i$ on the canvas with its top left corner at $(x, y)$
    $h_{box} \leftarrow max(h_{box}, h_i)$
    $x \leftarrow x + w_i + s$

ELK will serve as the platform for its implementation.

A second thing most algorithms for laying out graphs assume is that nodes are simply points without any dimensions. In the best case they treat nodes as rectangles with two dimensions, but do not use the area inside of them for anything but maybe putting a text label or a picture inside of it. ELK provides further functionality: any node can contain other graphs, as seen in Figure 1.3. This feature provides a different kind of relationship between nodes, which can be used semantically by graphical notations employing this kind of graph: instead of only relating nodes to each other by connecting them with edges, the containment of nodes by other nodes implies a hierarchy among them. This is why this kind of graph is called a *hierarchical graph*.

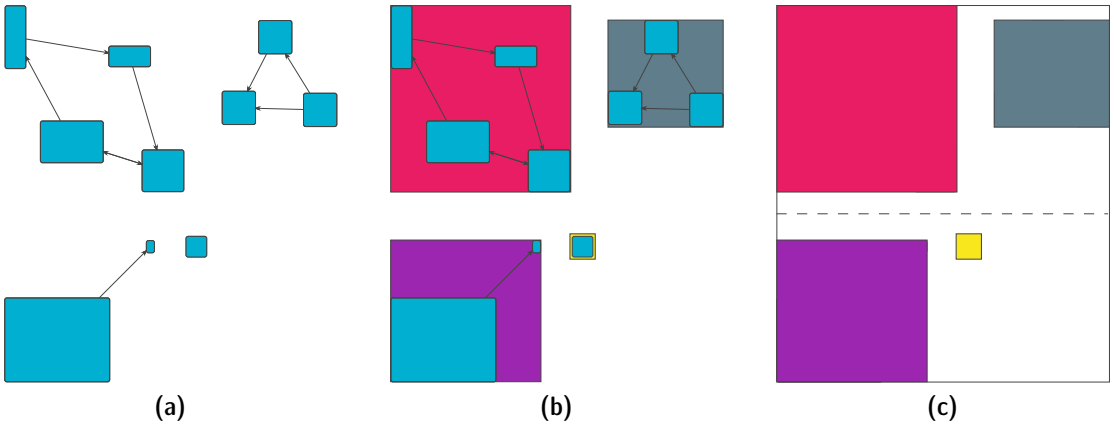



**Figure 1.2.** Examination of a small example layout: (a) example of a graph consisting of four connected components, laid out by ELK with the current approach for packing connected components. The origin of the coordinate system is in the upper left corner like on a computer display. (b) The same layout is shown depicting the surrounding bounding boxes of each component. (c) The components have been sorted by decreasing number of nodes within each component. They occupy two rows, so that they adhere to a desired aspect ratio of 1.0 (a square). The top row has been laid out first until the computed maximum width for a row has been reached. Then the algorithm switched to the bottom row to place the two remaining components.

As ELK supports hierarchical graphs and the layout algorithm for placing connected components presented in this work is implemented in ELK, it makes sense to keep the added functionality of hierarchical graphs in mind when designing the algorithm. Therefore this thesis will not only offer a packing algorithm for components to achieve drawings with a small overall area, but also an extension of this algorithm to support hierarchical graphs. Especially the support for edges which connect between nodes on differing levels of hierarchy emerges as a non-trivial problem in this context.

## 1.1 Contributions

To tackle the problems stated above, this work examines an alternative approach to compaction of components not using bounding boxes to represent components but low resolution approximations of them called *polyominoes*. This lead to the development of Disconnected Components Compactor (DisCo) as part of ELK.

This thesis explains the ideas and conception of the newly developed approach. First, the use case of laying out *simple graphs* is examined which closely resemble the graph theoretic notion of a graph. Second, the approach is extended and heavily modified to

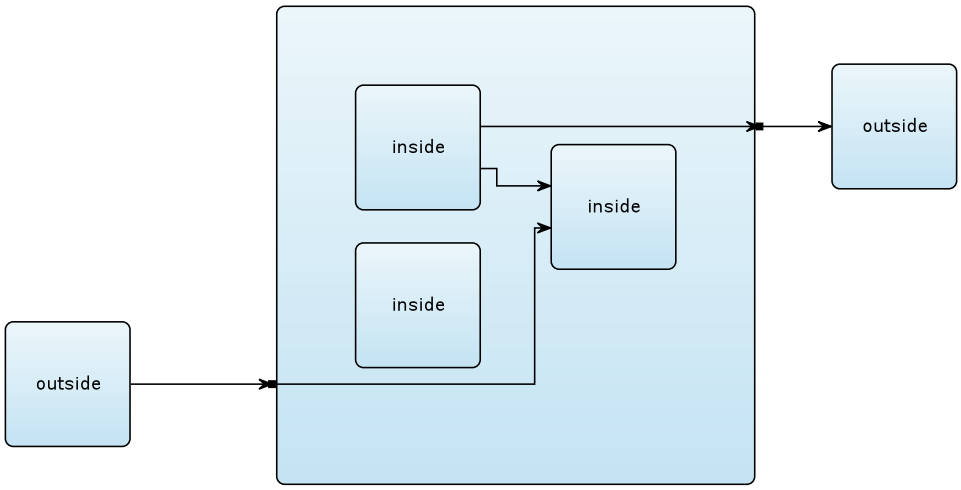



**Figure 1.3.** An example of graph drawn with ELK. Any node can contain another graph. Moreover, edges may connect these inner graphs to their surrounding node or even ones outside of it.

lay out *hierarchical graphs* in which every node can contain another graph, forming a hierarchy of arbitrary depth.

Further contributions are the quantitative evaluation concerning the quality of drawings DisCo renders in these two use cases and finally some suggestions for improving DisCo in the future.

## 1.2 Outline

The rest of this thesis is organized as follows: Chapter 2 establishes some basic definitions of the technical terms used in this work and some relevant information about ELK. Chapter 3 gives an overview of related work in the field of automatic graph drawing. Chapter 4 explains the concepts and features of DisCo concluding with some information on its integration into ELK. Chapter 5 evaluates the implemented algorithm with respect to a sample of aesthetic criteria. Chapter 6 concludes thesis with a summary and proposals for future research.

Chapter 2

# Preliminaries

This chapter introduces key terms used throughout this thesis and gives a short introduction to ELK.

## 2.1 Terminology

To comprehend the rest of this thesis, this section defines the kind of graphs this work is based on. The most basic graph theoretical definitions are taken from Di Battista et al. [DET+99]. Additional terminology is based on the ELK documentation[1] if not stated otherwise.

**2.1.1 Definition** (graph, node, edge). A *graph* $G$ is a pair $G = (V, E)$. $V = \{v_1, \ldots, v_n\}$ is a set of *nodes* and $E = \{e_1, \ldots, e_m\}$ is a subset of $\{e \in \mathcal{P}(V) \colon |e| = 2\}$ representing *edges* connecting nodes of the graph. Each edge $e = \{u, v\} \in E$ is an unordered set, so there is no sense of direction.

**2.1.2 Definition** (digraph, directed edge, source, target). A *directed graph* or *digraph* $G = (V, E)$ differs from an undirected graph in its edge set $E$: in this case, each edge $e = (u, v) \in E \subseteq V \times V$ is *directed* from $u$ to $v$. Node $u$ is called the *source* of $e$, $v$ its *target*.

**2.1.3 Definition** (subgraph, induced subgraph). Let $G = (V, E)$ be a (directed) graph. A (directed) *subgraph* of graph $G$ is a (directed) graph $G' = (V', E')$, such that $V' \subseteq V$ and $E' \subseteq E \cap \mathcal{P}(V')$ ($E' \subseteq E \cap V' \times V'$). If $E' = E \cap \mathcal{P}(V')$ ($E' = E \cap V' \times V'$), then $G'$ is *induced* by $V'$.

**2.1.4 Definition** (connected graph, maximal connected subgraph, connected component). Let $G = (V, E)$ be a graph. $G$ is called *connected* if there is an undirected path between each two of the vertices of $G$. A subgraph $G'$ of $G$ is called *maximal connected* if $G'$ is connected and no other node from $G$ with its respective adjacent edges can be added to $G'$ without the resulting graph becoming unconnected. A maximal connected induced subgraph of $G$ is called a *connected component* of $G$.

[1] `http://www.eclipse.org/elk/documentation/tooldevelopers/graphdatastructure.html` (visited 2017-02-28)

Beyond these classical definitions one can add further dimensionality to graphs. Rather than just associating nodes with points like in a graph theoretical setting, rectangles are going to represent nodes.

**2.1.5 Definition** (dimensions of nodes). A node $v \in V$ has an associated *width* and *height*: $(w_v, h_v) \in \mathbb{R} \times \mathbb{R}$

Now that a node is not represented by points anymore, edges can start and end at different points of the four different sides of the node. To model these end points, ports are introduced to the graph.

**2.1.6 Definition** (simple graph, port, port mapping). A *simple graph* is a tuple $G = (V, P, \pi, E)$ consisting of

- nodes $V = \{v_1, \ldots, v_n\}$, with dimensions $(w_{v_i}, h_{v_i}) \in \mathbb{R} \times \mathbb{R}$ for all $1 \leqslant i \leqslant n$,
- a set of *ports* $P$, each of them belonging to a side of a node $v \in V$,
- a *port mapping* function $\pi : P \to V \times D$ assigning each port to a node $v$ and one of the four cardinal directions $D = \{n, s, e, w\}$ (short for *north*, *south*, *east*, and *west*) depending on the side of $v$ they are placed at,
- and a set of directed edges $E \subseteq P \times P$ between the ports in $P$.

An interesting feature of nodes having nonzero dimensions is the opportunity to create a hierarchy by nesting graphs, i.e. putting graphs into nodes. For that purpose, one requires a structure for describing the inclusion of nodes by another higher level node. Sugiyama and Misue propose a tree for this [SM91].

**2.1.7 Definition** (inclusion tree, parent, child). An *inclusion tree* $T = (V, F, r)$ is a digraph, where an edge $(u, v) \in F \subseteq (V \cup \{r\}) \times V$ indicates that $u$ includes $v$. $T$ must be a tree (an acyclic, connected graph). $r \notin V$ is an artificial root node, connecting all subtrees with edge sets $\subseteq V \times V$.
$Pa(v)$ denotes a set containing the *parent* $u$ of $v$ in the inclusion tree; it is empty if $v$ is the root of the graph. $Ch(u)$ denotes the set of *children* of $u$, i.e. $v \in Ch(u)$ iff $(u, v) \in F$.

Note that $Ch(u)$ does not include all descendants of $u$ but only its direct children. If $(u, v), (v, w) \in F$, then $v \in Ch(u)$, but $w \notin Ch(u)$.
The following structure, also adapted from Sugiyama and Misue, is usually known as a *compound graph*, but is called a *hierarchical graph* in this thesis in accordance with ELK terminology. However, the term hierarchical graph has been used before by Lengauer [Len90] and should not be confused with the following definition.

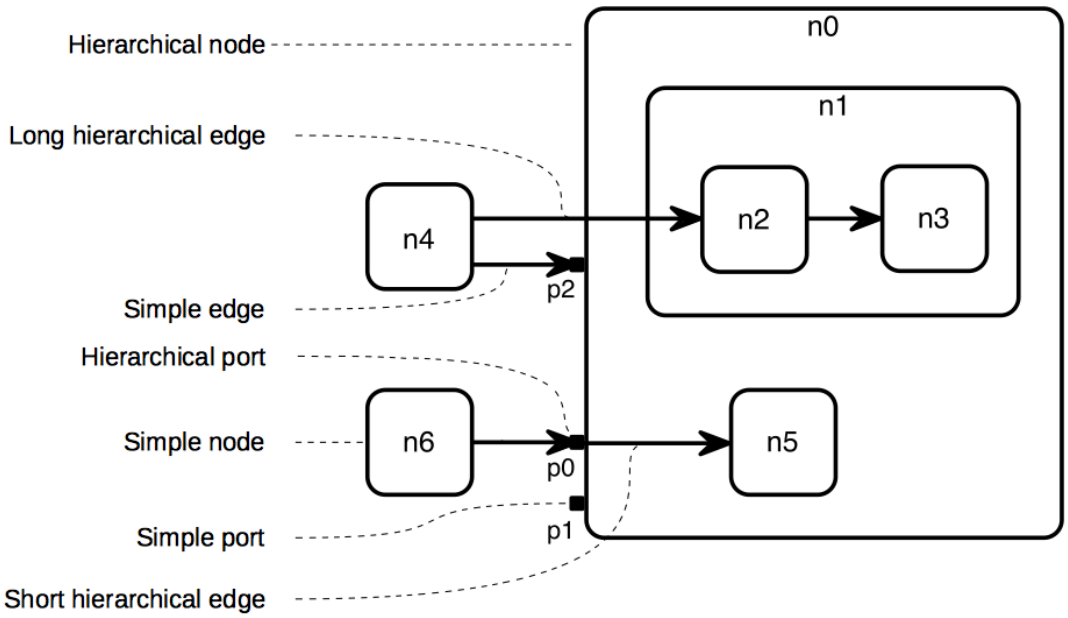



**Figure 2.1.** Visualization of a hierarchical graph with key terms according to the ELK documentation. Ports are shown as little black squares at the edges of nodes. All ports shown in this picture are pointed to direction $w$ (west). The node $n_1$ is a child of the node $n_0$, but $n_2$ is not. The long hierarchical edge between $n_4$ and $n_2$ can be split up into three short hierarchical edges between $n_4$ and $n_0$, $n_0$ and $n_1$, and finally $n_1$ and $n_2$ with some help of two newly introduced ports at the boundaries of $n_0$ and $n_1$.

**2.1.8 Definition** (hierarchical graph). A *hierarchical graph* is a tuple $G = (V, P, \pi, E, F, r)$. It is a simple graph extended by an artificial root node $r$ and an edge set $F \subseteq (V \cup \{r\}) \times V$ representing the edges of an inclusion tree.

The artificial root node has been introduced to model the hierarchy relation in a single inclusion tree. Otherwise, there would be a tree for each connected component of the hierarchical graph.

ELK builds upon this notion of a hierarchical graph and uses some additional terms for describing some of the features of the graph, see Figure 2.1 and Figure 2.2.

**2.1.9 Definition** (simple node, hierarchical node). There are two different kinds of nodes: a *simple node* $v$ does not have any children, i.e. $Ch(v) = \varnothing$. A *hierarchical node* is any node that is not simple and thus contains children.

**2.1.10 Definition** (simple edge, hierarchical edge, short hierarchical edge, long hierarchical edge). Let $G = (V, P, \pi, E, F, r)$ be a hierarchical graph, $u, v \in V$ two nodes and $p, q \in P$ two ports with $\pi(p) = (u, d)$, and $\pi(q) = (v, d')$ for some directions $d$ and $d'$.
A *simple edge* of $G$ is an edge that connects two ports of nodes with the same parent, i.e.

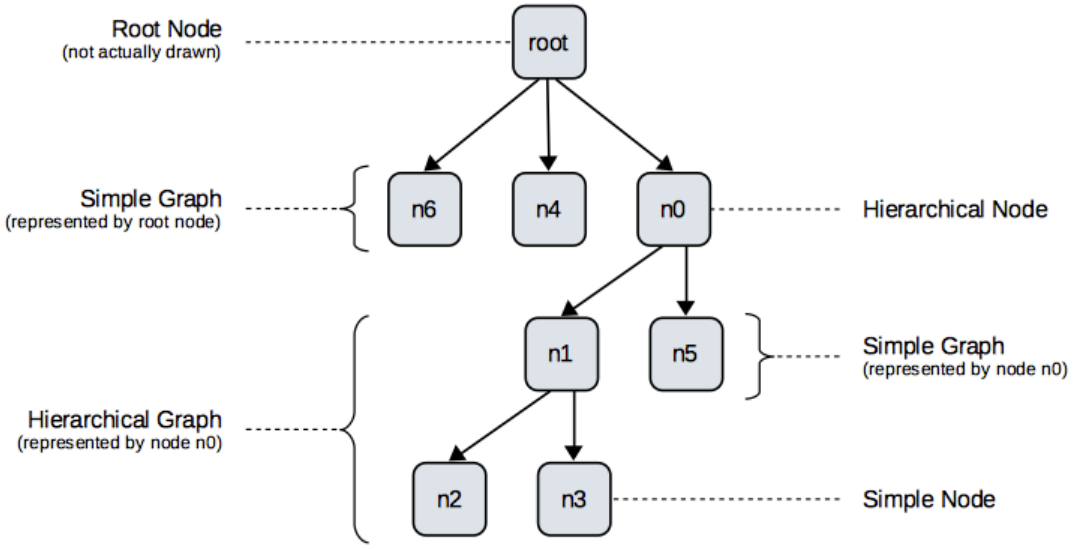



**Figure 2.2.** An inclusion tree of a hierarchical graph according to the ELK documentation.

the following equivalence applies: $(p, q)$ is a simple edge iff $Pa(u) = Pa(v)$.
In contrast, a *hierarchical edge* is any edge in $E$ that is not simple. There are two different kinds of hierarchical edges:

- a *short hierarchical edge* only leaves or enters one hierarchical node to reach its target, i.e. $(p, q)$ is a short hierarchical edge iff $Pa(v) = u$ or $Pa(u) = v$,
- a *long hierarchical edge* is a hierarchical edge that is not short.

**2.1.11 Definition** (simple port, hierarchical port). Let $G = (V, P, \pi, E, F, r)$ be a hierarchical graph, $p \in P$ a port, $v \in V$ a node and $\pi(q) = (v, d)$ for some direction $d$.
$p$ is called a *simple port* either if $v$ is a simple node or if $p$ has no incident hierarchical edges, meaning there are no hierarchical edges with $p$ as a source or a target port. A *hierarchical port* is a port that is not simple.

The definition of the hierarchical graph given above differs from the actual ELK documentation in one way: as seen in Figure 2.1, ELK allows edges to start and end in nodes directly rather than connecting to a port. Definition 2.1.8 was chosen for the hierarchical graph for sake of formal simplicity, as one can simply add invisible pseudo-ports whenever they are missing in the figure, so that it adheres to the given definition. Furthermore, this thesis does not cover the handling of long hierarchical edges explicitly, as they can simply be converted to multiple short hierarchical edges if one adds a pseudo-port each time the original long hierarchical edge crosses the boundary of a hierarchical node.

The last definitions do not relate to graphs themselves but to drawings of them.

**2.1.12 Definition** (width, height, aspect ratio). A drawing of a graph can be represented by a bounding box $B = (w, h) \in \mathbb{R} \times \mathbb{R}$. The first element $w$ is called the *width* of the graph, whereas $h$ is its *height*. The *aspect ratio* of the drawing is defined as the relation

$$a_{ratio} = \frac{w}{h}.$$

It follows that drawings with equal width and height result in an aspect ratio of 1, whereas drawings with $w > h$ have an aspect ratio greater than 1. Drawings with $w < h$ have an aspect ratio smaller than 1 but greater than 0.

Drawings of graphs are subject to a reference frame they are depicted on, e.g. a computer screen or a piece of paper. Rüegg et al. [RAC+17] define a measure facilitating quantitative statements about drawings with respect to the reference frames they are depicted on.

**2.1.13 Definition** (max scale value). Let $R_{w_r,h_r}$ describe a reference frame with width $w_r \in \mathbb{R}$ and height $h_r \in \mathbb{R}$. Let $B = (w, h) \in \mathbb{R} \times \mathbb{R}$ be the bounding box of a drawing of a graph. The *max scale value* $s$ of the drawing with respect to $R_{w_r,h_r}$ is defined as

$$s = min \left\{ \frac{w_r}{w}, \frac{h_r}{h} \right\}.$$

The max scale value denotes a scaling factor by which the drawing has to be scaled to fit closely into the given reference frame. As this measurement depends on the size of the reference frame it is best used to compare two drawings with respect to a common frame.

**2.1.14 Definition** (max scale ratio). Given two drawings with two max scale values $s_1$ and $s_2$ in relation to a common reference frame $R$, the max scale ratio $r = \frac{s_1}{s_2}$ states which drawing can be drawn with a larger scale factor within $R$. The first drawing can be depicted larger than the second if $r > 1$.

## 2.2 The Eclipse Layout Kernel (ELK)

The implementation of the packing algorithm for connected components introduced in this work is based on a framework called the Eclipse Layout Kernel (ELK). Figure 2.3 illustrates its role as an intermediary, as it provides an infrastructure to connect diagram editors and diagram viewers to automatic layout algorithms. The most important data structure provided by the framework is the *ElkGraph*. It is the graph representation developers of diagram editors and viewers have to transform their own models into to be

[2] `https://www.eclipse.org/elk/documentation.html` (visited 2017-05-03)

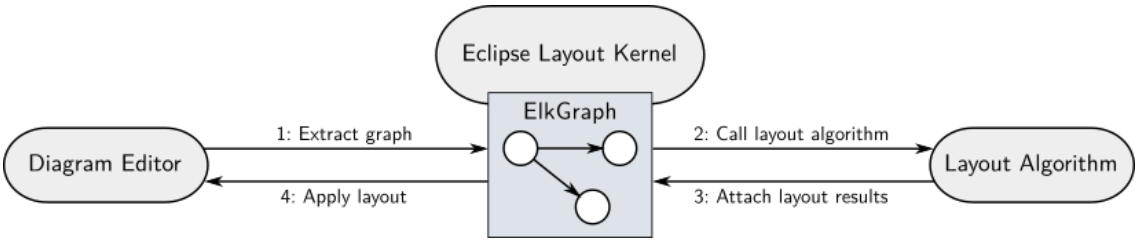


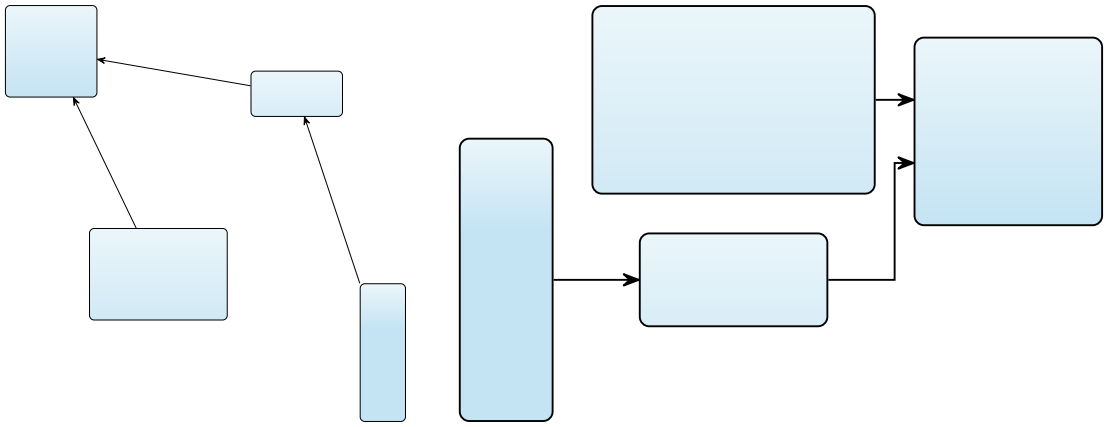

**Figure 2.3.** Overview of ELK taken from the ELK documentation[2]

able to use ELK for automatic layout. Therefore, it serves as a layer of abstraction for engineers of layout algorithms. The ElkGraph represents basically a hierarchical graph as described in Definition 2.1.8 and illustrated in Figure 2.1. The development of the layout algorithm DisCo as a basis for this thesis started close to the end of a transitional period: ELK was formerly part of a different project, Kiel Integrated Environment for Layout Eclipse RichClient (KIELER), and was migrated into a standalone framework. This is why DisCo still works on the predecessor of the ElkGraph, called the *KGraph*. The KGraph is still an implementation of a hierarchical graph and previously held the same intermediate position between diagram editors and layout algorithms, though. Therefore, DisCo can be conceptually treated as a layout algorithm within the current ELK framework for all intents and purposes of this thesis.

ELK does not only provide the ElkGraph but also some layout algorithms by itself. As the layout algorithm presented here only lays out components, these come in handy to draw the actual nodes and edges within components beforehand. Two of these algorithms helped in illustrating this thesis, wherefore they are mentioned in this section: Eclipse Layout Kernel Force (ELK Force) and Eclipse Layout Kernel Layered (ELK Layered).

ELK Force implements a force-based approach to laying out graphs. Edges are interpreted as physical springs that are pulling the nodes they connect together, whereas nodes not connected by an edge repulse each other. This thesis already showed an example layout using ELK Force in Figure 1.2, see Figure 2.4a for a different example. The algorithm currently supports two physical models, one by Eades [Ead84] and the other by Fruchterman and Reingold [FR91].

Figure 2.4b depicts an example layout for ELK Layered. Sugiyama et al. introduced the layer-based method [STT81]. It works on directed acyclic graphs and emphasizes the reading of the resulting drawing in a specific order by routing as many edges as possible into the same direction. The nodes are placed in *layers*, sometimes called hierarchies, which should not be confused with Definition 2.1.8 of hierarchical graphs in this thesis. The nodes are also ordered within their layers to minimize the number of edge crossings in the final drawing. For instance, the example layout shows a graph whose edges are pointing to the right. One can discern three distinct layers and the middle one of them



**(a)** ELK Force **(b)** ELK Layered

**Figure 2.4.** The same graph laid out by two different algorithms

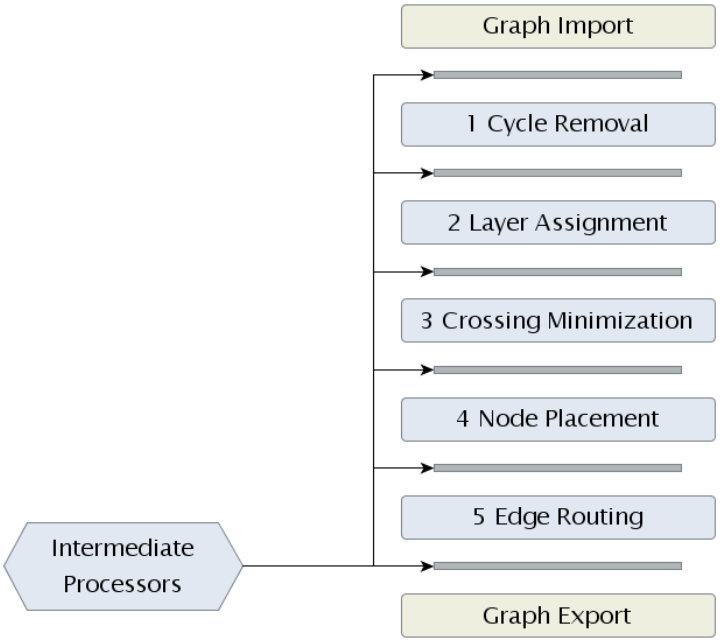



**Figure 2.5.** The five phases of the ELK Layered algorithm taken from the old KIELER documentation[3]

contains two nodes, whereas the other two layers consist of only one node.

One of the distinctive features of the implementation ELK Layered is its degree of modularity as demonstrated in Figure 2.5. First of all, the algorithm starts with a graph

[3] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KLay+Layered` (visited 2017-05-03)

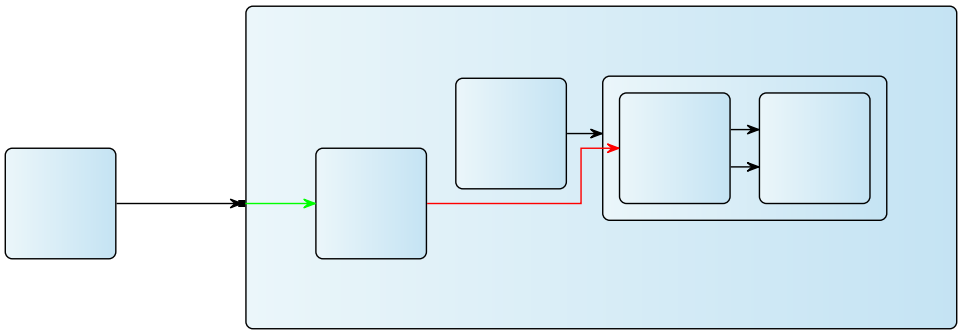


**Figure 2.6.** A hierarchical graph with three levels drawn in ELK Layered. The algorithm can process both short hierarchical edges (green) and long ones (red). The long hierarchical edges are handled as a sequence of short hierarchical edges internally.

import and ends with a graph export, as ELK Layered does not work on the ElkGraph itself but on a specialized structure, called the *LGraph*. In between these two steps the algorithms is divided into five phases. In between these phases so called *intermediate processors* can be invoked for pre- and post-processing of a phase. If a phase has more than one implementation, these are interchangeable without the need to adjust any other phase. Schulze et al. [SSH14] describe the phases in more detail. At this place, a short synopsis is sufficient:

*Cycle removal.* The layer-based approach only works on directed acyclic graphs. To process directed graphs with cycles, a subset of edges is determined by a heuristic such that the graph becomes free of directed cycles if these edges are reversed. The edges will be reversed back again in the final layout.

*Layer assignment.* Nodes are assigned to layers such that all edges point in the same direction.

*Crossing Minimization.* Nodes of each layer are ordered within them with the goal of minimizing edge crossings.

*Node Placement.* Adhering to the previously determined order the positions of the nodes are fixed in such a way that edges can be routed as straight as possible.

*Edge Routing.* Finally, bend points are added to the edges depending on the routing style, e.g. orthogonal line segments.

Finally, one of the most important features of ELK Layered is its capability to draw hierarchical graphs with support for short and long hierarchical edges. As explained in Schulze et al. [SSH14], this is a bottom-up process, recursively applying the layout

algorithm to all levels of the hierarchy, beginning with the innermost ones, i.e. the leaves of the inclusion tree. Figure 2.6 shows such a layout.

Chapter 3

# Related Work

The central problem addressed in this thesis covers how to efficiently pack connected components of a drawing of a hierarchical graph using polyominoes. Therefore, the solution is oriented towards four different areas of research: first, there is hierarchical layout. Second, different authors have taken on the work of improving compaction of connected components in general and third, also taken care of components of hierarchical graphs. Finally, some researchers proposed using polyominoes for packing components. This chapter serves as a quick overview.

## 3.1 Hierarchical Layout

The term *hierarchical layout* used in this thesis can be a little bit misleading, as it often refers to what is known as layer-based drawing in ELK, see Section 2.2. In accordance to the ELK terminology defined in Definition 2.1.8, a hierarchical graph is what Sugiyama and Misue [SM91] call a *compound digraph*. The authors give a formal definition of these graphs using an inclusion tree to model hierarchical nodes containing graphs themselves very similar to Definition 2.1.7. Furthermore, they present an algorithm for computing a layer-based drawing for any given hierarchical graph. However, their algorithm differs from the ELK approach of building each level starting at the leaves of the inclusion tree of the graph and working bottom-up to the root.

Their algorithm tries to improve upon some aesthetic criteria, among them, pursued *closeness* among nodes connected to each other, the prevention of *line crossings* of edges with each other and finally the reduction of *line-rect-crossings*, i.e. edges crossing the borders of hierarchical nodes.

## 3.2 Compaction of Connected Components

The compaction of connected components, isolated nodes, and rectangles in general has been intensively studied in the last few decades. This is why, this section only presents selected examples. A more comprehensive overview can be found elsewhere, e.g. Lengauer [Len90].

One approach pretty similar to the simple row graph placer from Algorithm 1 is *strip packing*. Baker et al. [BJR80] and Coffman et al. [JGJ+80] specify this method as a solution for the following two dimensional packing problem: given a strip of a certain fixed width and an semi-indefinite height, i.e. a strip with a border to the bottom but no bounds at the top, and rectangles at most as wide as the strip itself, pack the rectangles into the strip, so that no two rectangles overlap and the height to which the strip is filled is minimized. Furthermore, the rectangles are assumed to have a fixed orientation with one side running parallel to the bottom of the strip and are not allowed to be turned in any way. One example packing is given in Figure 3.1a. As this problem is known to be NP-complete, one has to work with heuristics to compute an efficient solution. Coffman et al. present two solutions based on the same principle. The rectangles are first sorted by non increasing height and are placed in that order on so called *levels*. The first level is simply the bottom of the strip, the second is marked by a horizontal line drawn on top of the rectangle with the biggest height placed on the first level, and so on. The rectangles are always place left-justified within their level. The authors then distinguish their two algorithms by the process for deciding on which level to put the next rectangle.

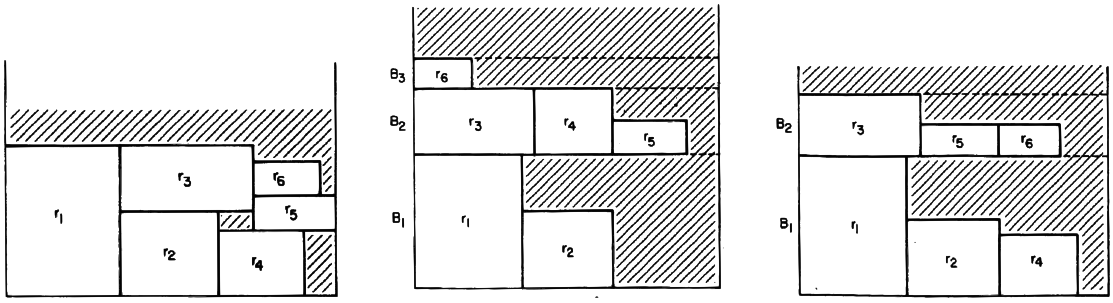
*Next-Fit-Decreasing-Height (NFDH)* Rectangles are placed on a level until the free horizontal space on the current level is not wide enough to fit the current rectangle to be positioned. Then a new level is introduced and all remaining rectangles are placed on that level until it runs out of space. Then a new level is introduced and so forth, see Figure 3.1b.

*First-Fit-Decreasing-Height (FFDH)* Rectangles are placed like in the previous approach but not only the newest level is a candidate level for the current rectangle. All levels are tested for sufficient place to store it beginning with the bottom level. Only if no level has enough horizontal space, a new level is introduced, see Figure 3.1c.

Another paper by Coffman and Shor [CS93] introduces a third heuristic, *Best-Fit-Decreasing-Hight (BFDH)*, similar to FFDH. In this case rectangles are not placed on the lowest level they fit, but rather on the level they fit best, i.e. where the least free horizontal space is left after inserting the rectangle.

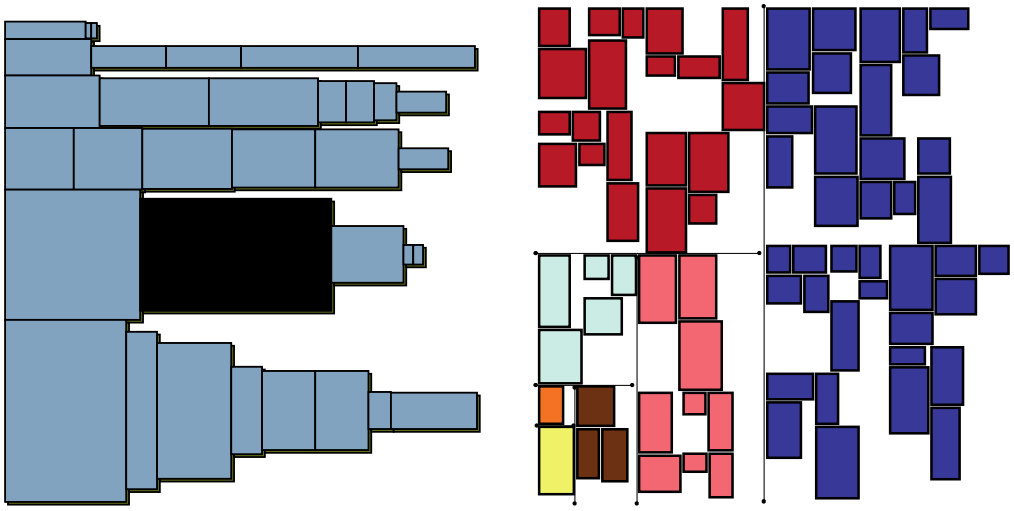
Freivalds et al. [FDK02] and Dogrusoz [Dog02] introduce two further approaches for packing components. The first one is called *tiling* and is variant of BFDH but this time the strip does not have a fixed width. Instead the algorithm starts placing components on the lowest level of a narrow strip. If the current component cannot be placed on any of the already existing levels the algorithm either creates a new level or increases the width of the whole strip. The decision depends on a desired aspect ratio of the final layout given as a parameter. An example layout is given in Figure 3.2a

The second approach is a divide and conquer method called *alternate bisection*. The set of components is first bipartioned recursively using a metric like the total area of




**(a)** A two-dimensional packing **(b)** NFDH **(c)** FFDH

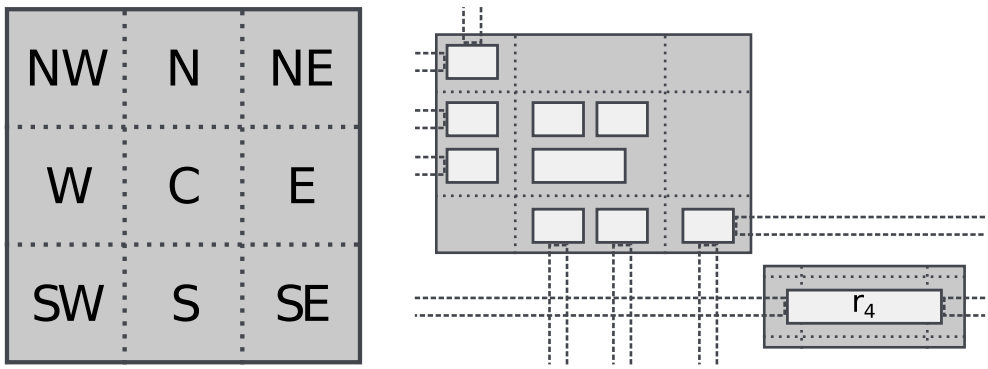
**Figure 3.1.** Some strip packing examples from Coffman et al. [JGJ+80] for compacting six rectangles $r_i$ with $1 \leqslant i \leqslant 6$: (a) This depicts a packing of optimal height. The next two pictures show two heuristic approaches to filling a strip using levels $B_j$ with $j \in \mathbb{N}$. The NFDH algorithm (b) inspects only the most recent level to place the current rectangle, while the FFDH algorithm (c) also checks for gaps in discarded levels.



**(a)** Tiling **(b)** Alternate bisection

**Figure 3.2.** Illustrations of a tiling and an alternate bisection layout from Freivalds et al. [FDK02]

components. The recursion stops when a partition consists of a number of components that is easily laid out, e.g. 1. The partial solutions are combined by putting them side by side alternating between a horizontal and a vertical orientation, see Figure 3.2b.




**(a)** Container with nine cells **(b)** An example placement using cell packing

**Figure 3.3.** A container used in cell packing and an example placement taken from Rüegg et al. [RSG+16a]. Components are drawn as light gray rectangles, external extensions are beams marked by dashed borders attached to the components. The example placement uses two containers, one containing a single component $r_4$.

## 3.3 Compaction of Components in Hierarchical Graphs

The amount of research of packing components of simple graphs by far outweighs the work published concerning the compaction of components in hierarchical graphs. The biggest difference from the simple graph case is the additional complexity added by hierarchical edges connecting to a surrounding parent node. From the perspective of the simple graph contained within it, these edges span across the whole drawing area. As they are allowed to intersect with other edges but must not do so with nodes, they have to be considered while laying out components.

Rüegg et al. [RSG+16b; RSG+16a] model hierarchical edges as so called *external extensions*, which are indefinite beams connecting to a component and stretching out into one of the four cardinal directions. Details are explained in Section 4.1.2 as external extensions play a major role in the DisCo algorithm presented in this thesis.

The components with external extensions attached to them are subsequently laid out by an algorithm called *cell packing*. The algorithm places the components into containers divided into nine *cells*, with rules concerning which cells a component is allowed to be placed into, depending on the orientation of the external extensions present. If a container cannot accommodate a specific component a new container is added to the layout, see Figure 3.3b.

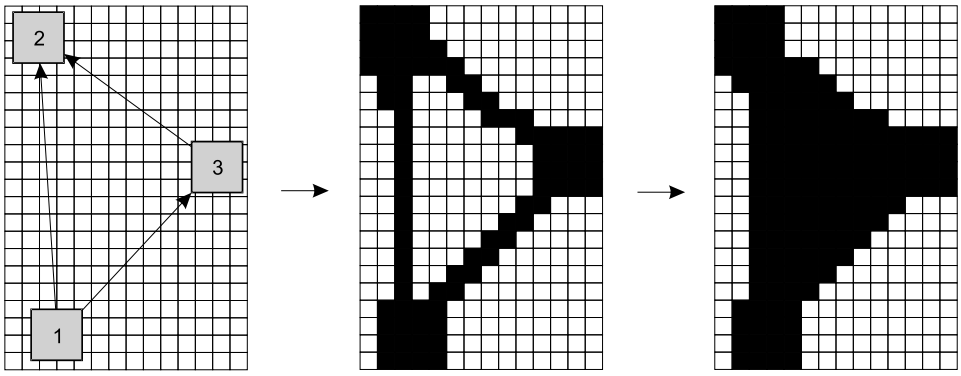



**Figure 3.4.** An example of how a component can be transformed into a low resolution polyomino taken from Goehlsdorf et al. [GKS07].

## 3.4 Polyominoes

All algorithms for compacting components presented in this chapter so far use rectangles as a way to represent a connected component. Freivalds et al. [FDK02] designed a different approach using polyominoes. A Polyomino is a geometric figure consisting of squares of equal size which are aligned on a Cartesian grid. The authors use these figures to better approximate the shape of a component. The polyominoes are than placed on an underlying grid by using a heuristic. Goehlsdorf et al. [GKS07] expand this approach with further suggestions for improvements. Section 4.1 explains all the details of the polyomino approach as it is the basis of the layout algorithm presented in this thesis. Figure 3.4 shows how a component can be translated into a polyomino.

A different use case for polyominoes concludes this chapter. All of the approaches and ideas presented so far, have been concerned with the *creation* of new layouts from scratch. However, in interactive applications users might want to draw diagrams themselves and after that improve their creation by hitting a button for applying an automatic layout. The goal of this so called *layout adjustment* is to preserve relative positions between graph elements while still improving the overall layout. Gansner et al. present a solution for this problem based on using polyominoes for approximating connected components [GHN13]. The actual layout algorithm using these polyominoes has nothing in common with the component placer of Freivalds et al. and the one presented in this thesis. The idea of the authors is based on the more general problem of *overlap removal* as presented by Gansner et al. [GH10] and Nachmanson et al. [NNB14] and uses a method related to force-based layout algorithm, such as ELK Force. Figure 3.5 shows an example.

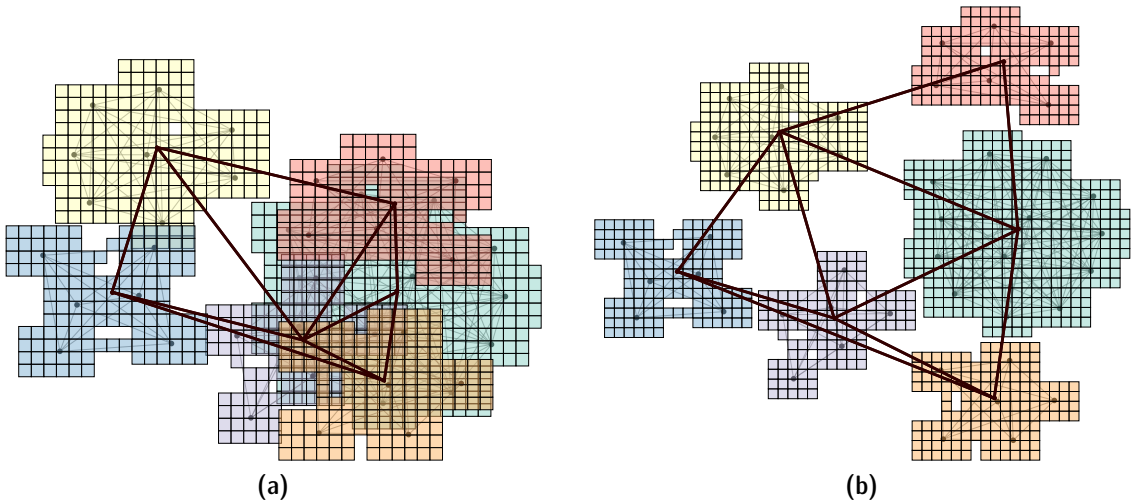



**Figure 3.5.** The approach of Gansner et al. uses polyominoes for the approximation of the shape of the different connected components of the graph [GHN13]. The actual algorithm for the placement is based on stress which is related to the force-based approach implemented in ELK Force. The picture on the left shows the initial placement with overlapping components. The right picture shows the final layout with overlaps removed while retaining the positions of the components relative to each other.

Chapter 4

# Compaction of Disconnected Components (DisCO)

This chapter explains the underlying mechanisms of compacting connected components with DisCo using polyominoes. It serves to present ideas and reasons for the design choices of the approach without going into details about its implementation in ELK. Nevertheless, the last section of this chapter presents some basic information about the place of DisCo in the greater ELK framework.

## 4.1 Polyominoes in Hierarchical Graphs

As already mentioned, DisCo uses polyominoes to compact connected components. The explanation of the approach is broken up into two sections. The basic idea is described using simple graphs first, before going into the details of how to extend the algorithm to hierarchical graphs.

### 4.1.1 The Polyomino Approach

Freivalds et al. have proposed an alternative approach to laying out connected components [FDK02]. In their packing algorithm each connected component is represented by a *polyomino* instead of its bounding rectangle. This type of shape consists of squares of the same size, each of them aligned in a discrete planar grid. An example is given in Figure 4.1, which shows that polyominoes approximating their component resemble low resolution pictures of the structure they are enclosing. The implementation in ELK is called Disconnected Components Compactor (DisCo).

Freivalds et al. describe polyominoes more formally [FDK02]:

**4.1.1 Definition** (Polyomino). A *polyomino* is finite set of $k \geqslant 1$ cells of an infinite planar square grid $G$ that are fully or partially covered by the drawing of the graph.

Unlike a bounding box, the polyomino representation of a connected component is not unique by any means. The biggest challenge to achieve a reasonable polyomino is to determine a good value for the width and height of a polyomino cell. On the one hand, if

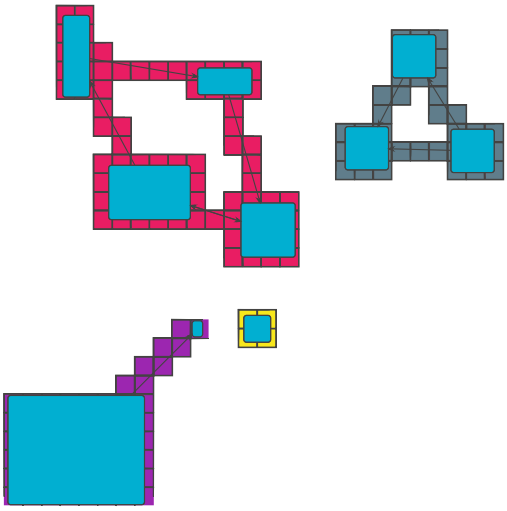


**Figure 4.1.** The same layout as in Figure 1.2a, but this time each component is enclosed by a polyomino approximating its shape instead of a bounding box. The components are internally laid out using the spring embedder ELK Force.

one chooses a big value, the approximation of the shape of the component is inaccurate, which might lead to layouts that are not much more compact than simply using bounding boxes. On the other hand, a small cell size is more precise but working with it is more computationally expensive. Freivalds et al. have done an extensive study on this [FDK02]. They approach the problem of finding a suitable cell size by bounding the average area $s$ of a polyomino, specified in number of polyomino cells, by some constant $c$. Given $n$ components, $s$ depends on the width $w_i$ and height $h_i$, with $1 \leqslant i \leqslant n$, of the components of the original graph. $l$ then denotes the height and width of the polyomino cell with respect to the reference frame of the original graph:

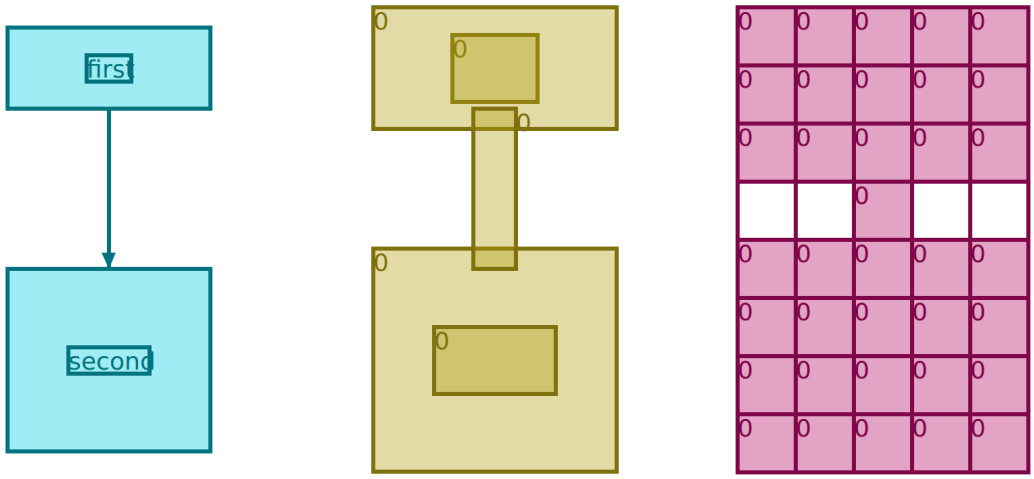
$$s = \frac{1}{n} \sum_{i=1}^{n} \left\lceil \frac{w_i}{l} \right\rceil \left\lceil \frac{h_i}{l} \right\rceil \leqslant c$$

The authors then simplify this equation to a more easily solvable quadratic equation:

$$(cn - 1)l^2 - \sum_{i=1}^{n}(w_i + h_i)l - \sum_{i=1}^{n} w_i h_i = 0$$

The only value missing to determine $l$ at this point is the constant $c$. After extensive testing, Freivalds et al. found that $c = 100$ is a good compromise between accuracy of the polyomino shape and speedy performance.

DisCo simply uses this equation to find the best step size $l$ depending on a the set of components it has to lay out. The conversion of a component to a polyomino is a two step process in the ELK implementation, as illustrated in Figure 4.2. The graph




**(a)** Graph structure **(b)** Polygonal abstraction **(c)** Polyomino

**Figure 4.2.** Transforming a graph component into its polyomino representation: (a) shows the structure of a component consisting of two nodes. (b) The graph has been abstracted to set of polygons. They have a greater area than the original component, as a spacing has been added that is supposed to visually separate the components in the final drawing. (c) A polyomino is obtained by rasterizing the polygons from (b) using the computed step size $l$.

structure is first transformed into a simpler set of polygons, representing nodes, edges, ports and labels. These polygons may have a bigger area than the graph elements they represent. This is because of a configurable component spacing option offered by the implementation. Its purpose is to give the connected components a discrete look in the final drawing, as packing them too tightly might make them harder to discern for people. The dimensions of the bounding box of the final polyomino is then computed in advance and the polygonal representation of the component is put at the center of it. Finally, all polyomino cells that intersect with the polygons are marked as filled.

After converting each component into a polyomino, Freivalds et al. use a heuristic for packing the newly created shapes as tightly as possible by minimizing a cost function [FDK02]. According to the paper the optimal position of a polyomino is the grid cell $G_{xy}$ at position $(x, y)$ with $x, y \in \mathbb{Z}$ such that the function $max(|x|, |y|)$ is minimal and none of its cells overlap with any polyomino already placed on grid $G$. As the authors allow the coordinates to be integral instead of constraining them to natural numbers, components are placed from the origin of $G$ outwards. The authors chose this method to emphasize *symmetry* as an aesthetic criterion in their approach. This complicates drawing graphs on a canvas of fixed dimensions, as one cannot simply fix the origin of $G$ to one of the corners of the underlying canvas, but has to wait for the whole drawing to

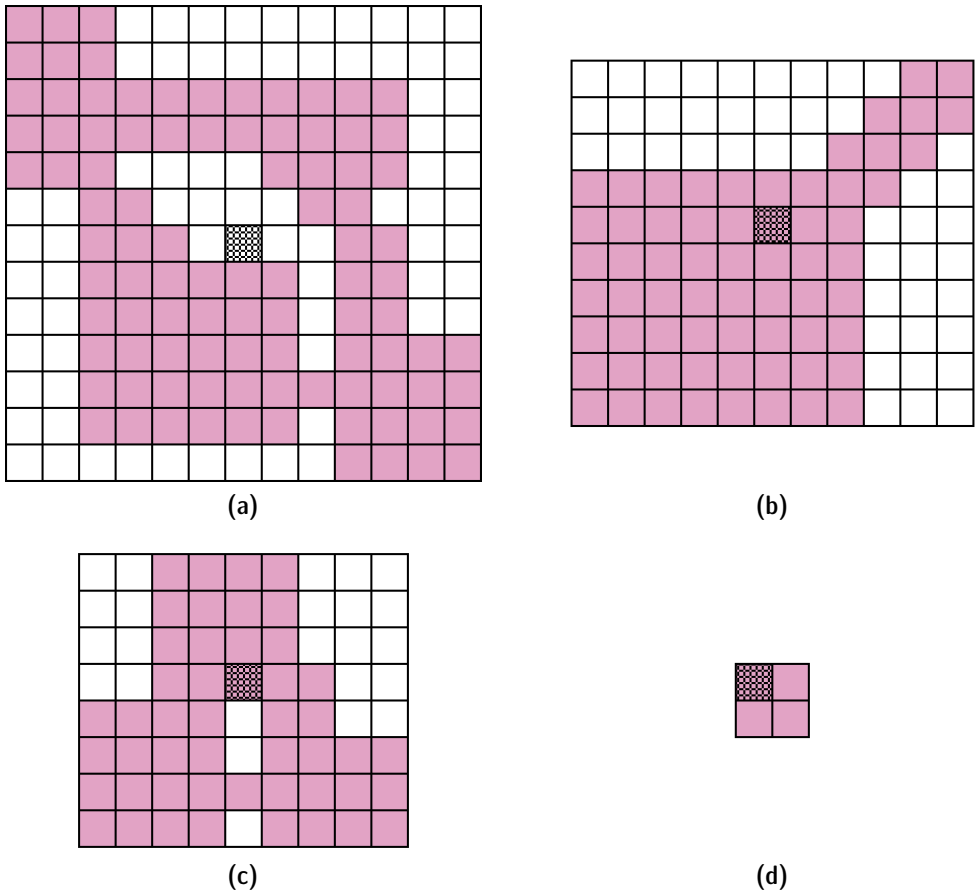



**Figure 4.3.** Some polyominoes marked in pink with their centers highlighted in a black pattern. The centers are determined by taking the rectangular bounding box of the polyomino into account, represented here by the entirety of the drawn grid cells. Please note that the center does not necessarily have to be part of the polyomino as shown in Figure 4.3a.

be laid out. This makes an extra step necessary after the creation of the layout: As the underlying grid structure has in principle to be unbounded to all four sides, a bounding box has to be determined afterwards to place the result onto the drawing area without wasting white space.

The position of a polyomino mentioned earlier refers to its center. The dimensions of the bounding box of a polyomino are used to determine its center and the values for the x- and y-coordinates are truncated if they are not integral, see Figure 4.3.

Moreover, Freivalds et al. also mention that the order the cells of $G$ are examined in is the same for all the polyominoes. But their given cost function is not precise in the sense that $max(|x|, |y|)$ can have the same value for different values of $x$ and $y$. Therefore,

**Algorithm 2:** Pack Polyominoes, from Freivalds et al. [FDK02]

**input :** $n$ polyominoes $P_i$, $1 \leqslant i \leqslant n$

```
1 sort P_i, 1 ⩽ i ⩽ n in the order of decreasing size
2 initialize the grid G using the sizes of P_i, 1 ⩽ i ⩽ n
3 foreach polyomino P_i do
4     calculate (x, y) such that the cost function is minimized
5     while cannot place P_i centered at (x, y) do
6         calculate next (x, y) using the cost function
7     mark the cells in G covered by P_i as occupied
```

there are many possible ways to traverse candidate positions.

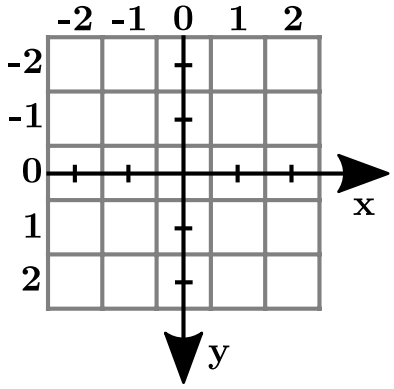
This thesis introduces three different traversal orders for possible cells, all of them respecting the given cost function but handling ambiguous cases differently: one proceeds in a *spiral* pattern, the other walks through cell candidates of equal cost *line by line*, the third one is supposed to look random, jumping between candidate cells in all cardinal directions, and is thus called *jitter*. Additionally a second cost function, $|x| + |y|$, is added based on the *Manhattan distance*, penalizing candidate positions being located diagonally from the origin. Figure 4.4 shows all implemented traversal orders.

Furthermore, the quality of the packing depends on the order in which the polyominoes are placed on the grid. The authors claim that the best results are achieved by ordering polyominoes by decreasing size, where the size is approximated by the perimeter of the bounding box of the polyominoes. Goehlsdorf et al. who build upon the polyomino approach by Freivalds et al. propose an alternative sorting method for deciding which component to layout first [GKS07]: their polyominoes are ordered by decreasing values of the formula $s_{short}^2 + s_{long}$, where $s_{short}$ is the short side and $s_{long}$ is the long side of the minimal bounding box encompassing the polyomino. This measure not only favors bigger components but also ones which are not elongated but have rather squarish dimensions. As both approaches are easy to implement, Chapter 5 will evaluate their quality.

The pseudo code of the algorithm is given in Algorithm 2 and Figure 4.5 shows the placement of some components using this first implemented polyomino strategy for ELK in more detail.

### 4.1.2 External Extensions

ELK provides one feature usually not considered by algorithms for laying out graphs: nodes can contain graphs themselves. This nesting of graphs can be applied to nodes of any level, meaning inner nodes in the contained graphs can also have graphs inside of




**(a)** Coordinate system

| | | | | |
|---|---|---|---|---|
| 12 | 13 | 14 | 15 | 16 |
| 11 | 2 | 3 | 4 | 17 |
| 10 | 1 | 0 | 5 | 18 |
| 9 | 8 | 7 | 6 | 19 |
| 24 | 23 | 22 | 21 | 20 |

**(b)** Spiral traversal

| | | | | |
|---|---|---|---|---|
| 9 | 10 | 11 | 12 | 13 |
| 14 | 1 | 2 | 3 | 15 |
| 16 | 4 | 0 | 5 | 17 |
| 18 | 6 | 7 | 8 | 19 |
| 20 | 21 | 22 | 23 | 24 |

**(c)** Line-by-line traversal

| | | | | |
|---|---|---|---|---|
| 24 | 17 | 9 | 13 | 21 |
| 16 | 8 | 1 | 5 | 18 |
| 12 | 4 | 0 | 2 | 10 |
| 20 | 7 | 3 | 6 | 14 |
| 23 | 15 | 11 | 19 | 22 |

**(d)** Jitter traversal

| | | | | |
|---|---|---|---|---|
| | | 5 | | |
| | 12 | 1 | 6 | |
| 11 | 4 | 0 | 2 | 7 |
| | 10 | 3 | 8 | |
| | | 9 | | |

**(e)** Manhattan traversal

**Figure 4.4.** Implemented traversal orders shown on a clipping of the base grid (a). The first three methods (b-c) show different possibilities of enumerating the first 24 candidates according to the cost function $max(|x|, |y|)$. The last one (e) is an enumeration of the first twelve candidates adhering to the Manhattan distance (the thirteenth position would be above the fifth, outside of the given clipping). Cells of the same cost are highlighted alternating between white and black.

them and so on.

How these inner graphs can relate to structures placed outside their parent node is shown in Figure 4.6a. The upper left outer node (blue) is connected to the bottom

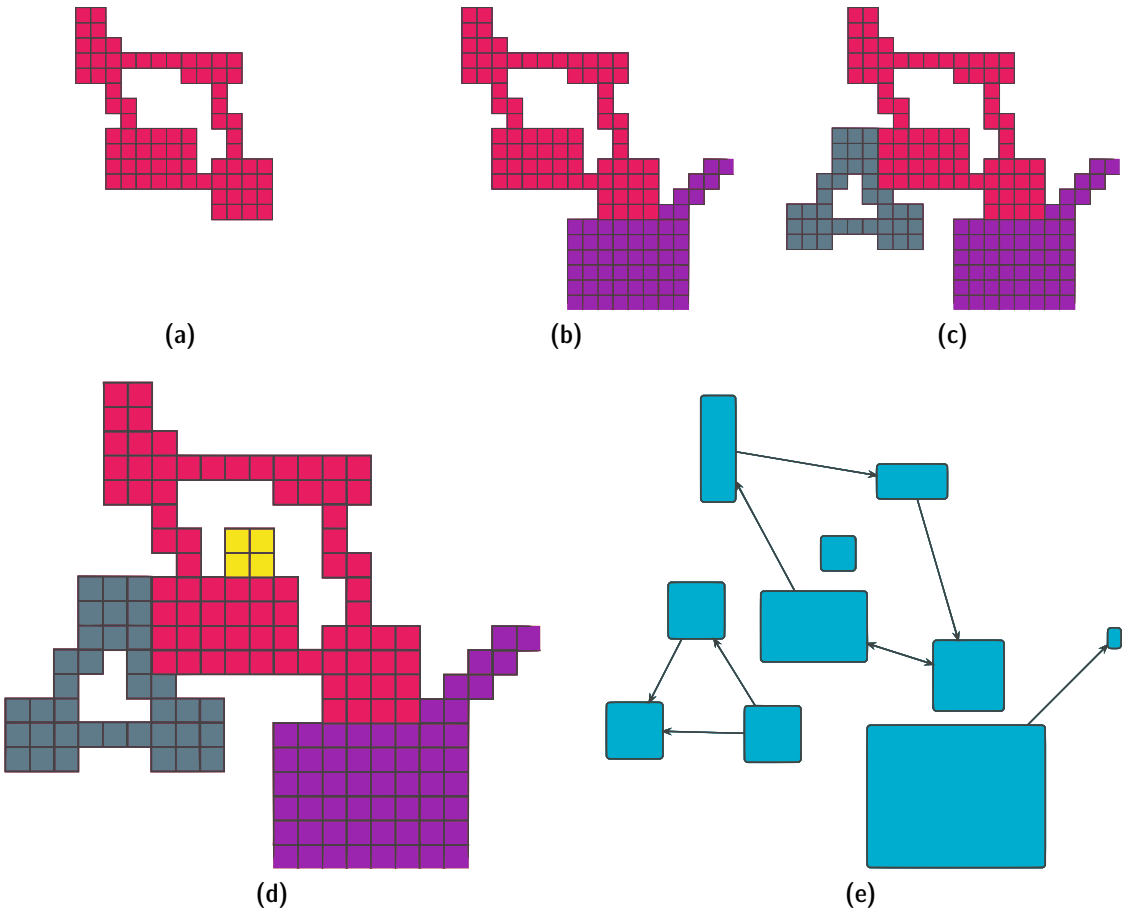



**Figure 4.5.** Placing polyominoes one by one: (a) the largest polyomino is placed first in the center of a discrete grid (not shown), where each unit has the same width and height as a quadratic cell of a polyomino. (b, c) The next two polyominoes are placed as near to the center of the grid as possible, without overlapping with any other polyomino. The resulting layout is not always optimal as a heuristic is applied. (d) The last polyomino (yellow) is small enough to fit into the hole in the center of the first polyomino (pink). It is placed there in order to be as close to the center of the underlying discrete grid as possible. (e) The resulting layout is different from the current approach of ELK shown in Figure 1.2a.

inner node (yellow) via two edges: one is leaving the outer node and ends in a port at the border of the large parent node. The second short hierarchical edge leaves this port inwards and ends in the bottom inner node. Another way of connecting nodes inside the parent node with nodes outside of it is a long hierarchical edge connecting the two nodes directly, as shown by the edge connecting the upper right outer node with the upper internal node. As mentioned in Section 2.1 the second edge can easily be transformed

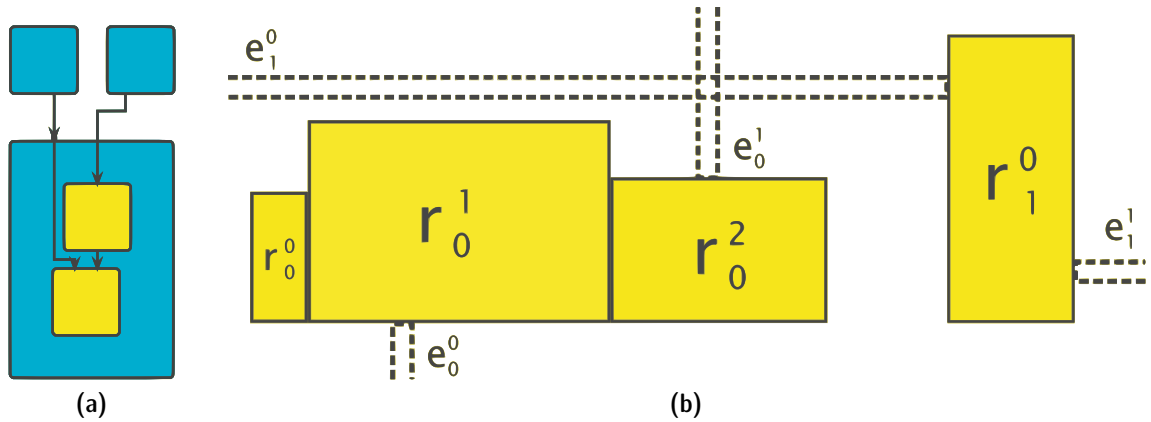



**Figure 4.6.** (a) A valid graph in ELK: any node of a graph (blue) can contain a graph itself (yellow). Outer and inner nodes can be connected by using a port of a parent node (left) or simply by direct edges (right). (b) Example for external extensions taken from Rüegg et al. [RSG+16b]: the two components $r_0$ and $r_1$ are represented by groups of rectangles. Each of them has two external extensions. $e_0^0$ and $e_0^1$ of $r_0$ are directed to the south and the north, whereas $e_0^1$ and $e_1^1$ of $r_1$ lead to the west and the east.

into two split edges connected by a port on the surface of the hierarchical node in this example.

The standard approach from the previous subsection based on Freivalds et al. does not consider this use case at all [FDK02]. Making the polyomino approach work with these kind of edges is an essential step to make the algorithm available to a broader range of diagrams.

The layout of a hierarchical graph can be computed in a bottom-up fashion as explained in Section 2.2 and long hierarchical edges can be split into a set of short hierarchical edges separated by ports whenever the surface of a hierarchical node is crossed along the way, as illustrated in the example on the left of Figure 4.6a. Therefore it is enough to handle edges connecting a node of a component to a port of a single surrounding parent node. An implementation solving this special case can then be generalized by using a mechanism already present in ELK. Section 6.2 covers this reasoning in more detail.

One approach for this problem has been explored by Rüegg et al. [RSG+16b; RSG+16a]. They call the constraint enforced by an edge connecting a port of an inner node with a port of its parent an *external extension*. The authors decided to model these extensions with straight beams leaving their respective components at the point where the edge connected to a port of a hierarchical node connects to one of its inner nodes, as shown in Figure 4.6b. They can have one of the four cardinal directions (north, south, east, west) depending on the border the port of the parent connected to the edge is

placed at.

The goal of laying out components is then to prevent components from overlapping and to not let external extensions overlap with other components. External extensions are allowed to overlap with each other, as the edges represented by them are allowed to do so. One minor goal is to minimize the actual length of the stretch between an external extensions starting point and the border of the parent node surrounding the inner graph.

Rüegg et al.'s placement method for the components differs from the polyomino approach described earlier, but the concept of external extensions as indefinite beams can be transferred to the polyomino based algorithm DisCo.

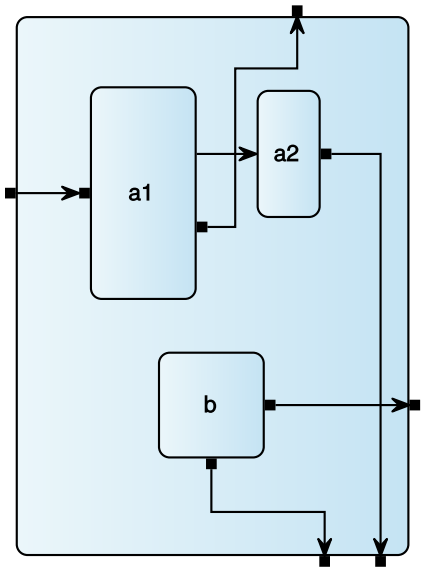
However, the integration of external extensions into DisCo has some limits for now. It only works well, if the short hierarchical edges of a graph are drawn with orthogonal line segments. Furthermore, not the whole edges are treated as external extensions, but rather the last edge segment connecting to the parent node is. The other segments are treated as filled cells of the polyomino. The reason for this constriction is that, at the time of writing, ELK does not have a stand alone edge routing algorithm that can lay out only a few selected edges of a partially finished layout. Implementing this feature alone offers enough problems to warrant its own Master's thesis. Until this new edge router has been introduced to ELK, DisCo simply adjusts a short hierarchical edge by stretching or compressing its line segment connecting to the parent depending on how the distance between a component and its parent node has changed after DisCo was applied. This is easy to implement as an interim solution.

Keeping these constraints in mind, external extensions are realized in DisCo as modeled in Figure 4.7:

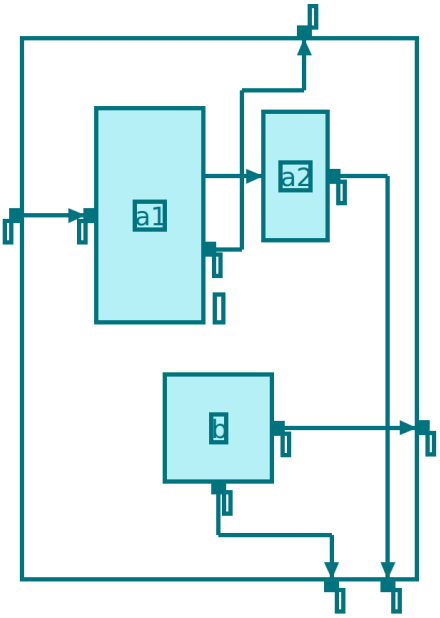
- Figure 4.7a shows a rendering of a small example already laid out in ELK using DisCo. It shows a component consisting of two nodes and three hierarchical edges connecting to ports at the north, east and south side of an enclosing parent node. A second component consisting only of one node has two edges connecting to ports on the east and south side of the parent node.

- Figure 4.7b reveals the internal structure of the graph. Compared to the fully rendered graph, there are some little extra rectangles visible. These are simply empty text labels describing the ports the edges connect to. As they have no content they are not rendered in the final drawing.

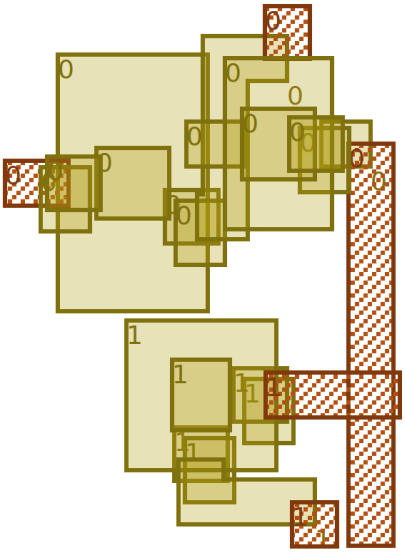
- Figure 4.7c shows the first processing step for extensions. The parent node is no longer drawn in this illustration. The graph is transformed into a simpler polygonal representation such as in Section 4.1.1. However, the last segments connecting to the parent node have been singled out. They are denoted by boxes with an orange pattern in this example. These segments are no longer saved as polygon, but are modeled



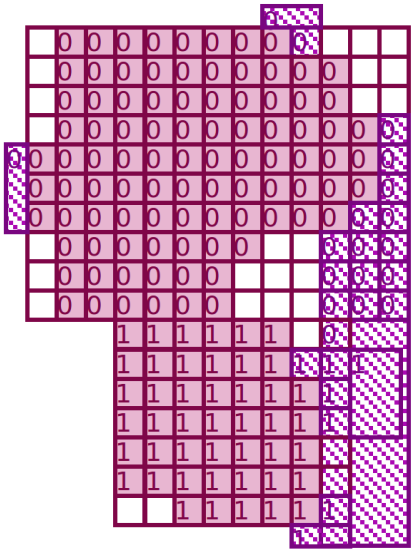

**(a)** Graph rendering




**(b)** Graph structure



**(c)** Polygonal abstraction



**(d)** Polyomino representation

**Figure 4.7.** Transformation of hierarchical graph components into polyominoes. The small extra rectangles visible in the graph structure (b) compared to the rendering of the graph (a) are empty text labels not shown in the final drawing.

like the potentially endless beams from Rüegg et al. [RSG+16b; RSG+16a]. They can be described by a triple $extension_{polygon} = (d, \delta, w)$, where $d$ is the direction the extension is pointed at. This either north, south, east or west. $\delta$ is simply the position where the extension connects to the component given as an offset in $\mathbb{R} \times \mathbb{R}$ relative to the origin of the component, which is in the upper left corner. $w \in \mathbb{R}$ is the width of the beam.

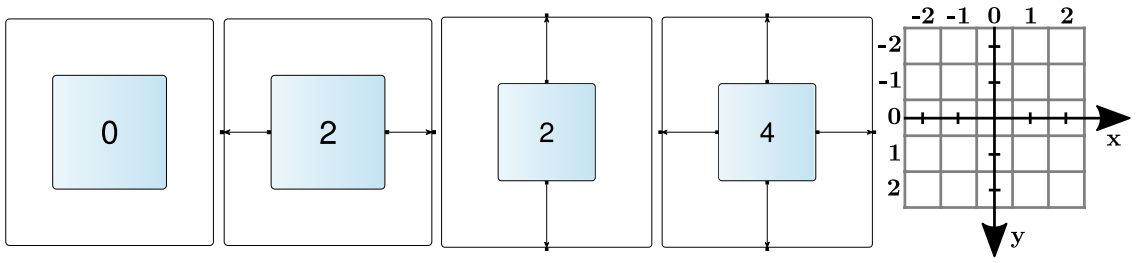
▷ Figure 4.7d displays the final polyomino representation. In Section 4.1.1 a polyomino cell could be in one of two states. It could either be *empty*, which is indicated by a white grid cell, or *filled*, which is indicated by a solid pink grid cell. However, these two states are not able to represent an external extension. Extensions cannot be labeled empty, as they are not supposed to interfere with filled cells, but they do not behave like filled cells either. As they are allowed to intersect with one another, just as edges in the final drawing are allowed to intersect, too. A third cell state, called *weakly filled*, is introduced to model this behavior. The polyomino is first rasterized as in Section 4.1.1. After that polyomino cells are only marked as weakly filled if they intersect with an extension and are not already marked as filled. To account for the endlessness of the extensions beyond the grid cells of the polyomino, extensions are additionally modeled by a triple $extension_{polyomino} = (d, \delta, w)$, but this time with $\delta \in \mathbb{N} \times \mathbb{N}$ and $w \in \mathbb{N}$ in relation to the polyomino, which uses discrete coordinates.

The last subsection included a presentation of different traversal methods for finding candidate positions when placing polyominoes. Recalling Figure 4.4, polyominoes are placed outwards from the origin of an underlying grid $G$, which has integral and not natural coordinates. Using the origin of $G$ to divide the positions of $G$ into four quadrants, it becomes apparent that not all quadrants provide equally good candidate positions for all polyominoes, as the quality depends on the direction the extensions of each individual polyomino point to. For instance, the small component from Figure 4.7 should not be placed in the upper left corner of a drawing because the length of its south and east facing extensions would be shorter if placed in the bottom right corner.

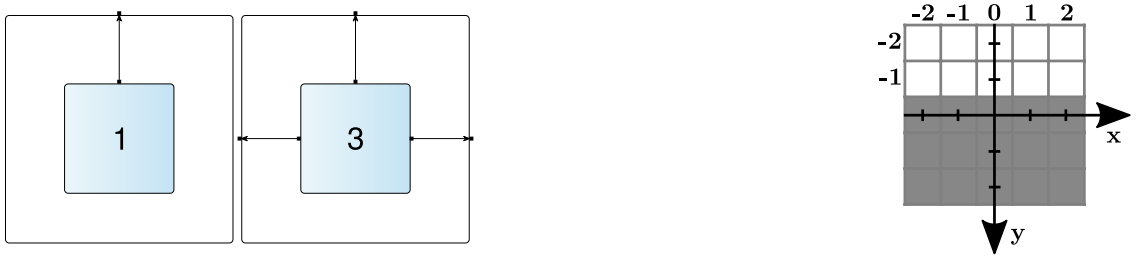
In general there are three cases to consider, see Figure 4.8:

*All quadrants are feasible.* This is the case if a component does not have any extensions, extensions only facing two opposing sides, or extensions in all four cardinal directions. They can be placed at any given candidate position.

*Two neighboring quadrants are feasible.* If a component has only extensions facing either one or three directions, candidate positions from two specific neighboring quadrants should be dismissed when placing the polyomino. These are the two quadrants extending into the opposite direction of the extensions of the given polyomino in




**(a)** All quadrants feasible




**(b)** Two quadrants feasible


**(c)** One quadrant feasible

**Figure 4.8.** Examples for feasible quadrants for polyominoes with extensions: some simple example components with short hierarchical edges are shown at the left, while the feasible quadrants for these examples are shown on a clipping of the underlying grid $G$. The positions covered in gray are not considered when placing the example components. Note that the origin of $G$ is treated as if it belonged to the quadrant in which the values of $x$ and $y$ are positive.

the case of it having extensions only facing one side, or the ones extending into the direction of the side no extension is facing in the case of a polyomino having extensions facing three directions,

*Only one quadrant is feasible.* If the polyomino has extensions only facing two directions orthogonal to each other, only candidate positions from one quadrant should be considered. This quadrant is the one extending into both directions the extensions of the polyomino are facing.

DisCo incorporates these constraints in the following way: the traversal methods

presented in Section 4.1.1 are still used, but each polyomino is treated individually based on its extensions. When a traversal method returns a candidate position outside of feasible quadrants for a specific polyomino, the position is skipped until the traversal method offers a feasible position.

This is one way traversal methods are adapted to better suit external extensions. Another consideration worth exploring is whether polyominoes with extensions should use other traversal methods in general as opposed to ones without extensions. The Manhattan traversal was introduced into DisCo as a tool for placing polyominoes with external extensions in particular. The reasoning is that penalizing diagonal steps from the origin of the base grid keeps polyominoes with extensions from clustering in the corners of the final layout, which might be more visually appealing. Chapter 5 evaluates some combinations of the already introduced methods.

A last refinement with respect to extensions affects the order of polyominoes placed on the grid $G$. Previously, polyominoes have been sorted by decreasing size as proposed by Freivalds et al. [FDK02] or by decreasing size while also preferring squarish shapes as Goehlsdorf et al. suggest [GKS07]. However, polyominoes with extensions should be placed last to ensure that they are closer to the boundaries of their surrounding parent node, so that the length of short hierarchical edges can be minimized. Therefore, DisCo uses the following as the main sorting criterion:

- ▻ Polyominoes with no extensions, only extensions on opposing sides or extensions on three or all sides are placed first.
- ▻ Polyominoes with extensions only connecting to one side are placed next.
- ▻ Polyominoes with extensions only facing two orthogonal directions are placed last as they have the least possible candidate positions to begin with.

Within these three constraints, polyominoes are sorted by increasing number of extensions. If there are still polyominoes considered equal after these steps they are ordered by the previous sorting conditions proposed either by Freivalds et al. or Goehlsdorf et al.

This last adaption concludes the treatment of short hierarchical edges via external extensions. The next section provides some adjustments of the algorithm to make drawings produced by DisCo more readable. It also contains some example drawings, see Figure 4.12.

## 4.2 Enhancing Readability

So far, this chapter demonstrated how to layout connected components using polyominoes to minimize the total area of the resulting drawing. Making a minimal area the only

priority does not take into account how readers of a graph actually comprehend the semantics of it, though. Therefore, this section will describe two optional adaptions to make drawings more easily readable to humans.

### 4.2.1 Filling Gaps in Polyominoes

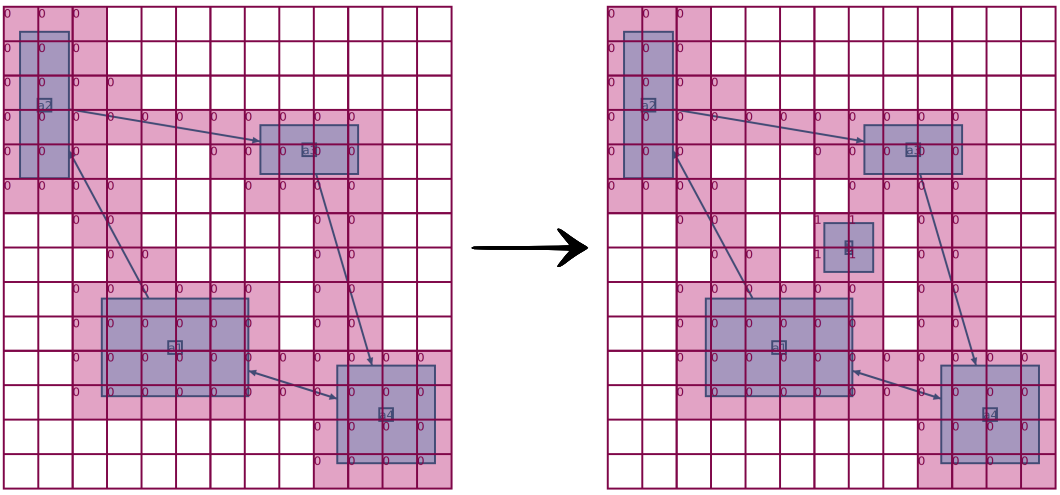
Chapter 1 already mentioned that a hierarchical graph provides two kinds of relations between nodes that can be used to attribute different meanings depending on the semantics of the graphical model. Nodes can relate either by edges or by containment. But there is a third, albeit less formal, relation people tend to give meaning to and that is proximity. Putting nodes adjacent to each other suggests a togetherness on its own, even without connecting edges.

Applied to DisCo, this aspect of perception might become a problem. The approach introduced so far allows for polyominoes having holes as illustrated in Figure 4.9a, A smaller polyomino, e.g. one representing an isolated node, might then be placed within the hole. The small component can then be perceived as having a semantic relation to the larger component surrounding it, even though the placement was rather arbitrary. To prevent such perception manipulating placements, a simple filling algorithm has been implemented.

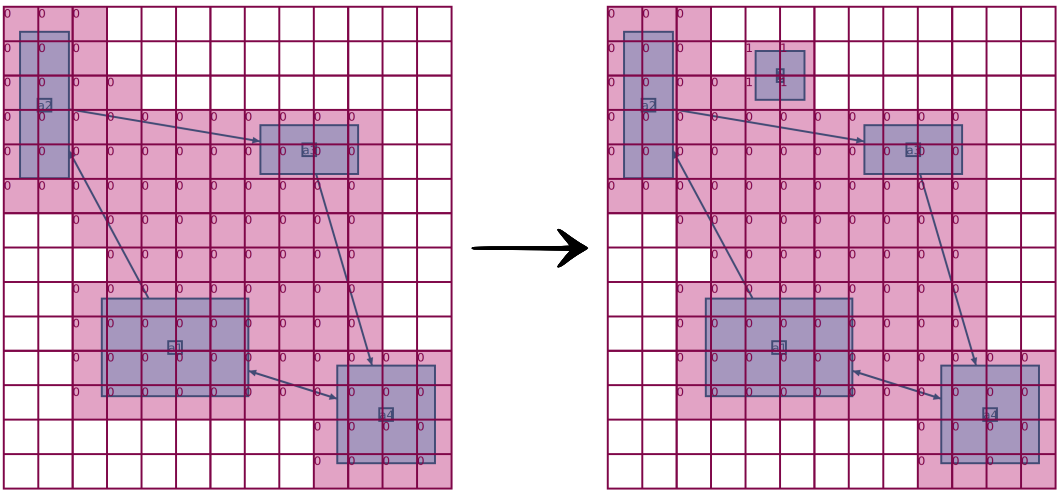
The following procedure is called *profile fill*, as it is based on the notion of *profiles* of polyominoes introduced in Goehlsdorf et al., even though they use profiles for a different purpose [GKS07]. Figure 4.10 shows a polyomino with four profiles, one for each of the sides of its bounding box. A profile can be implemented as a list or an array, where each element contains the number of empty polyomino cells visible from the respective side of the polyomino's bounding box until a filled cell is reached. For each profile this can be implemented in a nested loop with a time complexity of $\mathcal{O}(w \cdot h)$, where $w$ is the width of the polyomino in cells and $h$ is its hight. Together, the four profiles bound the area of the polyomino which should be filled. Another nested loop uses this information to fill the cells belonging to holes of the polyomino. Therefore, the whole algorithm is in $\mathcal{O}(w \cdot h)$ time complexity for each polyomino. This is acceptable, as the polyominoes are typically of low resolution.

Referring back to Figure 4.9b where profile fill has been applied to a big polyomino previously containing holes, the example shows that the approach successfully prevents big components from engulfing smaller ones. The algorithm also works with external extensions. Extensions are simply ignored as weakly blocked cells are treated as empty ones.

The profile fill approach has some drawbacks, however. It only works as expected if all the filled cells of a polyomino are connected. One would assume that this is the case, as in a component all nodes are connected by edges. But ELK supports free floating



**(a)** Polyominoes without filling



**(b)** Polyominoes with filling

**Figure 4.9.** Example for polyominoes with and without applying a simple filling algorithm: a connected component consisting of four nodes (blue) is overlaid with its polyomino representation (pink, each cell marked with 0). Then a much smaller isolated node is placed next (polyomino cells marked with 1). (a) The polyomino representing the bigger component has a hole in its center. The smaller polyomino of the single node fits into this gap and is thus placed inside the first component. (b) The simple filling algorithm has closed the gap in the middle of the big polyomino. The small polyomino is then placed next to the big one.

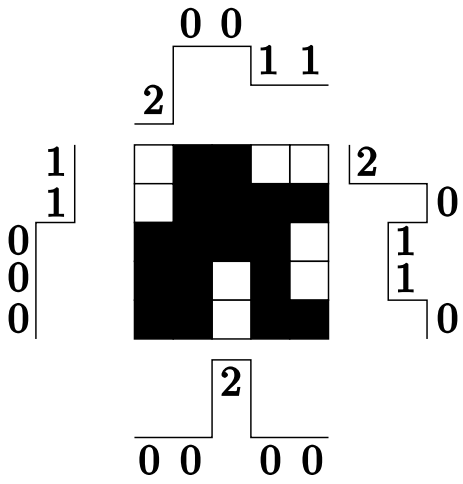



**Figure 4.10.** A polyomino with four profiles. Each number denotes the number of cells separating the respective edge of the bounding box of the polyomino from the first filled cell. The illustration is taken form Goehlsdorf et al. [GKS07] and adapted to better reflect the new purpose of profiles in this work.

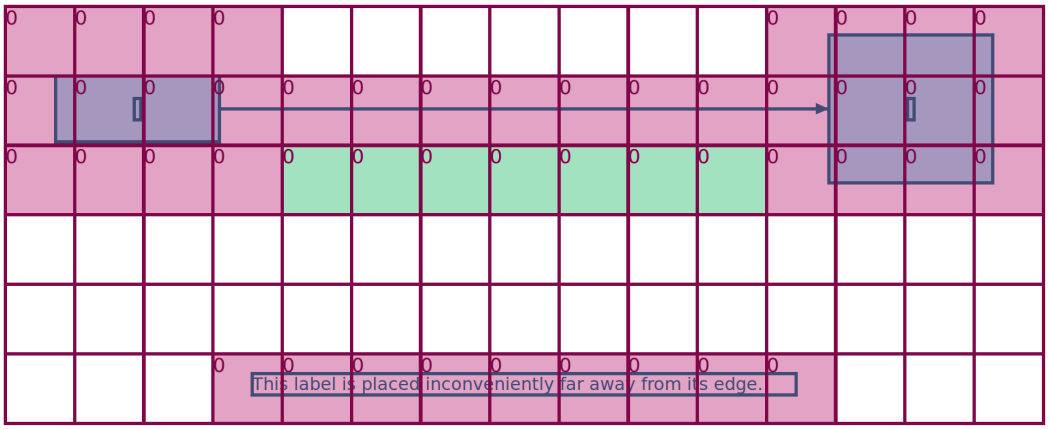
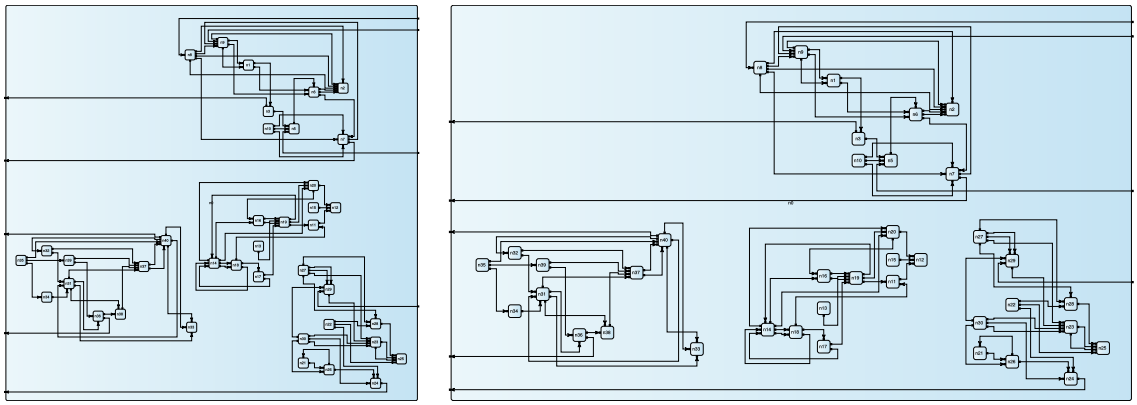
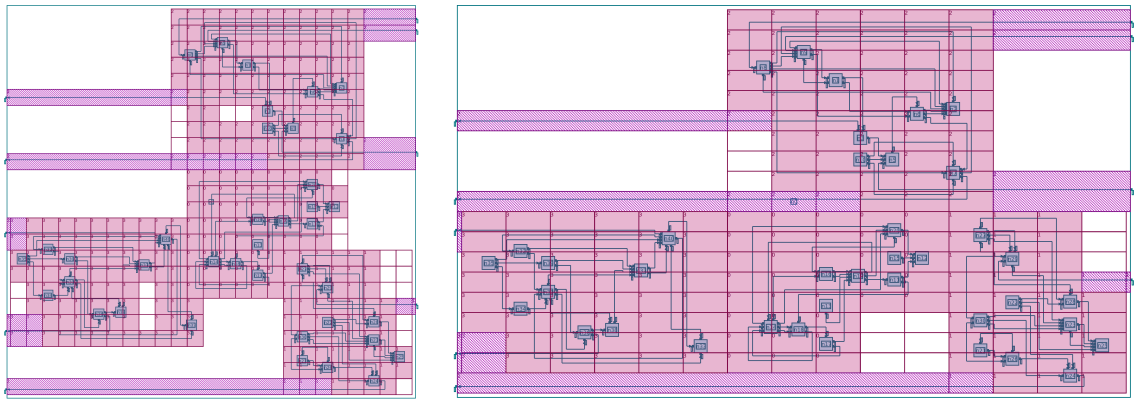



**Figure 4.11.** A rather extreme example of a label placed far away from the edge it describes. Using the profile fill approach does not close the gap between the label and its edge, as it is not a closed hole. Small components could still be placed within the gap. Furthermore, the algorithm fills previously empty cells (marked in green) because they are not visible from any of the four sides of the bounding box of the polyomino.

text labels for annotating other elements of the graphs, i.e. edges. Figure 4.11 shows a component with a text label placed sufficiently far away from its source edge that there is a significant gap between polyomino cells. This gap persists after using the current filling method. See Section 6.2 for suggestions for alternative approaches to filling.



**(a)** Graphs rendered in ELK



**(b)** Polyomino overlay

**Figure 4.12.** Drawing the same graph with two different aspect ratios: (a) the left drawing approximates an aspect ratio of 1.0, while the right one approaches 2.2. (b) Superimposing the underlying polyominoes reveals the distorted cells, each having the desired aspect ratio. The graphs use ELK Layered to lay out the nodes, using orthogonal edge routing. DisCo uses the traversal method *jitter* and does not fill holes of components in this example. The secondary sorting criterion after considering the extensions of each polyomino is the one proposed by Goehlsdorf et al. [GKS07].

### 4.2.2 Supporting a Desired Aspect Ratio

Another feature which might improve the readability of drawings produced by DisCo is the ability to specify a desired aspect ratio the layout algorithm is supposed to approach. There typically is a bounded canvas the final layout will be drawn upon, e.g. a computer screen or a sheet of paper. The implementation presented so far roughly approximates a square layout, which leads to wasted space on differently shaped canvases. Using this wasted space instead by adjusting the layout to the given reference frame enlarges the

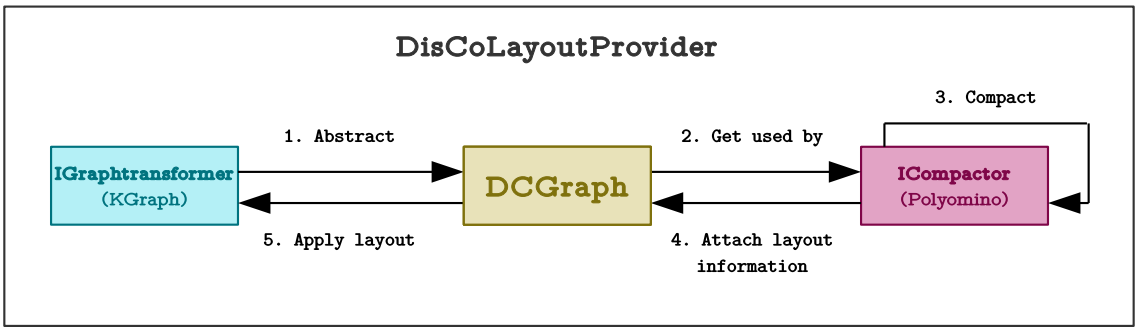



**Figure 4.13.** A schematic of key components of the DisCo algorithm

features of the graph and makes them more discernible.

The polyomino approach makes accommodating a desired aspect ratio $a_{ratio}$ a fairly easy affair, as one can simple adjust the way components are rasterized into polyominoes without touching the core algorithm to achieve this goal. Freivalds et al. mention this in their paper [FDK02]. So far, polyomino cells had the same width $w_{step}$ and height $h_{step}$, using a computed step size $l$. DisCo supports a desired aspect ratio by adjusting the step size in one dimension. If $a_{ratio} > 1.0$ then the step width is adjusted to $w_{step} = l \cdot a_{ratio}$, otherwise the step height is changed to $h_{step} = \frac{l}{a_{ratio}}$. Figure 4.12 shows example drawings of a graph with two different desired aspect ratios using this method.

## 4.3 Implementation in ELK

This chapter concludes with a short high level description of the structure of the DisCo implementation within ELK. It is implemented as a layout algorithm which can be invoked using the `DisCoLayoutProvider`, similar to other algorithms like ELK Force and ELK Force.

As mentioned before, DisCo works on the KGraph which has been replaced by the ElkGraph during its implementation. As this was known in advance, it was of special importance to design the algorithm with modularity in mind to facilitate an easy transition to other graph structures.

The key decision for achieving this is the introduction of a data structure specialized to the problem domain of laying out components, which resulted in the *DCGraph*. Algorithms for laying out nodes need to differentiate between nodes, edges, ports and labels within a component, whereas DisCo only needs to know that a component consists of shapes that should not overlap with shapes from other components. This is why these elements are simply represented by polygons, losing their original identity in the process. Hierarchical edges on the other hand are modeled as extensions within the DCGraph to

accommodate for their indefinite length during the component layout process. Figure 4.2b and Figure 4.7c are visualizations of the actual DCGraph as used in DisCo.

The DCGraph assumes a bridging position in DisCo similar to the one the ElkGraph holds in the greater context of ELK, as illustrated in Figure 4.13. On one end, there are graph structures which are supposed to work with DisCo. A developer only has to implement the `IGraphTransformer` and convert the constituents of their graph structure into suitable elements of the DCGraph. On the other end, there are layout algorithms for connected components using the DCGraph as an interface by implementing the `ICompactor`. As of writing of this thesis, each of these interfaces only has one concrete implementation, i.e. the `KGraphTransformer` and the `PolyominoCompactor`. In the future the DCGraph can be used to switch out graph structures and component placing algorithms independently from each other, however.

Chapter 5

# Evaluation

This chapter presents experimental results with respect to the quality of drawings laid out by DisCo in comparison to the default approaches in ELK. The examined graph sets and used measures are introduced first. The second section pursues four questions regarding the drawing of simple graphs and the third concludes with two issues regarding hierarchical graphs.

## 5.1 Experimental Setup

The evaluation in this thesis uses random graphs created with ELK. This decision was made particularly because of the current constraints of DisCo affecting external extensions. As the implementation only works on graphs consisting of a single parent node containing a single simple graph with short hierarchical edges instead of the whole superset of possible hierarchical graphs, there simply is not any domain specific set of graphs available for testing. Section 6.2 describes how to solve this problem.

For now, this evaluation uses two randomly generated sets of 250 graphs. Section 5.2 will use *simple graphs* generated with the following attributes, which are distributed uniformly if not stated otherwise:

- Each component contains between 5 and 20 nodes.
- A graph consists of between 1 and 50 components in total.

The set of hierarchical graphs for Section 5.3 shares these attributes with the simple graph except for the semantics of the number of components, which refers to the number of components inside one single parent node in this case. They also have some additional attributes, as only hierarchical graphs can have external extensions:

- Each graph has between 0 and 50 hierarchical edges, but these are chosen via normal distribution.
- A hierarchical edge has a chance of 0.5 to be going out from a component leading to the parent node. Otherwise, the edge is starting at the parent node and ending in a component inside that node.

The following measurements will facilitate quantitative statements about drawings of these graph sets produced by different layout methods:

*Area* The area will be measured as the product of width and height of the drawing of a graph in pixels.

*Aspect ratio* Definition 2.1.12 explains the aspect ratio as the division of the width of a drawing by its height.

*Max scale ratio* Definition 2.1.14 introduces this measure in relation to a reference frame $R_{w_r,h_r}$ of a certain width $w_r$ and height $h_r$. The evaluation will adapt this metric to see how well a layout adheres to a specified desired aspect ratio $a_{ratio}$ by using $R_{a_{ratio},1}$ as a reference frame.

These measures are presented as two-dimensional plots with the number of components serving as an $x$-axis, whenever these demonstrate a trend. Less legible measurements are still given in tabular form, showing mean, median, minimum and maximum values over all 250 graphs of the set plus the standard deviation (SD).

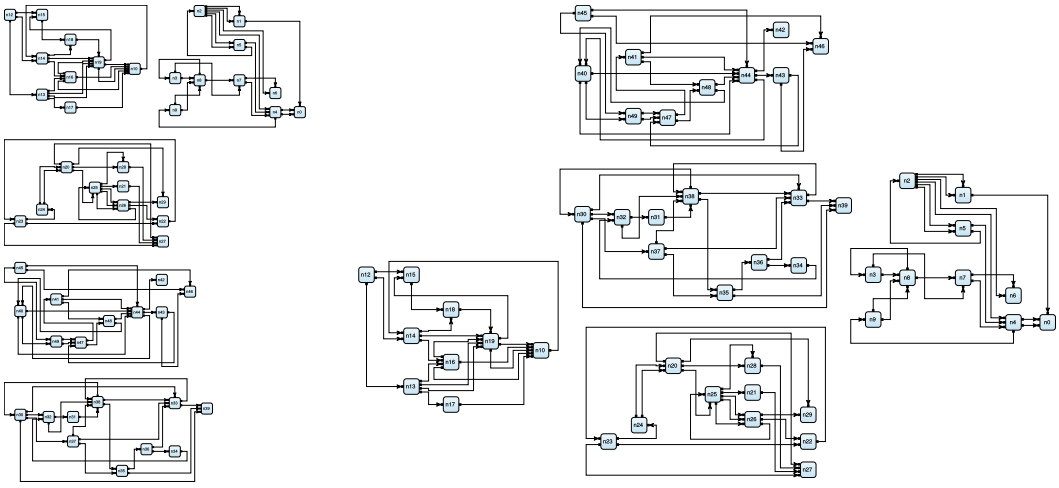
## 5.2 Basic Packing

This section compares drawings achieved by laying out simple graphs. The algorithm used to lay out the nodes of each connected component is ELK Layered with orthogonal edge routing. The difference among the computed layouts is the algorithm used for laying out connected components. As ELK already has a component placer called the *simple row graph placer*, it will be used as a baseline to compare DisCo against. It works like Algorithm 1 from Chapter 1. Components have equal priority in ELK Layered, meaning components will be placed by decreasing size.

DisCo is tested with the four traversal methods from Figure 4.4, i.e. spiral, line-by-line, jitter and Manhattan traversal. The other options stay the same, if not stated otherwise: polyominoes do not use the filling algorithm from Section 4.2.1, they will be presorted only by their size as proposed by Freivalds et al. [FDK02] and the desired aspect ratio is 1. Figure 5.1 shows two example drawings of one of the graphs from the simple graph data set.

These five different methods for laying out simple graphs are then used to answer the following four questions:

1. Does using DisCo result in a more compact drawing compared to the approach already implemented in ELK?



**(a)** Simple row graph placer **(b)** DisCo with jitter traversal

**Figure 5.1.** One example graph from the simple graph data set consisting of five components. The nodes have been laid out with ELK Layered utilizing orthogonal edge routing and a desired aspect ratio of 1. (a) The simple row graph placer is used. (b) DisCo with jitter traversal is used. Polyominoes are not filled and sorted in advance by size only.

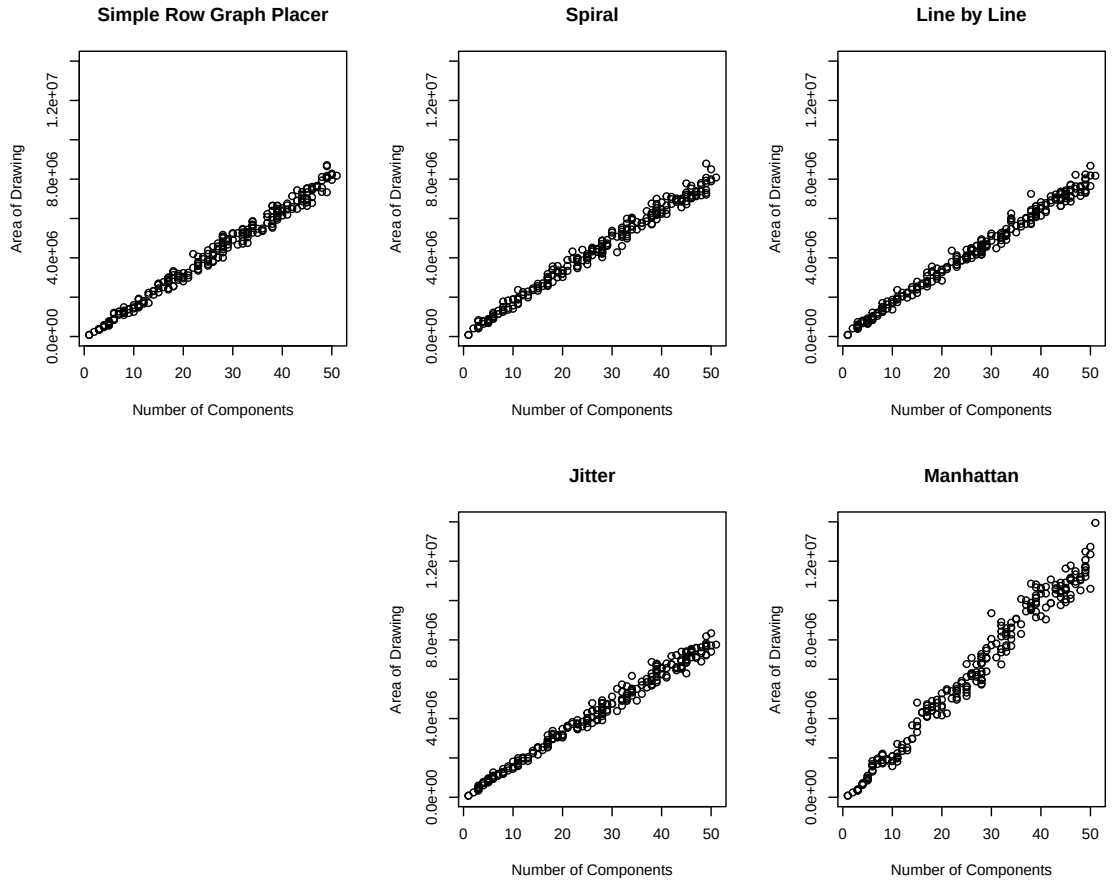
2. Do drawings produced by DisCo adapt better to the reference frame of the canvas they are drawn upon?

3. Does filling the holes in polyominoes affect the overall area of drawings?

4. Should polyominoes be presorted by size alone or should their shape be considered, too?

To answer the first question, have a look at Figure 5.2. The overall area of each drawing is presented relative to its number of components. The resulting plots do not differ that much except for the Manhattan traversal which clearly results in the least compact drawings. The other four methods seem to behave fairly similarly to each other. Looking at the mean area across all graphs shows that DisCo has a slight edge over the simple row graph placer. This means DisCo produces more compact results than the existing approach on this data set, but only if jitter traversal is used. The poor results of the Manhattan traversal are not really surprising, as it produces layouts in a diamond shape which is rotated by 45° with respect to the bounding box used to measure the area of the drawing.

Two measurements help to answer the second question whether drawings produced by DisCo adapt better to a given reference frame: the adherence of the drawings to a specified desired aspect ratio and the max scale ratio. First, see Figure 5.3 for the aspect

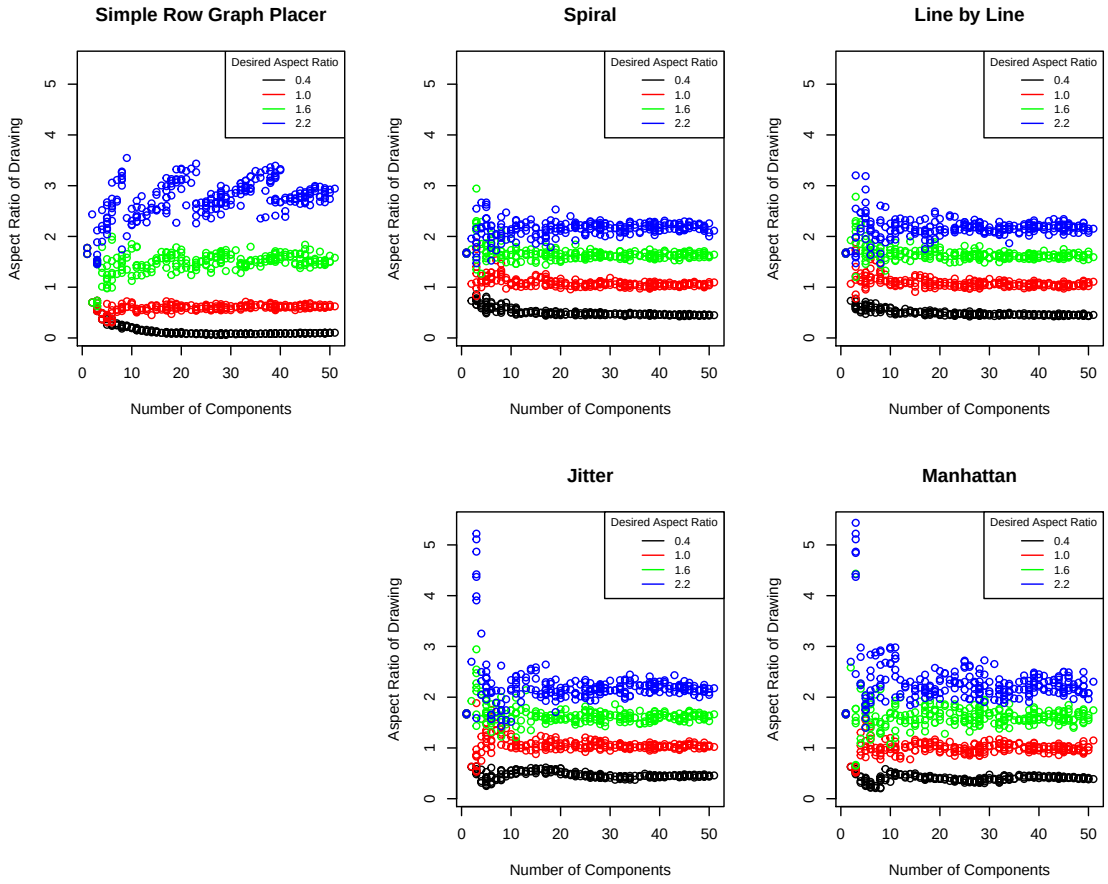



**(a)** Plots of the total area of a drawing depending on the number of components

| Packing Method | Mean | Median | SD | Min | Max |
|---|---|---|---|---|---|
| **Simple Row Graph Placer** | 4213374 | 4412046 | 2354268 | 75756 | 8712347 |
| **Spiral** | 4285624 | 4394608 | 2267760 | 75543.59 | 8790747 |
| **Line by Line** | 4279505 | 4402595 | 2285271 | 75543.59 | 8677257 |
| **Jitter** | 4177734 | 4298240 | 2252319 | 75543.59 | 8337383 |
| **Manhattan** | 6345536 | 6271854 | 3601578 | 75543.59 | 13947530 |

**(b)** Numerical data concerning the area of drawings

**Figure 5.2.** Does using DisCo result in a more compact drawing compared to the approach already implemented in ELK? Comparison of different traversal methods with respect to the area of drawings achieved by them. On average, DisCo with jitter traversal results in the the most compact drawings, followed by the simple row graph placer. Using DisCo with Manhattan traversal leads to the biggest area for drawings overall.




**(a)** Plots of the aspect ratio of a drawing depending on the number of components

| Packing method | 0.4 | 1.0 | 1.6 | 2.2 |
|---|---|---|---|---|
| Simple Row Graph Placer | 0.1550195 | 0.6024506 | 1.496696 | 2.727248 |
| Spiral | 0.5106017 | 1.095779 | 1.661399 | 2.15291 |
| Line by Line | 0.5015837 | 1.095673 | 1.657923 | 2.164024 |
| Jitter | 0.4752542 | 1.073841 | 1.642759 | 2.22425 |
| Manhattan | 0.41105 | 1.000048 | 1.625115 | 2.312406 |

**(b)** Mean aspect ratios for four different desired values

**Figure 5.3.** Do drawings produced by DisCo adapt better to the reference frame of the canvas they are drawn upon? Comparing how well the different methods adhered to a specified aspect ratio, the simple row graph placer performs the worst, whereas DisCo with jitter traversal approximates the target best for example ratios of 0.4, 1.0 and 1.6. Manhattan traversal works best in the 2.2 case. Unfortunately, these two traversal methods also have the most severe outliers when the number of components to place is low.

ratio. Four different desired values were tested to account for tall and short reference frames. 1.0 represents a square layout, whereas 1.6 can be interpreted as a modern 16 : 9 computer display. The plots show that the simple row graph placer has the most trouble adhering to a specified aspect ratio, with jumps clearly visible, especially when an aspect ratio of 2.2 is desired. Moreover, all methods struggle to achieve a given aspect ratio, when there are not a lot of components to place, but DisCo using jitter or Manhattan traversal produces the most severe outliers in this case. In contrast to this observation, jitter and Manhattan traversal approached their desired aspect ratios the best when looking at the means, while the simple row graph placer is the worst overall.

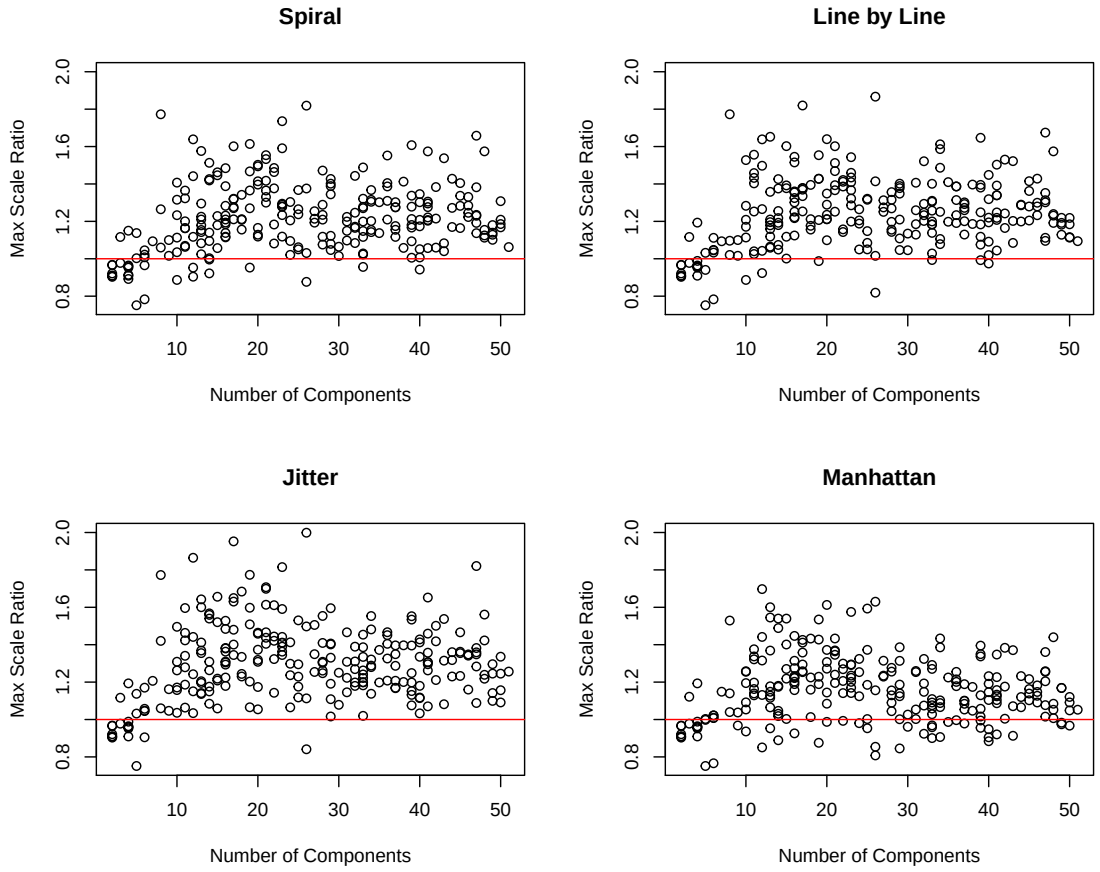
Second, the max scale ratio of the four traversal methods to the simple row graph placer was determined, this time simply using an aspect ratio of 1 as usual, meaning that the reference frame is a square, see Figure 5.4. The plots have a horizontal red line inserted into them, marking the value 1 of the $y$-axis. A traversal method can be displayed larger within a reference frame, if its max scale ratio is bigger than that value according to Definition 2.1.14. As the simple row graph placer has problems with achieving a given aspect ratio, all traversal methods perform better, indicated by their max scale values above the red line for most graphs from the data set. Inspecting the mean values across the four methods shows that jitter traversal works best with respect to scaling a drawing to its reference frame. Manhattan performs the worst but still better than the default approach in ELK Layered.

In summary the measurements regarding aspect ratio and the max scale ratio on the generated simple graph set indicate that DisCo does indeed fill a given reference frame better than the simple row graph placer.

This section concludes by comparing settings of two options of DisCo: the filling of polyominoes and the order polyominoes are placed in.

So far, the polyominoes used have been configured to not fill holes within them. The third question for this evaluation of simple graphs is whether filling the holes in polyominoes affects the overall area of drawings. Comparing the areas of drawings achieved by the four traversal methods either with or without filling polyominoes, there was not a single difference concerning the area of any graph of the generated simple graph set. The data was just like Figure 5.2. Either polyominoes previously placed in holes before using the filling algorithm from Section 4.2.1 have found a free position within the same bounding box defining the area of the drawing, or no components have been placed inside holes of other components to begin with. To investigate this issue further, future work is needed, which might use a data set with components with more variety in size than containing between 5 and 20 nodes.

Finally, the question is examined whether polyominoes should be presorted by size alone or by factoring their shape in, too. Up to this point Freivalds et al.'s method was used considering only the size of polyominoes [FDK02]. The formula used is $s_{short} + s_{long}$,




**(a)** Plots of the Max Scale Ratio comparing DisCo with the simple row graph placer

| Packing Method | Mean | Median | SD | Min | Max |
|---|---|---|---|---|---|
| **Spiral** | 1.21712 | 1.20611 | 0.1833171 | 0.7514672 | 1.818719 |
| **Line by Line** | 1.24347 | 1.226262 | 0.1873242 | 0.7514672 | 1.866779 |
| **Jitter** | 1.299431 | 1.297 | 0.2040914 | 0.7514672 | 1.998827 |
| **Manhattan** | 1.156624 | 1.14522 | 0.1742198 | 0.7514672 | 1.697393 |

**(b)** Numerical data concerning the Max Scale Ratio of drawings

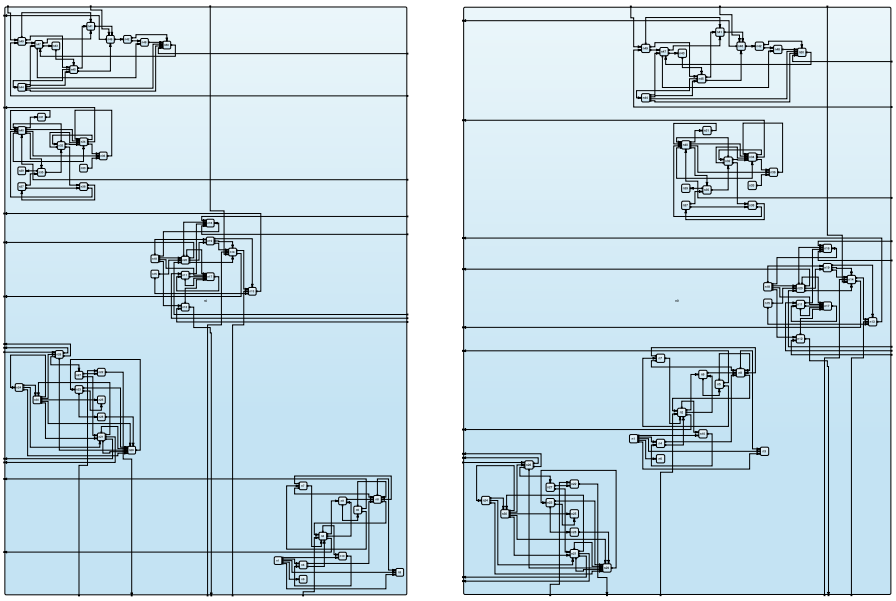
**Figure 5.4.** Do drawings produced by DisCo adapt better to the reference frame of the canvas they are drawn upon? The comparison of max scale ratios shows that all four traversal methods adapt better to a given reference frame. Jitter traversal has the best average performance, whereas Manhattan traversal comes in last.

| Packing Method | Mean | Median | SD | Min | Max |
|---|---|---|---|---|---|
| **Spiral** | 1.041669 | 1.035272 | 0.08025335 | 0.8123966 | 1.319631 |
| **Line by Line** | 1.042985 | 1.037381 | 0.09670763 | 0.8077698 | 1.785296 |
| **Jitter** | 1.036267 | 1.030308 | 0.09219116 | 0.8004416 | 1.68172 |
| **Manhattan** | 1.030706 | 1.022067 | 0.1034553 | 0.7078047 | 1.44459 |

**Figure 5.5.** Should polyominoes be presorted by size alone or should their shape be considered, too? Mean values of the ratio $\frac{area_{Goehlsdorf}}{area_{Freivalds}}$ are computed individually for each graph in the data set. Taking shapes into account when presorting polyominoes actually slightly increases the average area. However, looking at the minimum values reveals that there are instances with a more compact area.

where $s_{short}$ is the shorter side of the bounding box of each polyomino and $s_{long}$ its long side, i.e. half the perimeter of the box. In a later paper Goehlsdorf et al. try to improve the packing of polyominoes and suggest the formula $s_{short}^2 + s_{long}$ as a better alternative, which also favors polyominoes of square shape rather than a prolate one. To find out which sorting criterion fares better, first the area $area_{Freivalds}$ is determined for each graph using the sorting method by size alone. This data is already known because it was presented in Figure 5.2. Using the same settings except for the sorting method the area $area_{Goehlsdorf}$ is computed for each graph also taking the shape of polyominoes into account. To see easily whether the area of a drawing produced by the second approach is smaller, the ratio $\frac{area_{Goehlsdorf}}{area_{Freivalds}}$ is computed for each graph. Whenever the layout achieved by the second sorting criterion is more compact than that of the other one, this ratio should be less than 1. Plotting the values with respect to the number of components of the graphs from the data set does not show a clear trend, so the plots are left out here. However looking at the mean values given in Figure 5.5, the numbers show that Goehlsdorf et al.'s approach actually produces drawings with a slightly bigger area on average and for all traversal methods in DisCo.

Summarizing, these findings indicate that for simple graphs DisCo performs better than the simple row graph placer. The difference in area is not big, but there is improvement with regards to approximating a desired aspect ratio and usage of a given reference frame. The best traversal method to use right now is jitter. Using a filling algorithm for polyominoes does not seem to have an effect, but future studies of other graph sets is advised to look further into this issue. Concerning the sorting of polyominoes, preferring greater sized ones seems to work slightly better on average than also favoring square shaped ones, but both criteria should be compared when laying out a concrete graph.



**(a)** Component group graph placer **(b)** DisCo with jitter traversal

**Figure 5.6.** One example graph from the hierarchical graph data set consisting of one parent node containing five components with external extensions. The nodes have been laid out with ELK Layered utilizing orthogonal edge routing and a desired aspect ratio of 1. (a) The component group graph placer is used. (b) DisCo with jitter traversal is used, constraining feasible candidate positions if external extensions are present. Polyominoes are not filled. The criteria described in Section 4.1.2 serve as the primary sorting mechanism for polyominoes. Polyominoes equal by these criteria are then sorted by decreasing size only.

## 5.3 Packing with External Extensions

One can utilize the same measurements as in the simple graph case for the evaluation of the quality of using DisCo with external extensions. The data set is changed to the one containing 250 hierarchical graphs consisting of a single parent node containing a simple graph as described at the beginning of this chapter.

Some other aspects change with respect to the simple graph case from the previous section: first, the baseline algorithm the traversal orders of DisCo are compared against is no longer the simple row graph placer, as it does not support hierarchical edges. The standard algorithm of ELK Layered for handling this case is the *component group graph placer*. It works like the approach described by Rüegg et al. [RSG+16b], see Section 3.3.

There are some changes concerning DisCo, too. Polyominoes are still sorted by size, but this is now only a secondary sorting criterion, as they are now primarily sorted by their extensions as described at the end of Section 4.1.2. Furthermore, the traversal methods

behave slightly differently from the simple graph case, too. As previously described and illustrated in Figure 4.8, polyominoes might be constrained to certain quadrants of the underlying grid to minimize the length of extensions and the number of hierarchical edge crossings. This evaluation will use the versions of the traversal methods supporting these constraints. Figure 5.6 shows two example drawings of one of the graphs from the hierarchical graph data set.

Finally, a slightly changed set of traversal methods is examined. The spiral traversal makes room for one combined traversal approach: the *line-by-line with Manhattan* traversal method uses the line-by-line traversal for polyominoes without extensions and Manhattan for polyominoes with them. The reasoning for using Manhattan for external extensions is to examine whether layouts would be more appealing this way as mentioned at the end of Section 4.1.2. However, the Manhattan traversal already lead to poor results in the simple graph case from the previous section.

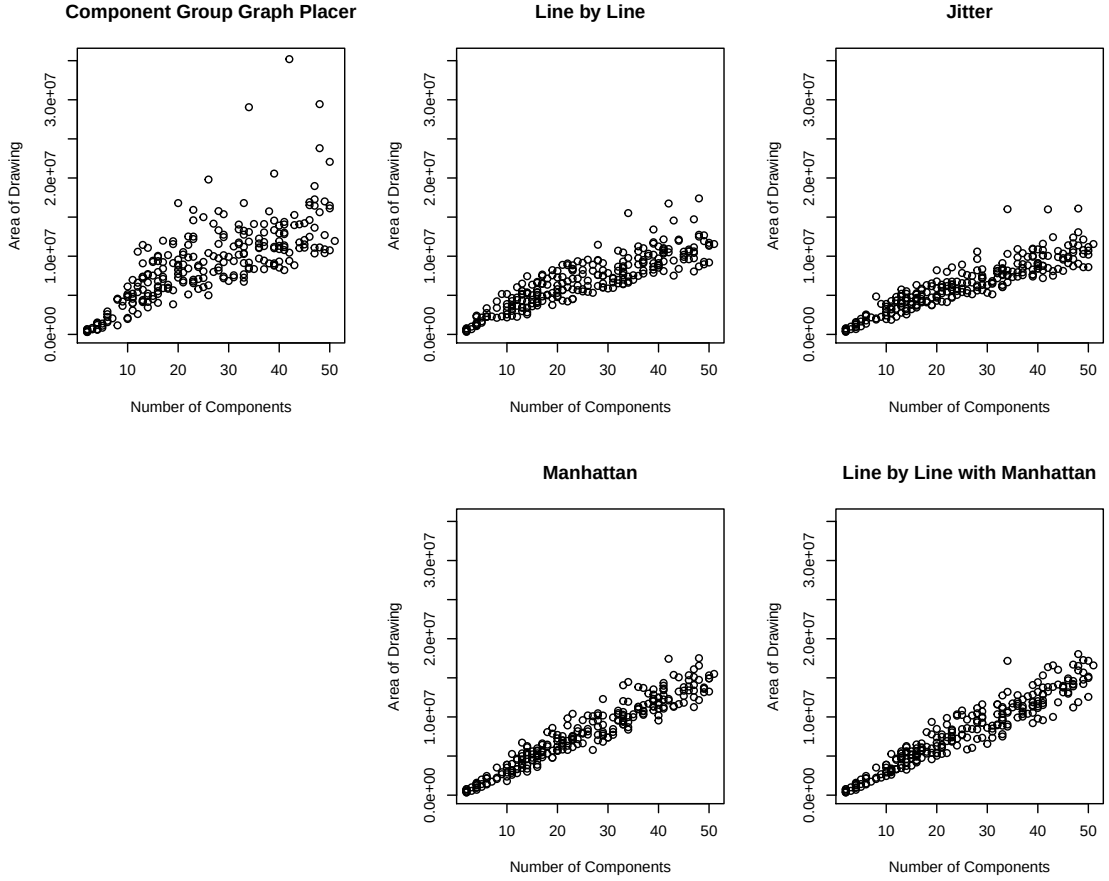
These five different methods for laying out simple graphs are this time used to answer the first two questions from Section 5.2 but with external extensions:

1. Does using DisCo with external extensions result in a more compact drawing compared to the approach already implemented in ELK?

2. Do drawings with external extensions produced by DisCo adapt better to the reference frame of the canvas they are drawn upon?

In contrast to the last section, the plots for the area of each drawing relative to its number of components in Figure 5.7 show apparent weaknesses of the component group graph placer currently used in ELK Layered. Its results show a greater variance for the size of the achieved drawings, especially when a greater number of components is involved. When looking at the mean area over all drawings it leads to the least compact drawings, too. The jitter traversal produces the most compact drawings on average. The combination of the line by line and Manhattan traversal methods actually performs worse than its constituents with respect to compactness, but is still better than the standard approach in ELK Layered. Summarizing, DisCo produces more compact drawings if extensions are involved.

The second question is whether DisCo can adapt better to a given reference frame than the component group graph placer. Like in the last section, the adherence to a specified aspect ratio will be inspected first. The comparison is a little bit unfair, as the component group graph placer does not support the option to specify an aspect ratio, but it is still performed as it is the only component layout algorithm in ELK supporting external extensions at all. Plots of the four desired values of 0.4, 1.0, 1.6 and 2.2 show a greater variance for all five methods and are thusly less easily legible than their counter parts for simple graphs in Figure 5.3a. They are not shown here. Figure 5.8 presents the




**(a)** Plots of the total area of a drawing depending on the number of components

| Packing Method | Mean | Median | SD | Min | Max |
|---|---|---|---|---|---|
| **Component Group Graph Placer** | 9174812 | 9100518 | 5088245 | 304719.8 | 35195204 |
| **Line by Line** | 6937493 | 6992064 | 3265693 | 344352.9 | 17388742 |
| **Jitter** | 6385343 | 6308843 | 3105784 | 344352.9 | 16094488 |
| **Manhattan** | 7974642 | 7732904 | 4171679 | 344352.9 | 17515074 |
| **Line by Line with Manhattan** | 8133374 | 7912027 | 4320216 | 344352.9 | 18037340 |

**(b)** Numerical data concerning the area of drawings

**Figure 5.7.** Does using DisCo with external extensions result in a more compact drawing compared to the approach already implemented in ELK? Comparing the areas of different methods for laying out graphs with external extensions. The component group graph placer produces the drawings of the greatest area on average, whereas DisCo with jitter traversal drawings become the most compact.

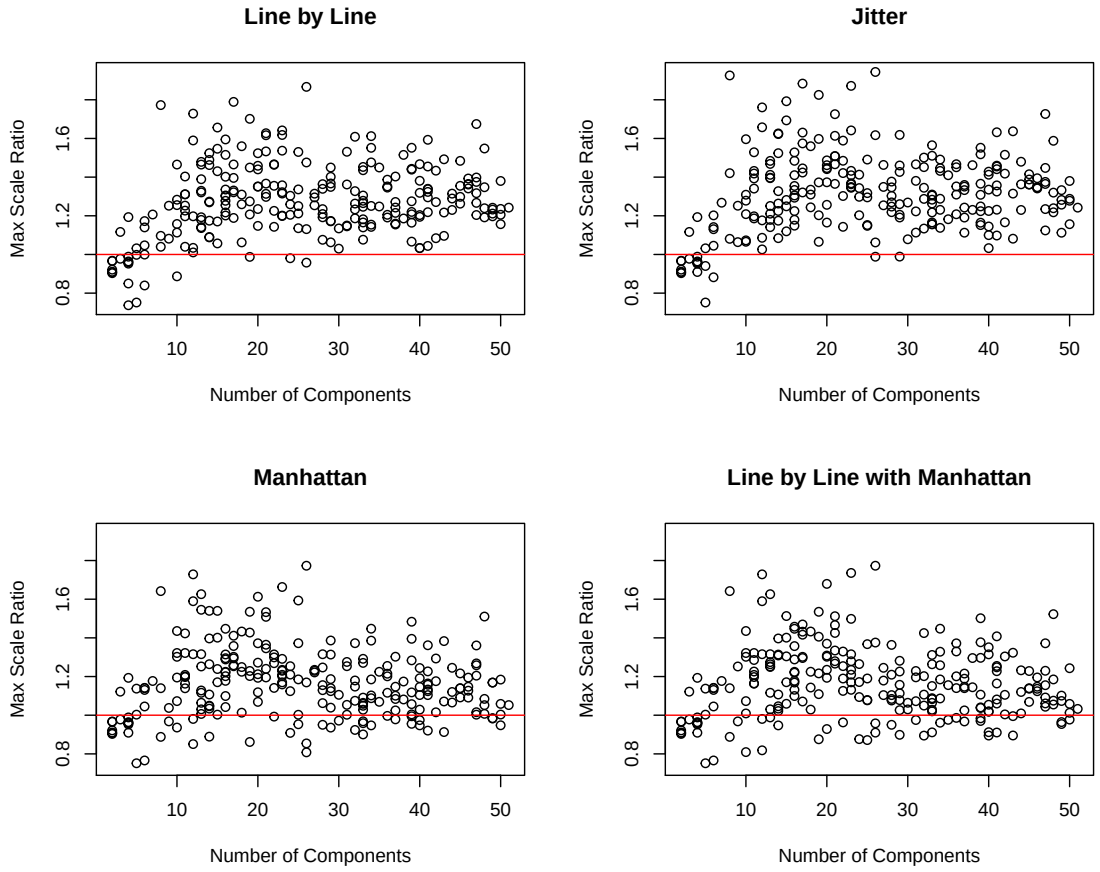
| Packing Method | 0.4 | 1.0 | 1.6 | 2.2 |
|---|---|---|---|---|
| **Component Group Graph Placer** | 0.5814792 | 0.78404 | 0.9880774 | 1.174229 |
| **Line by Line** | 0.4872426 | 1.002439 | 1.475552 | 1.900524 |
| **Jitter** | 0.4930947 | 0.9680382 | 1.379059 | 1.714606 |
| **Manhattan** | 0.4770318 | 0.8866256 | 1.264003 | 1.610365 |
| **Line by Line with Manhattan** | 0.4767735 | 0.8981573 | 1.270254 | 1.578896 |

**Figure 5.8.** Do drawings with external extensions produced by DisCo adapt better to the reference frame of the canvas they are drawn upon? Mean aspect ratios for four different desired values show that the component group graph placer does not approximate a given aspect ratio, as was expected. Line-by-line with Manhattan works best for an aspect ratio of 0.4, the standalone line-by-line traversal wins at the other three values.

mean values over all 250 graphs instead. The component group graph placer approximates a given aspect ratio most poorly. Differing from the simple graph case, jitter traversal is not the best performing traversal method. Line-by-line with Manhattan is the best performing option on average for a desired aspect ratio of 0.4, whereas line-by-line wins for the remaining three test ratios.

Finally, take a look at Figure 5.9 to compare the max scale ratio of the four traversal methods relative to the standard approach in ELK Layered. As in Figure 5.4 the reference frame to fill is a square and the red horizontal line in the plots marks the value 1 on the $y$-axis. Every ratio greater than 1 indicates a drawing that can be scaled larger within the square reference frame using DisCo and the examined traversal method than the drawings the component graph group placer produces. As in the simple graph case all four traversal methods outperform the standard approach in this regard. On average, drawings achieved by the jitter traversal can be scaled the most, whereas Manhattan offers the least benefits.

Concluding this chapter, the data suggests a measurable improvement of DisCo over the component group graph placer in terms of minimizing the area of drawings. In terms of approximating an aspect ratio or using a reference frame to the fullest, the standard approach in ELK Layered performs poorly because it has not been designed with these aspects in mind. When it comes to external extensions, there is no clear traversal method recommendation at this point. Jitter and line-by-line performed best depending on the measure. This is why they should be tried on any specific graph. Manhattan and line-by-line with Manhattan on the other hand performed rather poorly. Figure 5.10 shows two drawings of one graph from the hierarchical graph set which DisCo with jitter traversal laid out particularly well in comparison to the component group graph placer. Both subfigures have areas of the same size and proportions to display the resulting drawings to demonstrate the difference with regard to the max scale ratio.

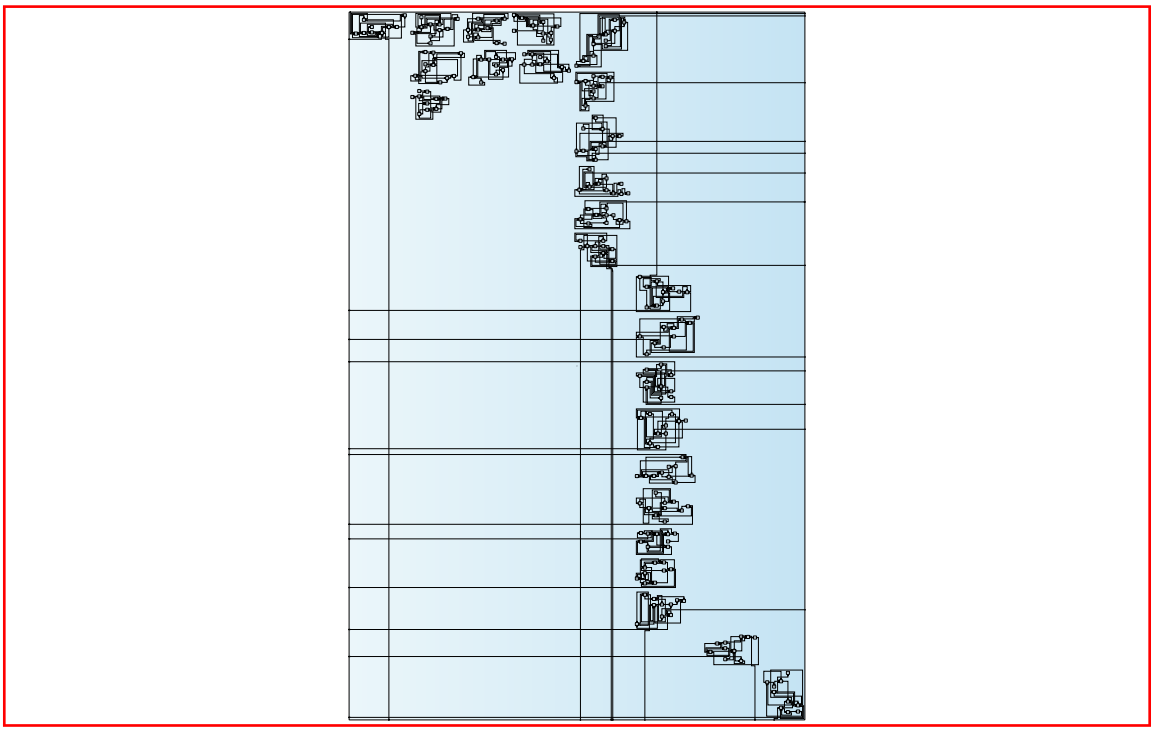



**(a)** Plots of the Max Scale Ratio comparing DisCo with the simple row graph placer

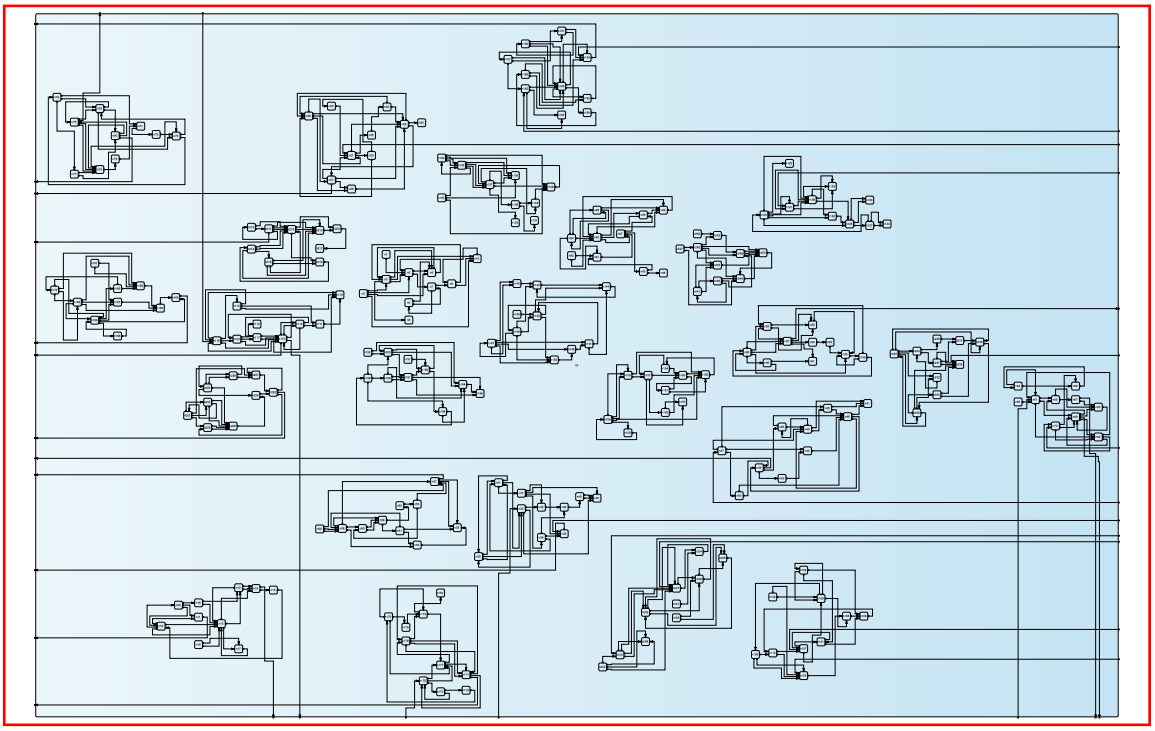
| Packing Method | Mean | Median | SD | Min | Max |
|---|---|---|---|---|---|
| **Line by Line** | 1.273485 | 1.261046 | 0.1903103 | 0.7373781 | 1.866779 |
| **Jitter** | 1.313741 | 1.309798 | 0.2015995 | 0.7514672 | 1.943827 |
| **Manhattan** | 1.164779 | 1.147585 | 0.180431 | 0.7514672 | 1.773072 |
| **Line by Line with Manhattan** | 1.165003 | 1.143812 | 0.1834989 | 0.7514672 | 1.773072 |

**(b)** Numerical data concerning the Max Scale Ratio of drawings

**Figure 5.9.** Do drawings with external extensions produced by DisCo adapt better to the reference frame of the canvas they are drawn upon? The comparison of max scale ratios shows that all four traversal methods adapt better to a given reference frame. Jitter traversal has the best average performance, whereas Manhattan traversal comes in last.



**(a)** Component group graph placer with a desired aspect ratio of 1.6



**(b)** DisCo with jitter traversal and a desired aspect ratio of 1.6

**Figure 5.10.** A last example graph from the hierarchical graph data set drawn in a fixed reference frame shown in red with an aspect ratio of 1.6

Chapter 6

# Conclusions

This chapter summarizes the contributions of this thesis and points out some ideas for future improvements.

## 6.1 Summary

This thesis examined an alternative approach for laying out connected components of graphs automatically in the Eclipse Layout Kernel (ELK). The approach is called Disconnected Components Compactor (DisCo) and is implemented as a standalone layout algorithm in ELK for now. In contrast to older approaches, such as strip packing, a component is not approximated by a rectangular bounding box but by a more fine-grained polyomino. This geometric figure is chosen to pursue the main goal of achieving layouts of minimal area.

Polyominoes are low resolution representations of components consisting of filled square cells on a Cartesian grid. Components translate into them by using a precomputed step size for rasterization. A grid of empty squares using this size is overlaid over each component and each grid cell intersecting with parts of the component is marked as filled.

The actual layout algorithm for compacting polyominoes works by placing them on a common integral base grid according to a cost function without overlaps between any two filled polyomino cells. After completing the compaction the base grid has to be cropped to the bounding box of the final layout as it is otherwise unbounded to all four sides. The layout of the polyomino packing is than translated back to a drawing consisting of the original components. A comparative evaluation with the standard approach implemented in ELK showed a slight improvement in minimizing the overall area of a drawing if the polyomino approach implemented in DisCo is used with a specific set of options. That is polyominoes are to be placed in decreasing order of the perimeters of their bounding boxes and the traversal method called jitter is used for enumerating candidate positions on the base grid according to the cost function $max(|x|, |y|)$.

Some minor contributions included a simple filling algorithm for polyominoes with holes to prevent big components with a lot of empty space within them to encompass small components, which improves the readability of the final drawing. A second adjustment was including an option to specify an aspect ratio for the drawing by allowing polyominoes

to consist not only of squares but of rectangles having dimensions in accordance with the desired aspect ratio. This approach worked really well compared to the standard one in ELK.

The major contribution of this thesis is the extension of the polyomino approach to hierarchical graphs. Short hierarchical edges are designed as indefinite beams, called external extensions, coming out of or going into polyominoes. They are represented by a third weakly filled polyomino cell state, allowing the indefinite beams to overlap with each other but not with filled polyomino cells. This third state allows edge crossings as not all graphs can be drawn planarly. Some other design changes were introduced to the algorithm to better accommodate hierarchical edges, including the order polyominoes are placed onto the base grid and restricting valid candidate positions on that grid based on the number and position of the extensions of each individual polyomino. The evaluation showed a great improvement in minimizing the area used by drawings produced with DisCo in relation to the existing component packer for hierarchical graphs in ELK Layered. Furthermore, DisCo approximates a target aspect ratio better than the existing algorithm.

However, one drawback of DisCo for hierarchical graphs is the circumstance that currently only a small subset of graphs is supported, i.e. the ones consisting of one parent node with a simple graph inside of them.

## 6.2 Future Work

A short compilation of potential areas of future research with respect to packing components with polyominoes concludes this thesis.

### 6.2.1 Integrating DisCo into ELK Layered

As mentioned in the last section DisCo works only on a specific subset of hierarchical graphs, i.e. one parent node with a simple graph inside of it whose nodes can connect to the surrounding node via short hierarchical edges. Section 4.1.2 mentioned that this was a conscious design decision and that a mechanism already present in ELK would make DisCo handle all hierarchical graphs without a lot of implementation effort. This mechanism is bottom-up layering which has been described in Section 2.2. The current implementation of DisCo works on the KGraph as a standalone layout algorithm, see Section 4.3. As of writing of this work, there is only one algorithm implemented in ELK that supports hierarchical edges and that is ELK Layered. Moreover, it works by using a bottom up layering, applying all of its five phases on each subgraph along the inclusion tree representing the hierarchy until it reaches the root node, see Definition 2.1.8.

ELK Layered works on its own internal graph representation called the *LGraph*. Therefore, the only steps necessary to make DisCo work with all hierarchical graphs is to implement a graph transformer as described in Section 4.3 that transforms an LGraph into a DCGraph and back and to integrate DisCo as an intermediate processor in between the edge routing phase of ELK Layered and the subsequent graph export.

### 6.2.2 Future Evaluation

When DisCo is finally able to lay out all kinds of hierarchical graphs, it can be tested on a broader range of diagrams. This thesis limited its evaluation to comparing areas, aspect ratios, and max scale ratios. But these represent only a small subset of aesthetic criteria for the drawing of graphs. For instance, another criterion called *crossing minimization* could assist in assessing the quality of drawings produced by the polyomino approach with external extensions. It is a measure counting all crossings between short hierarchal edges in this context. Minimizing the number of crossings in a drawing should lead to a more readable and thusly more aesthetically pleasing result.

In Chapter 5 the examination of the effect of using the filling algorithm from Section 4.2.1 did not lead to any final conclusions. This is why testing the filling algorithm on a different set of graphs is advised.

These graphs will not have to be necessarily randomly generated any longer if the integration of DisCo into ELK Layered is completed. Real world diagrams will become a viable source for testing, e.g. SCCharts. Applying DisCo to real problem domains can serve as the foundation of a user study to examine the readability of the drawings it produces. Furthermore, the importance of different graph measures, such as area, edge crossings or the distance between components can vary between problem domains. Szárnyas et al. published a paper on this very topic [SKS+16].

### 6.2.3 Improving Readability

DisCo currently makes diagrams more readable for humans by taking the underlying reference frame into account to closely fit into it, so that the elements of the diagrams can be displayed larger. But there is more to readability than size alone: alignment among elements of drawing can be important, too.

The current implementation of the polyomino approach does not produce alignment between different components relative to each other, which leads to a lack of straight lines in the final drawing for guiding the user's flow of reading. One possible adjustment can be made by changing how the cost function of the current implementation works. At the moment components are placed starting at the center. Subsequent components are placed around the already placed components spiraling outwards. This method was

chosen by Freivalds et al. for aiming at symmetrical layouts [FDK02], but it might hinder the creation of horizontal and vertical guiding lines. The authors also provide a different method for packing components onto fixed-size canvases starting in a corner of the the final drawing, meaning the underlying base grid would have natural instead of integral coordinates. Adapting this approach could improve alignment. However, placing components with hierarchical edges might be difficult using this approach, as the corner with the origin of the base grid would be preferably filled by a polyomino without external extensions by the current placement order in DisCo, rather than one with extensions leading in the direction of the corner.

A second approach to introducing alignment might deal with the positioning of the actual graph components within their polyominoes: there is some leeway for a component to be moved within a polyomino without crossing its boundaries, depending on the granularity of the resolution of the polyomino. Components could be moved within these boundaries to better align with each other in a post-processing step.

Two further ideas for improving readability concern the spacing between components. Right now, the spacing parameter between each component is a constant given in pixels, but depending on the shape of a polyomino and thus the shape of a component a variable spacing might be adequate. For instance, picture two polyominoes of rectangular shape. They might be placed close together without being mistaken for a single component because even a small spacing provides a straight dividing line between them. On the other hand, polyominoes with jagged edges might be less easily discernible, especially if two of them are placed interlocked with each other like well fitting puzzle pieces. A first heuristic for an algorithm deciding on suitable component spacings for each component might be based on computing its four profiles, similar to the filling algorithm introduced in this work, see Figure 4.10.

Finally, as an alternative approach to component spacing and the current filling algorithm at the same time, one could consider using different kind of hulls for filling the polyominoes. If a compact area of the final drawing is desirable, a rectilinear concave hull could be used, whereas filling polyominoes until they resemble rectangular bounding boxes would focus on separating the different components visually. Using a convex hull would be a compromise between a small area and readability.

### 6.2.4 Improving Performance and Packing

There has not been an extensive performance study of DisCo, yet. However as the enumeration of candidate positions during the placement of polyominoes so far always starts anew at the origin of the base grid, one should look into refinements for this procedure. Goehlsdorf et al. present two new ways, one is called *fast* and the other one *advanced* [GKS07]. Neither of them has been implemented in DisCo, yet.

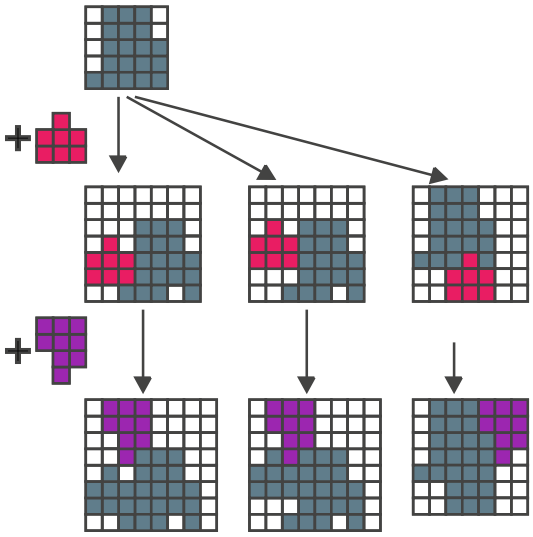


**Figure 6.1.** In a lot of cases the best solution for achieving a layout of minimal area is not apparent during one placement step. All placements of the pink polyomino have the same dimensions, if an aspect ratio of 1.0 is desired. Nevertheless, the third option is better than the first ones, as the next purple polyomino can be placed in a more space saving manner. The illustration is taken from Goehlsdorf et al. [GKS07].

Two suggestions concerning the minimization of the overall area occupied by a drawing conclude this thesis. The first one is a very simple change. So far, components have been rasterized so that they would be located in the center of the polyominoes representing them. This procedure ensures equal spacing between the edges of the component and the edges of the polyomino enclosing it, at least for each axis. If one does not deem this property important, the component might be simply put in a corner of the grid of its polyomino during the rasterization step. This might lead to fewer filled cells needed to cover the component and thus polyominoes of smaller size.

Finally, Goehlsdorf et al. propose an additional way to produce more compact drawings [GKS07]: as mentioned before, the algorithm for polyomino placing in a compact space uses a heuristic, as the underlying problem is NP-hard. One approach to reduce the risk of getting stuck in a local minimum is: when each new component is placed, try $k$ different candidate positions and save the results. In the next steps try $k$ different positions for each of these $k$ configurations and save the best $k$ ones. These best layouts should not only be picked by the smallest area but also provide for configurations as different as possible from each other. Figure 6.1 demonstrates the advantages of this procedure.

# Acronyms

***ELK*** Eclipse Layout Kernel

***ELK Layered*** Eclipse Layout Kernel Layered

***ELK Force*** Eclipse Layout Kernel Force

***UML*** Unified Modeling Language

***SCChart*** Sequentially Constructive Statechart

***DisCo*** Disconnected Components Compactor

***KIELER*** Kiel Integrated Environment for Layout Eclipse RichClient

# List of Figures

# Bibliography


[BJR80] Brenda S. Baker, Edward G. Coffman Jr., and Ronald L. Rivest. "Orthogonal packings in two dimensions". In: *SIAM J. Comput.* 9.4 (1980), pp. 846–855. DOI: 10.1137/0209064. URL: http://dx.doi.org/10.1137/0209064.

[CS93] E. G. Coffman and P. W. Shor. "Packings in two dimensions: asymptotic average-case analysis of algorithms". In: *Algorithmica* 9.3 (1993), pp. 253–277. ISSN: 1432-0541. DOI: 10.1007/BF01190899. URL: http://dx.doi.org/10.1007/BF01190899.

[DET+99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing: Algorithms for the visualization of graphs*. Prentice Hall, 1999. ISBN: 0-13-301615-3.

[Dog02] Ugur Dogrusoz. "Two-dimensional packing algorithms for layout of disconnected graphs". In: *Inf. Sci.* 143.1-4 (2002), pp. 147–158. DOI: 10.1016/S0020-0255(02)00183-4. URL: http://dx.doi.org/10.1016/S0020-0255(02)00183-4.

[Ead84] Peter Eades. "A Heuristic for Graph Drawing". In: *Congressus Numerantium* 42 (1984). Ed. by D. S. Meek and van G. H. J. Rees, pp. 149–160.

[FDK02] Karlis Freivalds, Ugur Dogrusoz, and Paulis Kikusts. "Disconnected graph layout and the polyomino packing approach". English. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 378–391. ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4_30.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph drawing by force-directed placement". In: *Software—Practice & Experience* 21.11 (1991), pp. 1129–1164. ISSN: 0038-0644. DOI: http://dx.doi.org/10.1002/spe.4380211102.

[FSM+10] Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden. "Automatic layout and structure-based editing of UML diagrams". In: *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED'10)*. Dresden, Mar. 2010.

[GH10] Emden R. Gansner and Yifan Hu. "Efficient, proximity-preserving node overlap removal". In: *Journal of Graph Algorithms and Applications* 14 (2010), pp. 53–74.

[GHN13] Emden R. Gansner, Yifan Hu, and Stephen C. North. "Interactive visualization of streaming text data with dynamic maps". In: *Journal of Graph Algorithms and Applications* 17.4 (2013), pp. 515–540. DOI: 10.7155/jgaa.00302.

[GKS07] Dennis Goehlsdorf, Michael Kaufmann, and Martin Siebenhaller. "Placing connected components of disconnected graphs". In: *6th International Asia-Pacific Symposium on Visualization, 2007*. Feb. 2007, pp. 101–108. DOI: 10.1109/APVIS.2007.329283.

[JGJ+80] Edward G. Coffman Jr., M. R. Garey, David S. Johnson, and Robert Endre Tarjan. "Performance bounds for level-oriented two-dimensional packing algorithms". In: *SIAM J. Comput.* 9.4 (1980), pp. 808–826. DOI: 10.1137/0209062. URL: http://dx.doi.org/10.1137/0209062.

[Len90] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN: 0-471-92838-0.

[NNB14] A. Nocaj, L. Nachmanson, and S. Bereg. "Node overlap removal by growing a tree". In: *EuroVis 2014: Eurographics/IEEE-VGTC Symposium on Visualization, Swansea, UK*. Poster and Extended Abstract. 2014. URL: http://algo.uni-konstanz.de/members/nocaj/publications/nnb-norgt-14/Node-Overlap-Removal-by-Growing-a-Tree.html.

[RAC+17] Ulf Rüegg, Marc Adolf, Michael Cyruk, Astrid Mariana Flohr, and Reinhard von Hanxleden. *Minimum-width graph layering revisited*. Technical Report 1701. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2017.

[RSG+16a] Ulf Rüegg, Christoph Daniel Schulze, Daniel Grevismühl, and Reinhard von Hanxleden. *Using one-dimensional compaction for smaller graph drawings*. Technical Report 1601. ISSN 2192-6247. Kiel University, Department of Computer Science, Apr. 2016.

[RSG+16b] Ulf Rüegg, Christoph Daniel Schulze, Daniel Grevismühl, and Reinhard von Hanxleden. "Using one-dimensional compaction for smaller graph drawings". In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS'16)*. 2016, pp. 212–218. DOI: 10.1007/978-3-319-42333-3_16.

[RSM+16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "Interactive model-based compilation continued – interactive incremental hardware synthesis for SCCharts". In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*.

Vol. 8802. LNCS. Corfu, Greece, Oct. 2016, pp. 443–462. DOI: 10.1007/978-3-662-45234-9.

[SFH09] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. *Automatic layout of data flow diagrams in KIELER and Ptolemy II*. Technical Report 0914. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009.

[SKS+16] Gábor Szárnyas, Zsolt Kővári, Ágnes Salánki, and Dániel Varró. "Towards the characterization of realistic models: evaluation of multidisciplinary graph metrics". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. MODELS '16. Saint-malo, France: ACM, 2016, pp. 87–94. ISBN: 978-1-4503-4321-3. DOI: 10.1145/2976767.2976786. URL: http://doi.acm.org/10.1145/2976767.2976786.

[SM91] Kozo Sugiyama and Kazuo Misue. "Visualization of structural information: automatic drawing of compound digraphs". In: *IEEE Transactions on Systems, Man and Cybernetics* 21.4 (July 1991), pp. 876–892. ISSN: 0018-9472. DOI: 10.1109/21.108304.

[SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing layered graphs with port constraints". In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (Feb. 1981), pp. 109–125.