Insa Marie-Ann Fuhrmann

Layout of Compound Graphs

Diploma Thesis
February 21, 2012

Christian-Albrechts-Universität zu Kiel
Department of Computer Science
Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden

Advised by
Prof. Dr. Reinhard von Hanxleden
Dipl.-Inf. Miro Spönemann

Abstract

Hierarchical graph structures are used to express abstraction and categorization in graphical modeling as well as in data visualization. The automatic layout of these structures brings key advantages to both application areas. Modelers and scientists benefit from tools that relieve them of the need of creating and maintaining diagrams manually. This thesis is concerned with the automatic layout of hierarchically structured graphs that allow hierarchy crossing edges, i.e. edges that connect nodes not contained by a common parent node. Hierarchically structured graphs comprising this kind of edges are referred to as *compound graphs*. These graphs raise special problems for automatic layout, since they cannot be placed by simple recursive application of a layout algorithm for flat graphs.

This explorative thesis presents a rather detailed synopsis of the literature on automatic drawing of compound graphs and a special subcategory, called *clustered graphs*. Furthermore it shows that a layer-based layout algorithm for flat graphs can be extended to enable it to place compound graphs in a single run without essential changes to the typical phases of a layered algorithm: cycle removal, layering, node ordering, node placement, and edge routing. The capacity of a given layer based algorithm is enhanced in this respect by a special graph import that creates a flat representation of a compound graph. Most of the specific problems that arise are solved with the help of pre- or postprocessing modules for the phases. The most prominent problems are cyclic dependencies of compound nodes and the treatment of subgraphs during the node ordering phase. The latter problem even demands an additional management module to handle the grouping of nodes presented to the crossing minimization.

This thesis demonstrates the feasibility of this concept. It also shows that the additional ability to draw hierarchy crossing edges is paid for with a considerable increase of computation time when compared to recursive application of the algorithm for flat graphs.

Eidesstattliche Erklärung

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

Kiel,

Contents

Abbreviations

CGA Clustered Graph Architecture
: Architecture for handling clustered graphs, introduced by Eades and Huang [14].

CGCM CompoundGraphLayerCrossingMinimizer
: A class in the node ordering phase of KLay Layered. Organizes the ordering of layer nodes for compound graphs.

CRG Crossing Reduction Graph
: Datastructure used in the crossing minimization phase for the layout of compound graphs.

GRIP Graph Drawing with Intelligent Placement
: Graph drawing method introduced by Gajer and Kobourov [25]

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client
: A research project on the pragmatics of graphical modeling.

KIML KIELER Infrastructure for Meta Layout
: KIELER subproject for abstract layout specification for diagrams.

KLay KIELER Layouters
: Project of KIELER for layout algorithms.

RCB Right Compound Border
: A dummy node in the representation of a compound node used by KLay Layered. Marks the right border of a compound node.

RCP Right Compound Port
: A dummy node in the representation of a compound node used by KLay Layered. Represents a port of the original graph that will be drawn on the right side of the compound node.

PATIKA Pathway Analysis Tool for the Integration and Knowledge Acquisition
: A software tool vor the visualization and manipulation of cellular events developed by Dogrusoz et. al [7].

LCB Left Compound Border
: A dummy node in the representation of a compound node used by KLay Layered. Marks the left border of a compound node.

LCP Left Compound Port
: A dummy node in the representation of a compound node used by KLay Layered. Represents a port of the original graph that will be drawn on the left side of the compound node.

SOP Subgraph Ordering Processor
: Intermediate Processor of the KLay Layered algorithm to postprocess the node ordering phase.

1

Introduction

▶ 1.1 Hierarchical Graph Structures and Automatic Layout

Abstraction is one of the main concepts of computer science. One means to express it in graphical modeling are containment relationships in models. Diagrams of hierarchical graph structures are appropriate to present those models. In hierarchical graph structures nodes may contain other nodes and edges. An example for hierarchical structures in a UML Statechart used to model a phonecall is shown in Figure 1.1.

If there are edges that connect nodes situated in different parent nodes, called *hierarchy crossing edges*, we speak of *compound graphs*, in contrast to *nested graphs*. Compound graphs restricted to edges that connect nodes containing no nodes themselves (*leaf nodes*) are called *clustered graphs*. Some graphical modeling languages are restricted in a way that they correspond to nested graphs. This is true for example for SyncCharts, which forbid edges that cross the boundaries of a node. Others, for example UML Statecharts, allow hierarchy crossing edges. In case this edge type is used, automatical compound graph drawing is called for.

Apart from graphical modeling, hierarchical graph structures are used in the field of data visualization. Especially in the realm of bioinformatics there is a strong need to present collected data in a way that is not only clearly arranged but displays data characteristics. This usually involves the grouping of elements that belong to the same categories, are strongly related, or share common properties. Clustered graphs are especially apt to display this kind of abstraction. Figure 1.2 shows an example of a clustered graph from the field of bioinformatics. It is used to visualize information about biological pathways, which are series of molecular reactions in a cell.

Hierarchical structures in diagrams may help to make research findings obvious and improve the understanding of diagrams in graphical modeling for the advanced user, as Cruz-Lemus et al. [5, 6] state. However, creating and maintaining diagrams of hierarchical graph structures by hand is even more effort prone than for flat graph structures. Adding a node for example often means that the developer has to resize a number of nodes, as the new node has to be spatially included by the node that

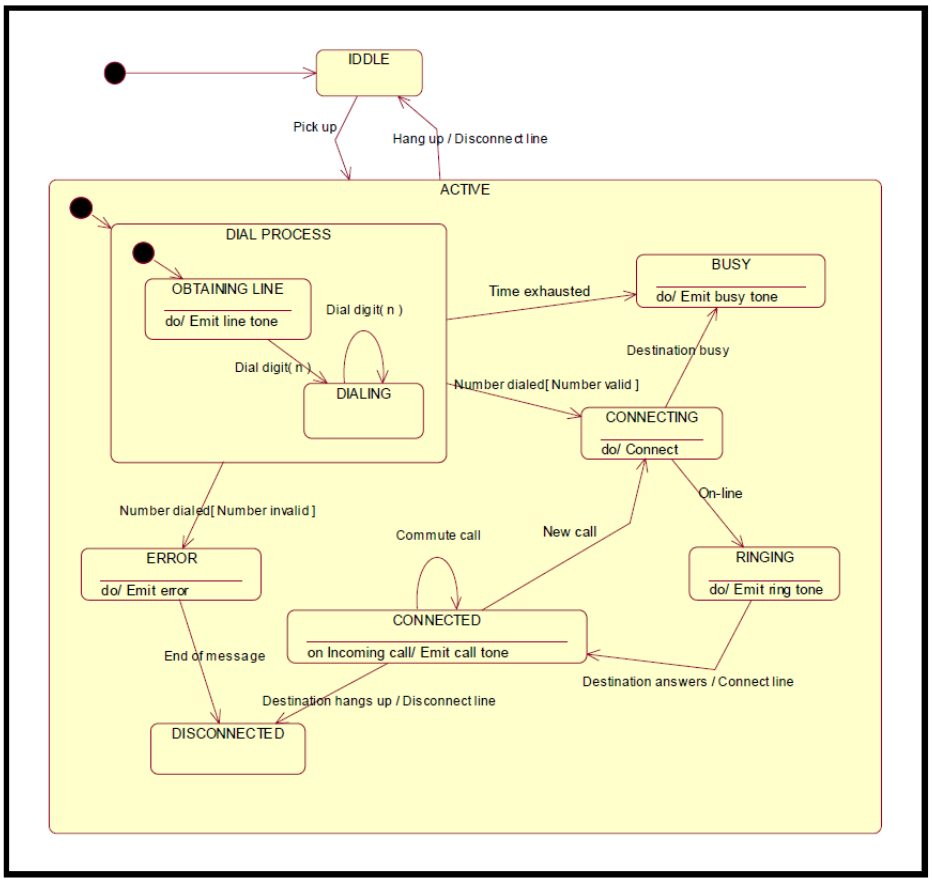


Figure 1.1. Example of an UML Statechart with hierarchical structures. Figure by [5].

contains it and making room involves all nodes in the sequence given by the chain of inclusion relations.

As Fuhrmann [24] states, a key enabler for improved handling of graphical models is the automatic layout of the diagrams—and this is especially true when hierarchical structures are involved. However the benefit of automatic layout is as strong for data visualization, since under the use of a layout algorithm, data can subsequently be added and displayed without effort to the scientist. Thus, at any time, the current state of work can be visualized. This requires only an update of one main diagram instead of the creation of a new drawing done by hand.

The advantages for both application areas present a motivation to offer an automatic drawing algorithm that can place clustered and even compound graphs.

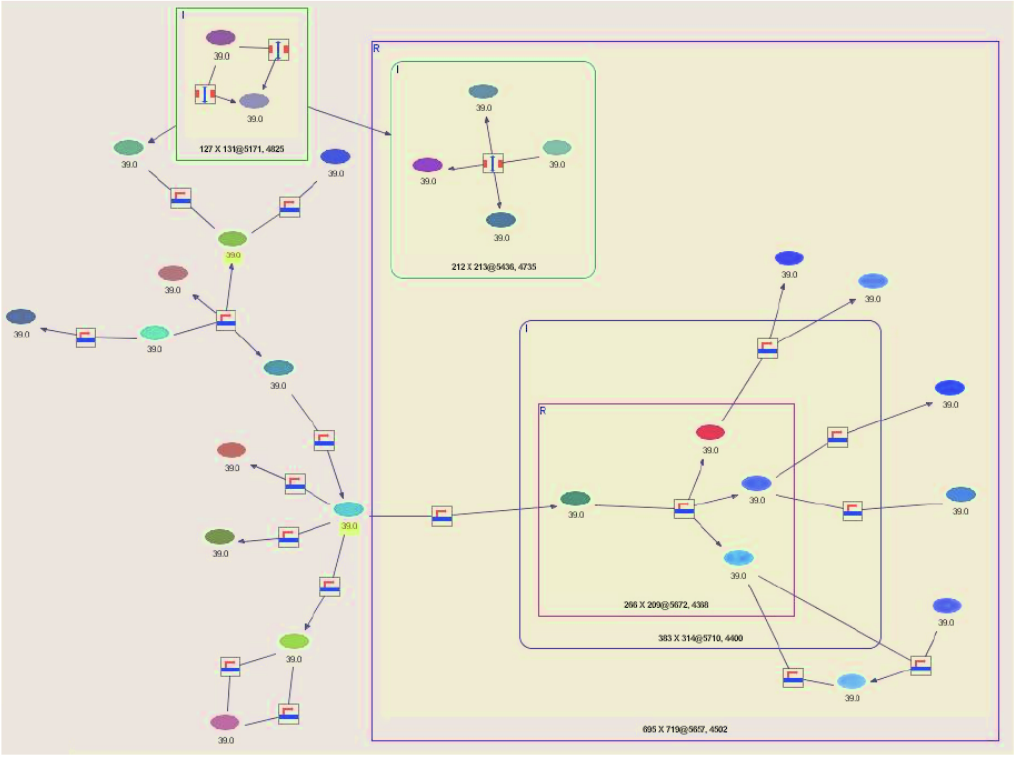


Figure 1.2. Example of a clustered graph used in the visualization of data on biological pathways (molecular cell reaction series). Figure by [7].

▶ 1.2 Drawing Conventions and Rules for Compound Graph Diagrams

For our own implementation we follow the conventions given below for the drawing of a hierarchical graph structures. These conventions are similar to those adopted by Sugiyama and Misue [43]:

1. Inclusion relations are denoted by full geometrical inclusion of the shape corresponding to the contained node in the shape of the parent node.

2. Nodes are laid out hierarchically in a way that all edges can be drawn as arrows pointing in the layout direction. A small amount of edges may point in the other direction, if the graph is cyclic.

Additional drawing rules are:

1. Nodes without inclusion relations never overlap.

2. The number of crossings between adjacency edges is reduced as much as possible.

3. Nodes connected by adjacency edges should be placed as close to each other as possible.
4. The number of crossings between adjacency edges and node shapes should be reduced as much as possible.
5. The number of edge bends should be reduced as much as possible.
6. The size of nodes containing other nodes should be adapted to fit the content.
7. The drawing should be compact.

The first rule is imperative to guarantee the readability of the diagram with respect to containment relations. The other rules are presented in the order of their importance for readability based on the study of Purchase [33] and the catalog by Sugiyama and Misue [43]. To my knowledge, Rules six and seven have not been subject to a study of their influence on human perception, but have been considered here as they affect the presentability of the final drawing in print or on a computer screen.

For the presentation of diagrams in this work, we represent nodes as follows.

- Nodes that do not contain other nodes are drawn as round shapes or—in case they own ports—as rectangular shapes with horizontal and vertical sides.
- Nodes that contain other nodes are drawn as rectangular shapes with horizontal and vertical sides, edges may be rounded.

▶ 1.3 Contributions of this Thesis

This thesis is an explorative work and situated in the KIELER Layouters (KLay) project that is part of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project. It offers a rather detailed synopsis of the literature on automatic drawing of clustered and compound graphs.

Furthermore it broadens the capability of the layer based layout algorithm KLay Layered to the drawing of compound graphs, which includes the layout of clustered graphs as well. The layer-based layout scheme aims to assign nodes to horizontal or vertical layers in a way that in general the edges point in the same direction, called layout direction. Beforehand Klay Layered did support the drawing of nested graphs by recursive application of the layout algorithm to the subgraphs. However this approach was not sufficient to draw graphs with hierarchy crossing edges.

This work demonstrates the concept of enabling a layer based layout algorithm for flat graphs to draw compound graphs without materially changing the main parts of the algorithm like cycle breaking heuristic, layering algorithm, crossing minimization heuristic, node placement, and edge routing.

I take advantage of the special structure of the KLay layered algorithm, which keeps each phase of the algorithm modular and allows to introduce adjustments by

pre- and postprocessing of the single phases as well as a graph import and export interface, see Schulze [41]. This work shows that this concept is flexible and general enough to support compound graph layout as well.

To enable KLay Layered to draw a compound graph in a single run of the whole algorithm, this work provides:

- A special graph import that translates the compound graph to a flat graph representation. This import supports the KGraph format of KIELER. The general idea follows Sander [38], but extends it with respect to port handling.
- A preprocessing intermediate processor before the cycle breaking phase to handle cyclic dependencies of compound nodes, special port-edge constellations, and insertion of dummy edges to support correct layering.
- A postprocessing intermediate processor after the layering phase to remove dummy edges used for the layering.
- A management class in the crossing minimization phase that influences the order of nodes processed and is able to group nodes according to their inclusion relations without affecting the layer sweep iteration and the crossing minimization of the algorithm. The implementation is inspired by an approach of Forster [20].
- A postprocessing intermediate processor after the node ordering phase to correct possible intertwining of subgraphs following a suggestion of Sander [38].
- A preprocessing intermediate processor before the node placement phase to insert dummy nodes and edges to reserve drawing space for upper and lower compound node rectangle sides based on Sander's method [38].
- A postprocessing intermediate processor after the edge routing phase to remove all dummy nodes and edges inserted for compound graph drawing and prepare the actual layout application.
- A special layout application respecting the more complex demands on the position calculation of hierarchy crossing edges and nested nodes. This application supports the KGraph format of KIELER.

The approach involves port handling, even when the graph contains portless edges as well as port edges or edges connected to a port only on one side. The placement of compound node ports (hierarchical ports) is supported by the introduction of special dummy nodes. Furthermore the implementation handles edges between a containing node and one of its content nodes.

▶ 1.4 What Is Not Covered by this Thesis

However the general problem was restricted in some aspects due to its general complexity. The approach does not yet support north-south ports, ports situated on

the top or bottom side of a node, and port constraints for compound nodes, special treatment for self loops of compound nodes, and parallel areas in a node to represent concurrency. Seperation of connected components and layout directions have also not been in the focus of this work. As label placement in KLay Layered is currently a subject to another work in the KIELER project, it has not been considered in this thesis.

▶ 1.5 Outline

Chapter 2 provides definitions referred to throughout the thesis. Chapter 3 presents a compendium of the related work. First, we discuss papers on automatic drawing of clustered graphs to move on to approaches for the layout of compound graphs. Works presenting a summary of algorithms are introduced afterwards and the chapter concludes with a paper less closely related to the topic, but offering interesting parallels. In Chapter 4 we briefly introduce KIELER, KLay and KLay Layered, since they offer the development environment for the implementational part of this thesis. The choices of algorithmic concepts are discussed in Chapter 5, following the order in which the problems arise during a run of the KLay Layered algorithm. The implementation is evaluated in Chapter 6, followed by the conclusion in Chapter 7, which gives a short summary and points to future work.

2

Definitions

Graph A *graph* is a pair $G = (V, E)$ where V is a set of *nodes* and and E is a set of *edges* $e \in \{\{u, v\} \mid u \in V, v \in V\}$. We call a set of sets of nodes $P = \{V_1, \ldots, V_n\}$ with $\bigcup_{i=1,\ldots,n} V_i = V$ and $V_i \cap V_j = \emptyset \ \forall i, j \in \mathbb{N}_{\leq n}$ a *(n-way) partition* of the graph $G = (V, E)$.

For a partition $P = \{V_1, \ldots, V_n\}$ of graph $G = (V, E)$ we define the *quotient graph* $Q = (V_Q, E_Q)$ with $V_Q = P$ and $(V_i, V_j) \in E_Q$ if $i \neq j \wedge \exists u \in V_i$ and $v \in V_j$ such that $u, v \in E$.

Drawing *Drawing* a graph $G = (V, E)$ on a surface S means to represent G on S such that points of S are associated to nodes and a curve in S is associated to each edge $e = u, v$ such that the endpoints of the curve correspond to u and v. The circular, clockwise order of edges in connection with one node determined by the drawing of the graph is called the *embedding*. A graph is called *planar* if it can be drawn in the plane without any edge crossings. In a graph drawn like that, regions bounded by edges are called *faces*. The infinite region surrounding the graph is called the *outer face*. A drawing is called *orthogonal* if the edge curves consist of a set of horizontal and vertical segments.

Directed Graph In a *directed graph* $G = (V, E)$ the edges additionally hold the notion of direction: $E \subseteq V \times V$. An alternative notation for $e = (u, v) \in E$ is $u \rightarrow_E v$. We refer to u as the *source* and to v as the *target* of the edge $e = (u, v)$. We call e an *outgoing edge* of u, while it is an *incoming edge* of v. We define the *indegree* of a node $v \in V$ ($indeg(v)$) as $|\{(u, v) \in E : u \in V\}|$ and its *outdegree* ($outdeg(v)$) as $|\{(v, u) \in E : u \in V\}|$. Nodes v with $indeg(v) = 0$ are called *sinks*, while we refer to those with $outdeg(v) = 0$ as *sources*.

If $e = \{v, v\}$ or respectively $e = (v, v)$, the edge is called a *self loop* or *directed self loop*.

Connectivity A sequence of nodes $v_0, \ldots, v_n$ is called a *path* in a set of edges E if $\{v_i, v_{i+1}\} \in E$ for all $i \in 0, \ldots, n-1$, or a *directed path* if $(v_i, v_{i+1}) \in E$. For a path in E we use the short denotation $v_1 \rightarrow_E^* v_n$.

We identify a (directed) graph as *connected*, if there exists a (directed) path between any two nodes. We call it *k-way connected* if there is no set of $k-1$ nodes, whose removal would render the graph disconnected. If at least k edges would have to be removed to disconnect the graph it is *k-edge connected*.

Tree A *tree* is an undirected graph that is connected and contains no cycles. A *directed tree* is a directed graph with no cycles and $indeg(v) = 1$ holding true for each $v \in V$ but one single node, called the *root*, which is connected to any other node in the tree by a path and whose indegree is 0. Nodes v of a directed tree with $outdeg(v) = 0$ are called *leafs*. A *branch* of a directed tree T is defined by one of the nodes of T, which is the root of the branch. The branch contains all nodes which can be reached by a directed path from the root of the branch and all edges whose source and target are within this node set.

Weighted Graph A *weighted* graph $G_w = (V, E, w)$ comprises sets of vertices V and edges E and the edge weight function $w : E \rightarrow \mathbb{R}$. For a weighted graph, define the *weighted quotient graph* $Q = (V_Q, E_Q, w_Q)$ with V_Q and E_Q as defined for non-weighted graphs and $w_Q : E_Q \rightarrow \mathbb{R}$ and $\forall e = (V_i, V_j) \in E_Q$:

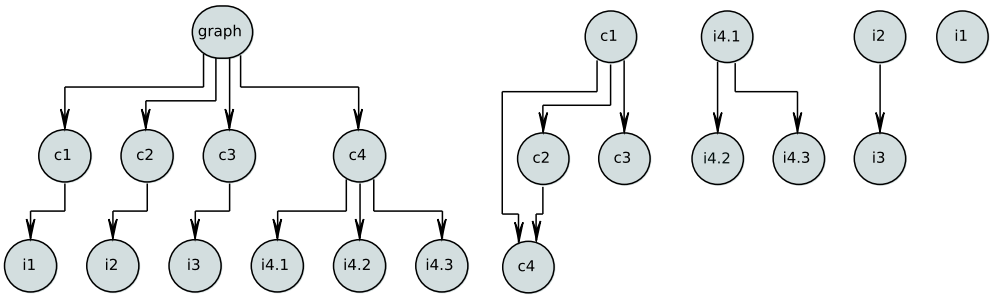
$$w_Q(e) = \frac{\sum_{u \in V_i, v \in V_j, (u,v) \in E} w((u,v))}{|\{(u,v) \in E : u \in V_i, v \in V_j\}|}$$

.

Compound Graph A *(directed) compound graph* is a triple $C = (V, E, I)$ with the pair (V, E) forming a (directed) graph, called the *adjacency graph*, and the pair (V, I) forming a directed graph without cycles. In this thesis we will restrict our notion of a (directed) compound graph to triples $C = (V, E, I)$, where the pair (V, I) is a directed tree, to which we refer as the *inclusion tree*. Notice that we regard the graph itself as the root node, though it is not represented as a node in the data structure of the implementation.

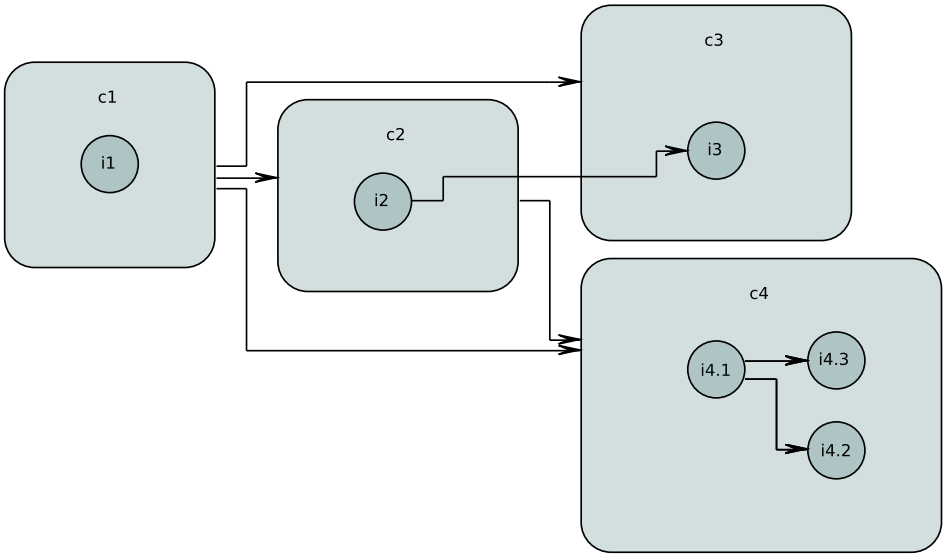
Figure 2.1 illustrates the relationship of the compound graph to its inclusion tree and adjacency graph. We display an example of a compound graph in 2.1(c) in the way it is actually laid out by the KIELER tool, see Chapter 4, and the corresponding adjacency graph and inclusion tree.

The edges $e = \{i, j\} \in E$ respectively $e = (i, j) \in E$ of a (directed) compound graph express adjacency relations and are referred to as *adjacency edges*. In the case of a directed edge $e = (i, j)$ we call the node i the *predecessor* of j and j the *successor* of i. We denote the set of predecessors for a node v as *pre(v)*, the set of its successors as *suc(v)*. An edge $i = (u, v) \in I$ represents a nesting relation meaning that u contains v. Therefore we call it an *inclusion edge*.



(a) An inclusion tree $T = (V, I)$.

(b) An adjacency graph $G = (V, E)$.



(c) The compound graph $G = (V, E, I)$.

Figure 2.1. A compound graph and its inclusion tree and adjacency graph.

Figure 2.1(c) pictures inclusion relations by drawing compound nodes as (slightly rounded) rectangular shapes, here labeled c1 to c4. Leaf nodes are drawn as round shapes, and the children of a node are drawn within the rectangular boundings of its shape.

We name the source of an inclusion edge to be the *parent* of the edge's target and the target to be the *child* of the source. We write $child(v)$ for the set of children of a node v and $parent(v)$ for the parent of v. Nodes that share the same parent are called *siblings*. We call a node without children a *leaf node* and other nodes *compound nodes*. The *depth* of a node v is the number of edges in $root \rightarrow_I^* v$ and we denote it as *dep(v)*. The term *descendants* of a node v means the set of nodes u connected to v by a nonempty path of inclusion edges $v \rightarrow_I^* u$ and can be written as $des(v)$. We denominate the set $anc(v) = \{u \in V \mid \exists\, u \rightarrow_I^* v, u \neq v\}$ as the *ancestors* of a node v. We say that descendants of a node v *belong* to v and that v *contains* its

descendants. We call adjacency edges that connect a node with one of its descendants *descendant edges* and an adjacency edge whose source and target do not have the same parent a *hierarchy crossing edge*. An example for a hierarchy-crossing edge is the edge from i2 to i3 in Figure 2.1. If there is a hierarchy crossing edge between a descendant of one compound node and a descendant of another compound node or this other compound node itself, we say that the compound nodes are *connected by a hierarchy crossing edge*, no matter which direction the edge has. We say that an edge *belongs* to a compound node, if it is a descendant edge of a child node or if the compound node is parent of the target of the edge, no matter if it is parent of its source as well. The *induced subgraph of compound node c* is a compound node, all nodes belonging to it and edges whose target is a node in this set. In Chapter 5, the term subgraph will refer to all nodes used to represent these graph elements. This includes dummy nodes.

We say that two compound nodes u and v of a compound directed graph $C = (V, E, I)$ have a *cyclic dependency*, if there are paths $i \rightarrow_E^* j$ and $x \rightarrow_E^* y$ with $(i, y \in \{u\} \bigcup des(u)) \ \wedge \ (j, x \in \{v\} \bigcup des(v))$.

A compound node with maximal depth of all compound nodes in a (sub-)graph is an *innermost* compound node, a compound node with minimal depth is an *outermost* compound node.

Clustered Graph A *clustered graph* is a compound graph, in which adjacency edges are only allowed between the leafs of the inclusion tree.

Ports If a directed graph has an attributive finite set P of *ports*, we call it a *port containing graph* and denote the subset of ports belonging to a node v with $P(v)$. Any edge of the port based graph can have a specified *source port* and *target port*. We will refer to edges that have a source port as *source port edges* and accordingly speak of *target port edges* in the case the edge owns a target port. The cut-set of the set of source port edges and the set of target port edges is called the set of *port edges*. Edges contained in none of the two sets are referred to as *portless edges*. We call graphs that own only port edges *port based graphs*.

3

Related Work

As this thesis is an explorative work, one of its contributions is to provide a more thorough view on the related work than would usually be given. A subproblem of the automatic layout of compound graphs is the drawing of clustered graphs, which is introduced in Section 3.1. The literature directly concerned with automatic drawing of compound graphs is presented in Section 3.2. Two main methods for layout algorithms that we will frequently deal with in the following are the *layered approach*, first introduced by Sugiyama *et al.* [45], and the *force-directed approach*, which was first proposed by Tutte [46]. However, perhaps the most referenced approach is that of Eades and his spring embedder model [9]. The first is a method developed for directed graphs and is based on sectioning the drawing plane into layers, into which the nodes are distributed such that all edges point in one direction. Since this is not directly possible for cyclic graphs, the methods usually comprise cycle breaking heuristics as well. For algorithms of the force-directed approach, the graph is associated with a physical model, for example the nodes are modeled as steel rings and the edges as springs connecting them. Physical calculation models developed to compute equlibria in such systems are used to compute a positioning of the nodes leading to a balanced state of forces in the model which is expected to correspond with an appealing drawing. Additional works of synoptical character are presented in Section 3.3. The Section 3.4 presents work on a distantly related topic—not directly important for this thesis, but offering interesting ideas. Section 3.5 summarizes the presented work.

▶ 3.1 Part of the Problem: Drawing Clustered Graphs

Drawing clustered graphs can be seen as a subproblem of drawing undirected compound graphs. If the clusters are to be displayed as shapes that surround the contained nodes, the problem is identical to drawing undirected compound graphs with the restriction that adjacency edges may exist between leaf nodes only. Due to this restriction, the approaches are no absolute candidates for the implementation part of this thesis. However, it is rewarding to consider the general ideas.

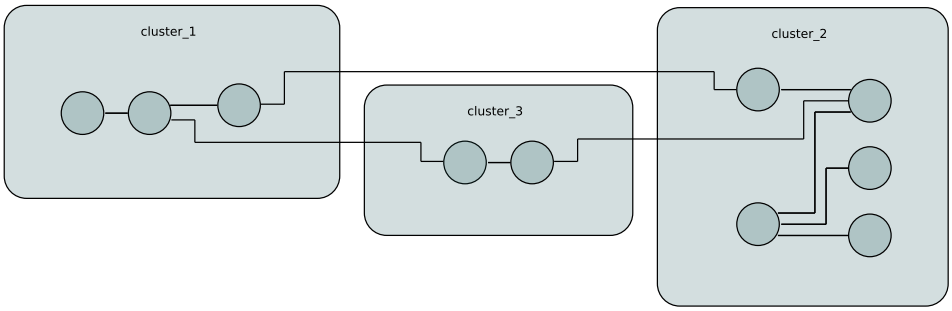


Figure 3.1. A clustered graph with clusters shown as rectangles surrounding the nodes of the cluster.

Figure 3.1 shows a drawing of a clustered graph in which the clusters are pictured as rectangles surrounding the nodes of one cluster.

Clustered graphs are usually defined as being undirected—which is consistent with the customary use of this datastructure in practice, especially in displaying of biological structures. Therefore, naturally the layering approach of graph layout is of no importance to this field of graph drawing. The literature is focussed on one hand on force directed graph drawing methods, which are well apt to mirror organic structures in graph layout and for this reason are widely favored in the field of biological practice. These approaches are introduced in section 3.1.1. On the other hand there are a lot of other approaches to the solution of this problem, a considerabley wider range of different ideas than in the field of compound graph drawing. Some of the methods are introduced in 3.1.2. Some methods are not only concerned with drawing a clustered graph, but also with the clustering itself. In this case, only the nodes are given, but not the inclusion tree structure of the clustered graph. These problems are often related to clustered graphs with only one hierarchy level, which means that clusters are to contain only nodes, not other clusters. Some of the clustering heuristics are given a fixed number of clusters and search for a node assignment that is optimal with regard to a certain criterion, as for example a minimal number of edges between nodes of different clusters. Some heuristics respect a certain criterion, like k-way connectivity of clusters, and try to reach a minimal number of clusters while holding the criterion.

▶ 3.1.1 Clustered Graphs in the Force Directed Approach

Since clustered graphs are usually defined as undirected, and in practice they are often used to illustrate biological structures, the force-directed method for drawing them is a quite obvious choice, for the display of organic structures and their symmetries are an advantage of force-directed methods. Several variants and employments of force-directed methods for clustered graphs have been presented in the literature.

Wang and Miyamoto [47] introduce the graph layout tool *LYCA*. It provides three features.

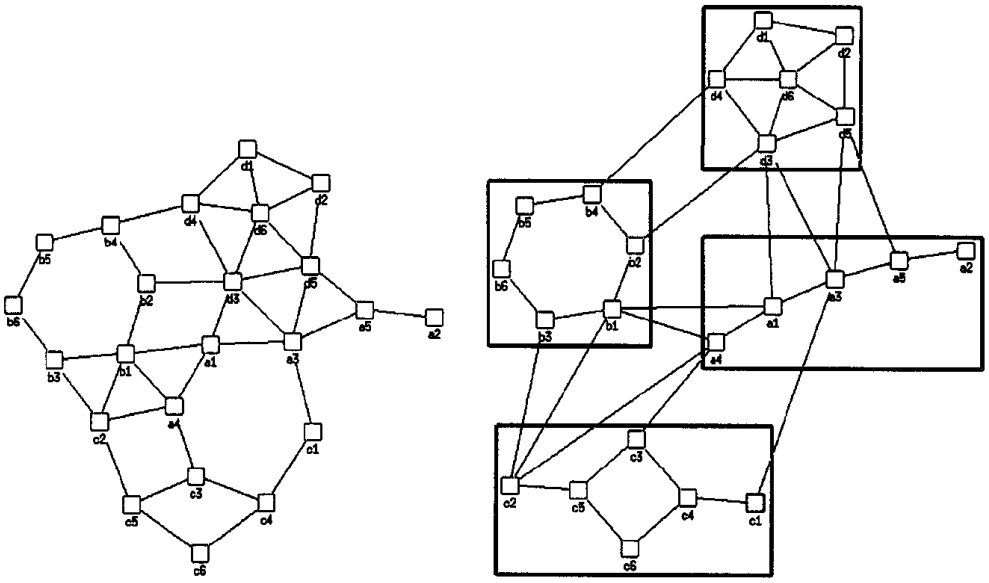


Figure 3.2. Visual structuring of layouts by the LYCA tool of Wang and Miyamoto [47]. Figure by [47].

1. First they suggest an improvement of the force directed layout algorithm by Fruchterman [23] to handle graphs with vertices of different sizes.

2. Then they present a divide-and-conquer approach to generate visually structured layouts.

3. In the last part, a constraint solver is introduced.

Figure 3.2 shows the structural layout done by the LYCA tool. On the left a drawing is shown, in which the structure is not shown visually, on the right there is the layout result of the LYCA tool with a visually obvious cluster presentation.

To adjust the force-directed algorithm to the demands of drawing graphs with different node sizes, the forces between a pair of vertices v and w are modified to be

$$F_a = \begin{cases} 0 & \text{if w, v overlap,} \\ \frac{d_{out}^2}{k'+d_{in}} & \text{otherwise,} \end{cases}$$

where F_a is the attractive force, d the distance between the node centers, d_{out} is the distance between the node boundaries and $d_{in} = d - d_{out}$, while k' is the optimal distance between vertices, and

$$F_r = \begin{cases} C\frac{k'^2}{d} & \text{if w, v overlap,} \\ \frac{k'^2}{d} & \text{otherwise.} \end{cases}$$

Here F_r is the repulsive force between v and w and C a constant. It follows that an equilibrium of repulsive and attractive force exists, when d_{out} is equal to k'.

The authors adjust the force model further to enable a divide-and-conquer mechanism to draw visually structured drawings of graphs with subgraphs[1]. They divide the edges into *intra-edges*, whose nodes are of the same subgraph, and *inter-edges*, whose nodes are not. Furthermore, the notion of a *meta-graph* is introduced, which is obtained by collapsing subgraphs into *meta-vertices* and transforming inter-edges to *meta-edges*. To calculate the position of a vertex, a composite force is calculated:

$$F_{comp} = F_{intra} + S(t)F_{inter} + (1 - S(t))F_{meta}$$

where F_{intra} and F_{inter} are the intra- and inter-forces exerted on a vertex respectively and F_{meta} is the meta force on a vertex. $S(t) \in [0, 1]$ is a function of layout time t, here relating to the iterations of the algorithm, such that $S(t)$ decreases as t increases after a threshold t' and reaches 0 at another threshold $t'' > t'$. Such, the layout traverses three phases:

Between time 0 and time t' only the inter- and intra-forces are at work. A layout with uniform edges and a small number of edge crossings is produced.

Between time t' and t'' the strength of the inter-forces decrease in favour of the meta-force.

At t'' only intra- and meta-forces are at work. Zones of subgraphs get displayed.

In the final part, the authors describe how LYCA integrates a constraint solver with the layout algorithm, which can handle the following constraints:

- an *absolute constraint*, which fixes a vertex at its current position,
- a *relative constraint*, which constrains its position in relation to others and
- a *cluster constraint*, which enables several vertices to be processed as a whole.

The constraint solver is able to handle situations in which constraints turn a node into a barrier for the movement of another by introducing a *rigid stick* between them, allowing them only to be moved as a whole. If a node is constrained as in the center of a set of vertices, the force on it is evenly distributed between them.
The interaction with the layout algorithm is organized in four steps:

Step 1: Calculate forces. In charge: Layout algorithm.

Step 2: Introduce sticks and distribute forces. In charge: Constraint solver.

Step 3: Calculate new positions. Layout algorithm.

[1] Note that the authors give examples of graphs only in the sense of clustered graphs with nesting depth 2.

Step 4: Satisfy constraints. Constraint solver.

Note that LYCA is reduced in its performance by the fact that the divide-and-conquer method draws a graph twice to create the structured layout. The layout quality is dependant on the number of constraints. LYCA performs well if either the constraint solver or the layout algorithm are dominant, i.e. if many or few constraints are defined. In a balanced case, as the authors state, a poor layout may result.

Eades and Huang [14, 26] introduce a tool for the visualization of large undirected clustered graphs named *DA-TU*[2] which also uses force-directed methods. The authors' presentation of the underlying ideas is split into three parts.

1. Introduction of the layered architecture Clustered Graph Architecture (CGA),
2. Description of the force-directed graph drawing method and
3. Introduction of animation methods.

The layered architecture CGA is intended to support the display of *abridgements* of clustered graphs, which are logical views of parts of the graph. The layers are distinguished as follows:

Graph layer: A graph in CGA is an undirected graph, consisting of nodes and edges. It may be large and dynamic, which means that node and edge set may be changing, for example by user interaction. Nodes and edges may have application-specific attributes, such as labels and semantics. Basic operations attributed to this layer are adding and removing nodes and edges from the graph, the changing of attributes and an operation that returns the list of neighbours for a node u.

Clustering layer: A clustered graph consists of an undirected graph and a *cluster tree* describing an inclusion relation between clusters. Operations on the level of this layer are the creation and the destroying of a cluster.

Abridgement layer: The fact that the clustered graph might be too large to be shown on the screen and comprehended by the user motivates the introduction of an abridgement of the graph. An abridgement is characterized by a set of nodes of the graph. It consists of all nodes and edges on paths between nodes of this set and the root. Additionally, there is an edge between any two nodes u and v of the characterizing set if and only if there is an edge in the graph between a descendant of u and a descendant of v. There are three elementary operations on abridgements. The first is the opening of a cluster, which replaces a node u that is in the characterizing node set of the abridgement by its children. The reverse operation is called closing of the cluster. The third operation is the hiding of a node, which deletes the node from the basic node set.

[2] "Big map" in Mandarin.

PICTURE LAYER: The picture layer is dedicated to the actual display of abridgements of large graphs. A static picture of the abridgement contains locations for each vertex, a route for each edge and regions of the plane for each cluster, the architecture here allowing overlapping regions. Dynamic pictures are sequences of static pictures used to animate changes in the drawing for to preserve the user's mental map, see Eades et al. [15] on the concept of mental maps. Operations associated to this layer are the manual movement of nodes, the gathering, which moves sibling nodes closer together, an operation used to make the cluster boundaries disjoint in the picture. A scaling operation can be used to increase or decrease the size of a picture. The layout operation is used to update the picture after changes to the graph.

The force directed algorithm employs a force model with three types of spring forces: internal springs between a pair of nodes which are siblings in the clustering tree, external springs between nodes which are not, and virtual springs. The latter connect each node of a cluster with a virtual node representing the cluster to keep its nodes together. Furthermore there are gravitational repulsion forces between all nodes. The movement of nodes is to be stopped when a locally minimal energy state is reached.

In the visualization tool DA-TU, of which a screen is shown in Figure 3.3, the changes are animated at the picture layer. If a new layout is calculated by the force algorithm, the movements are shown by a series of drawings. Additionally, when a node is deleted, it is animated to lose its edges first, then shrink until it disappears. An added node gains its edges first, then grows in its place to the full size. Rectangular regions of clusters are dynamically adapted to the minimum enclosing rectangle of the cluster's children. Rectangles move smoothly with the children, especially in the gathering operation. Furthermore, scaling operations are animated as well as cluster closing and opening, which lets the rectangular area of a cluster shrink/grow to a certain threshold before replacing the closed by the opened form or vice versa. The *camera* animations move the whole drawing to move specific nodes to the center of the screen.

It can be observed that the example sessions given in the paper show that the automatic layout often produces cluster overlaps which have to be removed by applying the gathering operation almost after every session step.

Frishman and Tal [22] present an algorithm for the force-directed incremental drawing of clustered graphs. Noticeably, the term of a clustered graph in this paper allows edges between the clusters themselves. however, the examples and evaluations in the paper only refer to clustered graphs where clusters only contain leaf nodes.

As most force-directed algorithms tend to produce strong changes in layouts for the same graph after a small alteration of the node or edge set, the authors aim to find an alteration of the standard force-directed methods that makes them more apt to preserve the user's mental map. The authors state that the key issue is the stability of the layout. They especially strive to minimize the movements of clusters

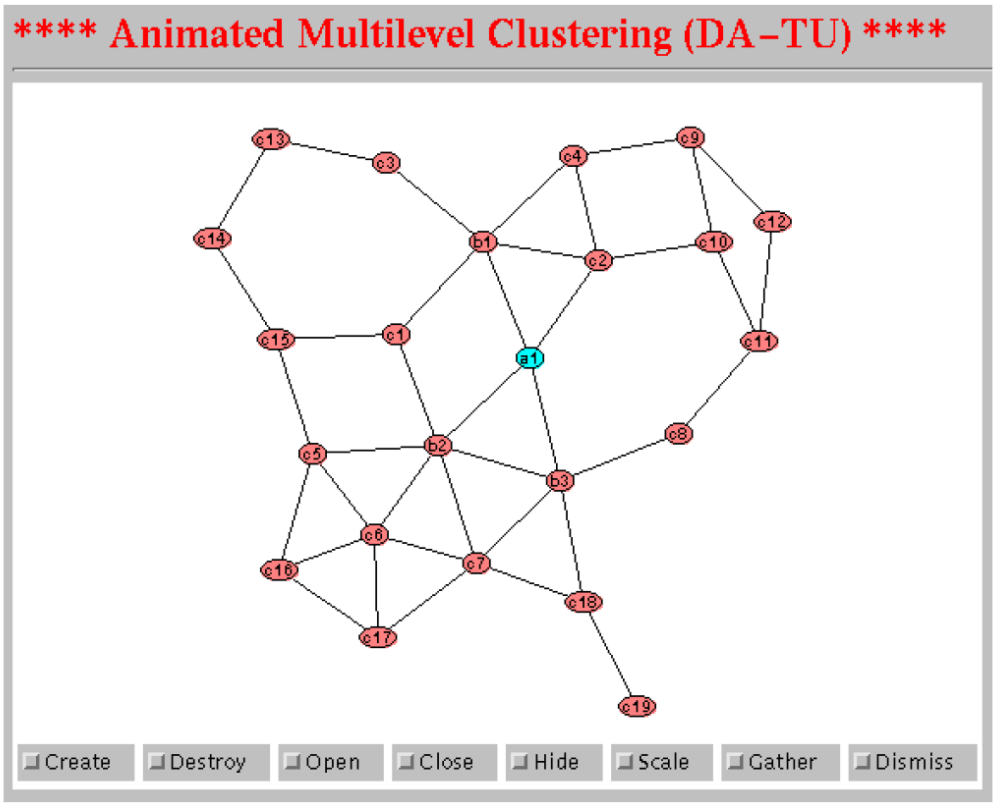


Figure 3.3. A screen of the visualization tool DA-TU, introduced by Eades and Huang [14, 26], Figure by [14].

in sequential drawings, secondly to minimize the movements of nodes within the clusters, and with least significance, to preserve length and routing of edges. The algorithm takes a layout of a graph and the changed new graph. It considers the layout as a good starting position and assigns all nodes and clusters that remain the old coordinates. The layout and the new graph are merged. In the following first layout phase, the clusters and nodes without changes get pinned not to leave their positions. Nodes removed from a cluster are replaced with virtual dummy nodes to preserve the layout within the cluster. Each cluster has got a virtual gravitational center node to which all nodes are connected with an edge. If the resulting layout does not fulfill given criteria, especially hitting a given range for the densitiy metric calculated as the ratio between area of the cluster's bounding box and the number of vertices, a second layout is produced without the pinning of nodes and clusters. The layout among the two fulfilling the criteria best is chosen.
The algorithm can be implemented based on any force-directed layout algorithm that allows for the pinning of vertices and the assigning of individual spring lengths and weights, as the algorithm divides between the following edge types:

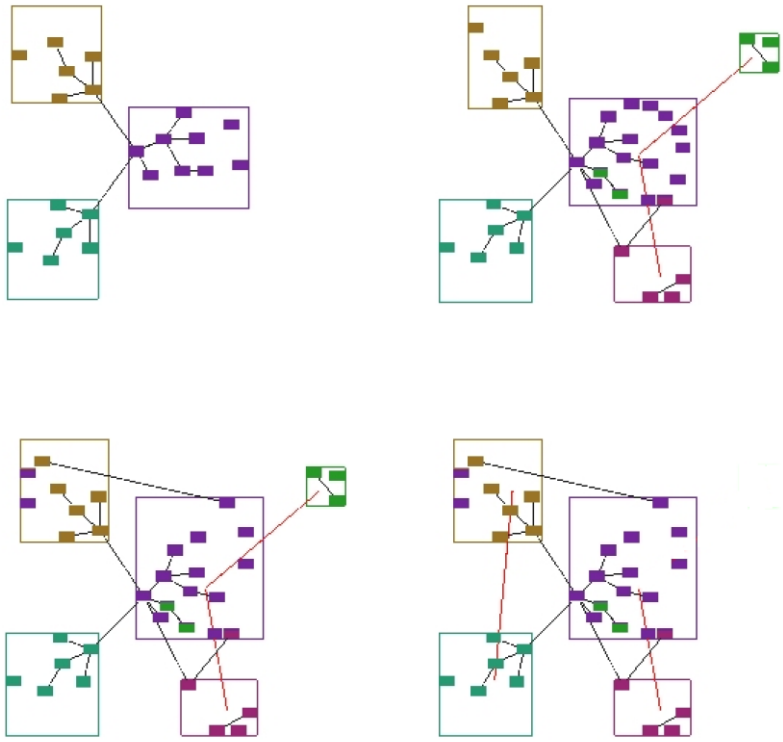
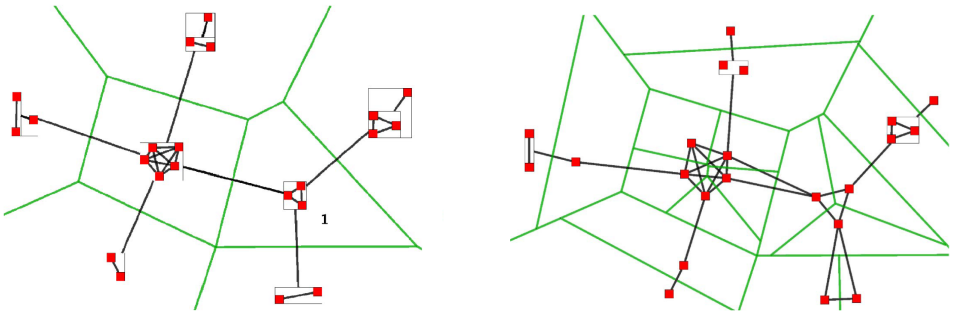


Figure 3.4. Snapshots of a small animation sequence showing incremental layouts of clustered graphs computed by the algorithm of Frishman and Tal [22]. Figure by [22].

1. Edges between dummy vertex of the cluster and static vertices
2. Edges between dummy vertex of the cluster and moveable vertices
3. Edges between vertices of the same cluster
4. Edges between vertices of different clusters
5. Edges between clusters

Of these the second has a longer assigned length than the first, the fourth longer than the third. and edges of the fifth type are varying with application specific constraints. In the implementation, the authors animate the changes with a sequence of drawings smoothing the step from one picture to another. Figure 3.4 shows snapshots of a small animation sequence computed by the tool of Frishman and Tal.

Bourqui and Auber To draw clustered weighted graphs in a force directed approach, Bourqui *et al.* [3] propose to extend the Graph Drawing with Intelligent Placement (GRIP) algorithm [25] in order to manage edge weights in drawing clustered weighted graphs. Additionally, the authors make use of *Voronoi diagrams*. A



(a) Voronoi diagram for the top depth level quotient graph.

(b) Voronoi diagram for the second depth level quotient graph.

Figure 3.5. Voronoi diagrams for quotient clustered graphs of differend depth levels. Figure by [3].

Voronoi diagram is a decomposition of a space given by distances to specified points in the space called *generator points*. A plane with n generator points is separated into n cells, one for each generator point. The cell is determined by its containment of all points whose distance to the point is not greater than its distance to any other generator point. Okabe [32] provides a closer look on this kind of diagrams and their usage. Bourqui and Auber use the diagrams to enable the drawing of clusters in non overlapping convex regions. The algorithm uses maximal independent set filtration and the so called intelligent placement as the GRIP algorithm, but extends this method to edge-weighted graphs by using weighted distances. As graph distances have to be computed in the intelligent placement the authors propose to use a tree t-spanner, a spanning tree that approximates graph distances with a factor t. As the problem of finding one is NP-complete, the authors introduce a heuristic.

The algorithm is a top-to-bottom multilevel drawing algorithm. It works by first drawing the top depth level quotient graph using the GRIP placement strategy and a force-directed algorithm capable of respecting edge weights. Then a Voronoi diagram for this quotient graph is computed. It displays the convex regions in which the nodes represented by the metanode can be drawn. After that, quotient graphs of each level are laid out with the same force-directed algorithm with the restriction that the nodes are forced to stay in their corresponding cell of the Voronoi diagram. New Voronoi diagrams are drawn in the cells of the Voronoi diagram of the level above. Figure 3.5 shows Voronoi diagrams for quotient graphs of different depth level. The restriction to movements within the Voronoi cells has a similar effect to the introduction of virtual nodes for each cluster.

Itoh et al. [28] are not directly concerned with drawing clustered graphs, but they introduce a layout algorithm for undirected multiple category graphs, in which each node can be attributed with one or more categories. The relation to clustering is introduced by the fact that the authors wish to display clustering with respect to category as well as to adjacency. The nodes are to be drawn in different colours

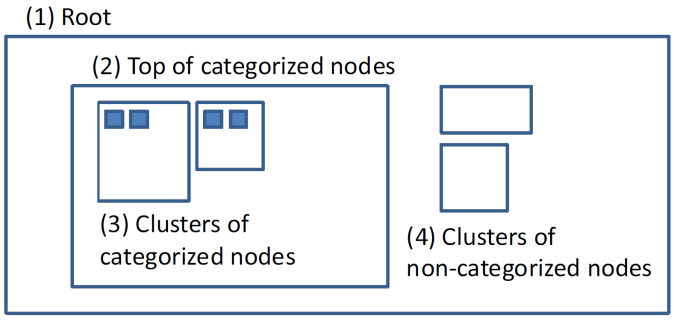


Figure 3.6. Space filling hierarchy layout. Figure by [28].

in style of a pie chart according to the categories they belong to. Additionally the radii of the nodes differ with the number of adjacent edges. Furthermore the edges are drawn differently according to their type: edges between two categorized nodes, edges between a categorized node and a noncategorized node and edges between two nodes without category. The authors employ a hybrid method, involving force directed layout as well as space filling/tree mapped technique. It is their goal to show clustering with respect to category (extrinsic) as well as adjacency (intrinsic). A formative structure of the algorithm is the following: all nodes are regarded to be included in a rootnode, all categorized nodes included under a top-of-categorized-nodes node. Clusters of noncategorized nodes are regarded to be children of the root node. The algorithm works in four steps:

1. build clusters of categorized nodes,
2. apply force directed layout,
3. use the calculated positions as a template, and
4. apply space filling layout.

In the first step, categorized and noncategorized nodes are divided and each set is connectivity clustered with the use of the Fast Modularity Community Structure Inference Algorithm by Clauset et al. [4]. A dendrogram, i.e. a tree diagram, is build by this algorithm, which is traversed bottom-up to build hierarchical clusters. This is computationally very costly, but the result can be stored for reuse. The second phase substitutes nodes for clusters and performs a force directed layout, whose outcome is used as a template for the final tree mapped space filling layout. This is based on splitting the drawing space according to the hierarchical structure of the tree. An illustration of the space filling tree-mapped layout is shown in Figure 3.6.

For a closer look on tree-mapped space filling see also Wattenberg [48].

▶ 3.1.2 Other Methods for Drawing Clustered Graphs

There are several further approaches to the drawing of clustered graphs.

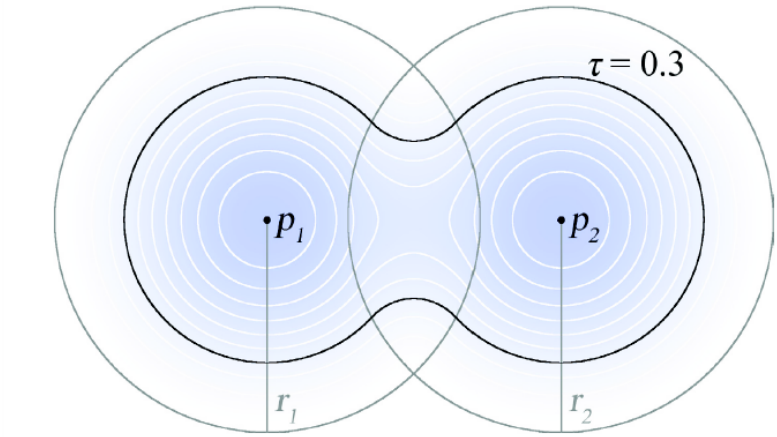


Figure 3.7. An implicit surface defined in $\mathbb{R}^2$ by two generator points p_1 and p_2 with radii of influence r_1 and r_2 and a threshold of $\tau = 0.3$. Figure by [1].

Balzer and Deussen [1] introduce the concept of *implicit surfaces*. Object of their paper is the level of detail visualization of clustered graph layouts. It describes the different handling of clusters and edges in the changeover between detail levels. First, the authors describe the cluster representation. The authors state that it is unsatisfactory to use just a single primitive or even a set of primitives to visualize a cluster when the representation of a cluster could well display the distribution of its children. The representation of the cluster in the next abstraction level should have the same shape as the distribution. To achieve this, the concept of an implicit surface is presented. An implicit surface can be described by a set of generator points P, each of the generator points $p_i \in P$ having a radius of influence r_i. A generator point influences a point q in a way defined by a density function $D_i(q)$:

$$D_i(q) = \begin{cases} (1 - (\frac{\|q-p_i\|}{r_i})^2)^2, & \text{if } \|p - p_i\| < r_i \\ 0, & \text{if } \|q - p_i\| \geq r_i \end{cases} .$$

The *density field* F is the summation of the density function for all generator points:

$$F(q) = \sum_i D_i(q) - \tau$$

with a threshold $\tau \geq 0$, thereby defining the implicit surface with $F(q) = 0$ as those points q where the sum of the density values of all generators equals τ. This concept is illustrated in Figure 3.7. The authors state that this concept can be extended to arbitrary generator objects by solely substituting the distance computation between the point p_i and g in the equation.
Using these implicit surfaces, the authors calculate clustered graph visualizations bottom-up: First, an implicit surface is generated for each cluster of a node, which has only vertex children. Then the generated implicit surfaces act as generator

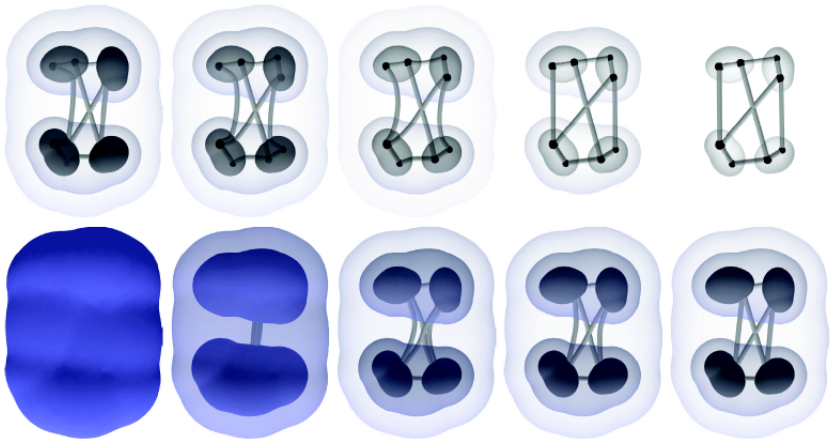


Figure 3.8. The opacity of clusters determines visibility in different levels of detail. Figure by [1].

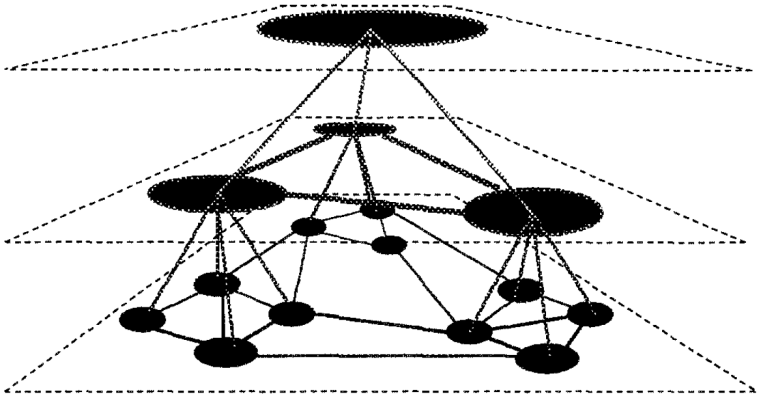


Figure 3.9. A multilevel visualization of a clustered graph. Figure by [11].

objects for implicit surfaces of the next level. In the following, the authors propose to use a transparency approach in the changeover: Objects adapt their opacity according to their distance to the current view point. How the visibility is determined by the opacity is shown in Figure 3.8. Furthermore the authors introduce the routing of an edge (u, v) along the path from u to v in the inclusion tree. If a number of edges share parts of this route, they are bundled to ease their display as abstracted cluster-to-cluster edge.

Eades and Feng [11] aim for the multilevel visualization of clustered graphs. The authors' goal is to display the structure of complex clustered graphs by drawing them in three dimensional pictures with several layers. Each level of the inclusion tree is drawn on a plane at different z-coordinates. An example is shown in Figure 3.9.

The *multilevel drawing* of a clustered graph $C = (G, T)$ consists of:

- A sequence of plain drawings of views for each level from the leaf level to the root level.
- A three-dimensional drawing of the inclusion tree. A node of height i is drawn in the plane $z = i$ and within the region of its cluster in the drawing of the view at that level.

Here, the view of a clustered graph $C = (G, T)$ at its level i is a graph $G_i = (V_i, E_i)$ where V_i consists of the nodes of C with height i in T.

The authors introduce the term of *c-planarity* for the level views as absence of edge crossings or edge-region crossings—with edge-region crossings denoting an edge crossing a region more than once. If the drawings of views at all levels of a multilevel drawing are c-planar, the drawing in the whole is called c-planar.

Two different drawing conventions are considered:

Straight-Line Convex Drawings: The goal is to represent the edges as straight-line segments and the regions for clusters as convex polygons. The authors suggest to use the algorithm of Tutte [46], which draws each triconnected planar graph such that every face is a convex polygon, the drawing is planar and straight-line. Before applying the algorithm, the authors create a *skeleton* for each cluster v, which is the subgraph consisting of the vertices and edges on the outer faces of the child clusters of v. The algorithm is applied recursively to every skeleton graph, and a convex polygon is computed for the outer face of each cluster. As Tutte's algorithm is restricted to triconnected planar graphs, the approach requires the skeletons of the clustered graph to have this connectivity property.
The authors add some remarks about alternatively using the layer based drawing scheme with an *c-st numbering*[3] as a basis for the layer assignment. This guarantees that each cluster occupies consecutive layers and such can be surrounded with a convex hull. The authors state that it can be shown that such a drawing contains no edge crossings and no edge-region crossings unless the regions are drawn in rectangular instead of convex style.

Orthogonal Rectangular Drawings: In this case, the edges are to be drawn as sequences of horizontal and vertical segments, leaf nodes as grid points and regions as rectangles. The authors suggest to use the method they introduced in [13].

The multilevel drawings are constructed by drawing the views at every level. Edges connecting clusters c_1 and c_2 in a higher level are summarizing representatives of edges between nodes belonging to these clusters in lower levels. The node positions

[3] For an edge (s, t) in a biconnected graph with cardinal number of vertices n, an st numbering is a numbering of all vertices so that s receives number 1 and t number n and any other vertex is adjacent both to a vertex of lower and a vertex of higher number. If all vertices in a cluster are numbered consecutively, the st numbering is called c-st numbering.

are drawn recursively by placing the leaf nodes as in a two dimensional plane drawing, and for every node of level i, the average of the xy-coordinates of its children at level $i-1$ is set as position.

▶ 3.2 Drawing Compound Graphs

The approaches to automatic layout for compound graphs can be roughly subdivided into force based approaches, which are introduced in section 3.2.1, and layered approaches, which are introduced in section 3.2.2.

▶ 3.2.1 Compound Graphs in the Force Directed Approach

There are different approaches to the automatic drawing of compound graphs by the force directed approach. See section 3.1.1 on the term of force directed approaches. As the implementation part of this thesis extends a layered layout algorithm, the approaches in this section could not be chosen as basic ideas. However they will have to be considered for future work, if a force directed algorithm for compound graphs is to be implemented.

Bertault and Miller [2] present an algorithm called *nuage* for drawing directed or undirected compound graphs. The algorithm can be parameterized with a drawing algorithm for compound graphs without hierarchy crossing edges, which are referred to as *nested graphs* in the paper. The recursive appliance of an algorithm for flat graphs would suffice. The algorithm works in two steps and optionally one refinement step, which is performed as a preprocessing step.

Refinement step: As the drawing of the compound graph in the first two steps simply ignores edges between nodes that do not have the same parent, unnecessary edge crossings are introduced. The goal of the refinement phase is to eliminate those crossings. A new graph G_r is built. The node set is the same as the one of the compound graph to be drawn, but the edges are not adopted unless they contain nodes with different parents. The initial node positioning is obtained by applying a force directed algorithm with disabled repulsion forces. The relative node positioning is to be regarded in the following steps, thus requiring an algorithm for the drawing of nested graphs that is apt to take adequate layout constraints. The complexity of the refinement step is stated to be $O(|E| \log |V|)$.

Step 1: First step of the algorithm nuage is to construct a nested graph $N = (V, E', I)$, V, E' and I denoting vertex set, adjacency edges and inclusion edges respectively, from the given compound graph $G = (V, E, I)$. Any edge $e = (u, v) \in G$, for which parent(v) = parent(u) holds is part of E'. Otherwise the edge is replaced by $e' = (u', v')$, u' being an ancestor of u[4], v' an ancestor of

[4]Note that the definitions of ancestor(v) and descendant(v) in the paper of Bertault and Miller are defined to return a set comprising v itself.

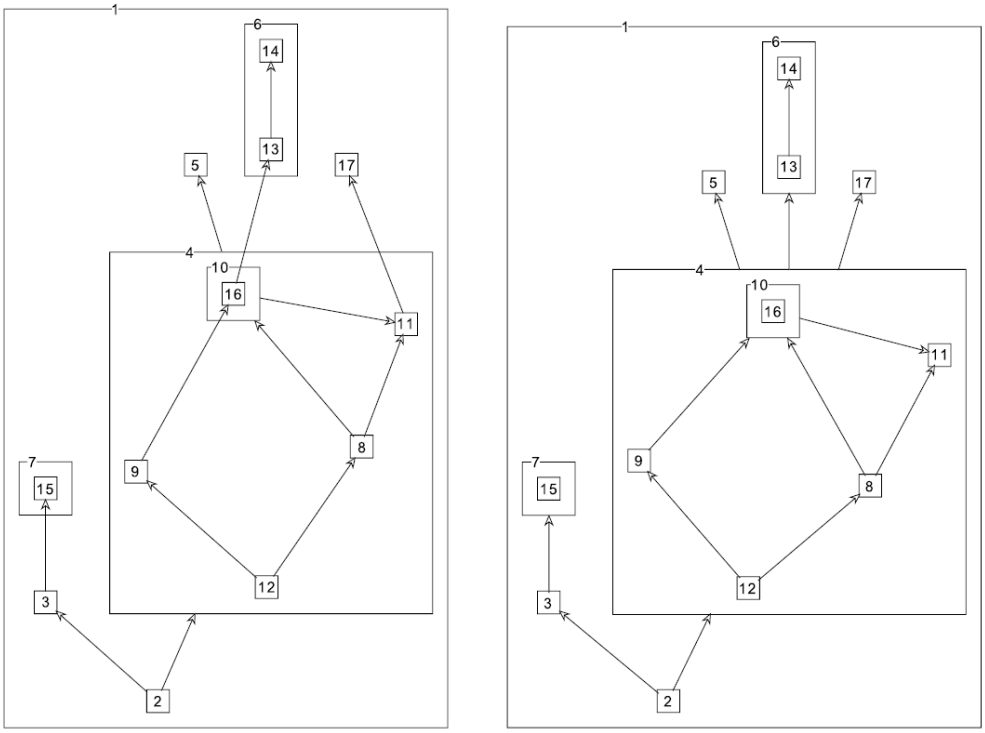


Figure 3.10. A compound graph and its associated nested graph. Figure by [2].

v with $a' \neq b'$ and $Pa(a') = Pa(b')$. An example of a nested graph constructed for a compound graph is shown in Figure 3.10. The complexity of this step is stated to be $O(|E| \log |V|)$ in average and $O(|E||V|)$ in worst case.

Step 2: In the second step, the drawing of the nested graph takes place, in which the algorithm for the drawing of nested graphs is applied. The authors suggest the recursive employment of a classical graph drawing algorithm.

Dogrusoz et al. I. Another force directed layout scheme for compound graphs is that of Dogrusoz *et al.* [7]. It is extended to support application-specific constraints. The algorithm is implemented within the analysis tool for biological pathways, Pathway Analysis Tool for the Integration and Knowledge Acquisition (PATIKA). The base model is that of repelling particles that are connected with springs of individually defined ideal lengths. Nested graphs are viewed as "carts" that can move in all directions, possibly carrying another cart on top. Nodes of the associated subgraph can only move within the bounds of the cart, these however, are considered to be elastic enough to adjust to actualized dimensions of the contained subgraph. Any subgraph is supplemented by a virtual center of gravity, exerting attractive forces on all nodes within this subgraph, in order to keep them together. The length of

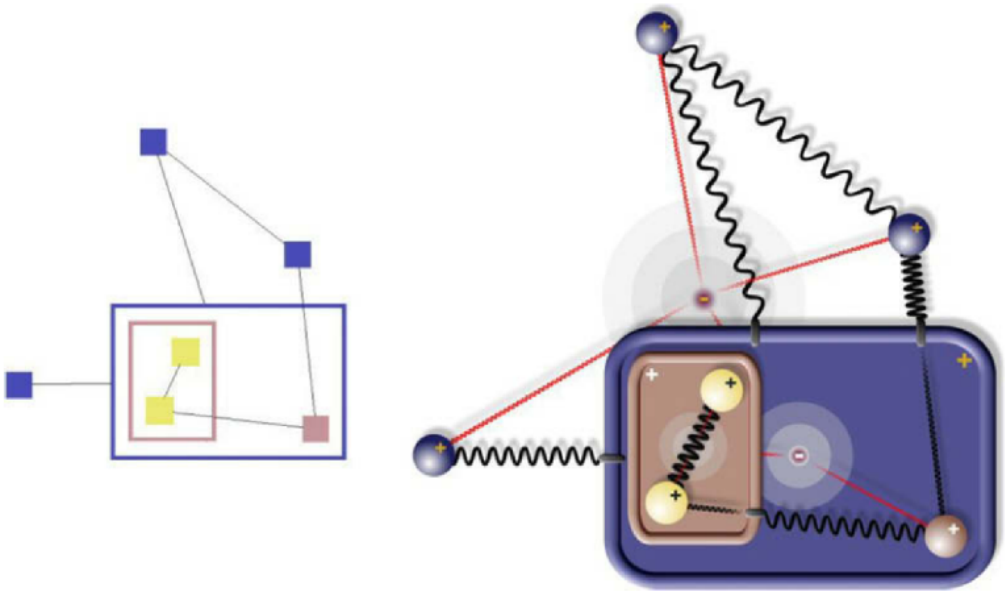


Figure 3.11. A compound graph and its associated force model. Figure by [7].

an inter-graph edge is treated specially with respect to the calculation of its ideal length: It is calculated in proportion to the nesting depth of the graphs the end nodes belong to. Figure 3.11 shows a sample compound graph and illustrates the associated model. As PATIKA is intended to draw biological pathways, two application specific constraints are integrated as well: First, the nodes have to keep to special biological compartments, which are represented by regular regions. This is achieved by a regular adaptation of the compartment bounds. Second, the convention to align substrates and products of a transition in the pathway requires a special flow, called orientation. To comply with this request, the current relative positions are analysed to calculate a relativity force to act upon the nodes in question. The algorithm consists of three phases, each comprising a number of iterations, set by an interaction count per phase:

1. In the first phase the skeleton graph (original graph without subgraphs that are trees) is laid out with the spring embedder model, but with relativity and gravitational forces disabled.

2. The trees are introduced back to the graph level by level, while the complete force model is applied.

3. In the third phase the layout is improved by more iterations with all forces activated.

The simplified pseudo-code of the algorithm is as follows:

Listing 3.1. CompoundLayout

```
1  call INITIALIZATION()
2  set phase to 1
3  if layout type is incremental then
4      increment phase to 3
5  while phase ≤ 3 do
6      step:= maxIterCount(phase)
7      while (step > 0 or !allTreesGrown do
8          call APPLY_SPRING_FORCES()
9          call APPLY_REPULSION_FORCES()
10         if phase ≠ 1 then
11             call APPLY_GRAVITATION_FORCES()
12             call APPLY_REPULSION_FORCES()
13         call CALC_NODE_POSITIONS_AND_SIZES()
14         call UPDATE_COMPARTMENT_BOUNDS()
15         if phase = 2 and !allTreesGrown and step%growStep = 0 then call
               GROW_TREES_ONE_LEVEL()
16         step + = 1
17     phase − = 1
```

Dogrusoz et al. II. In a subsequent paper, Dogrusoz *et al.* [8] introduce a force directed layout algorithm for undirected compound graphs. It handles multilevel nesting with edges between arbitrary nesting levels and varying node sizes, and is open to application specific constraints. The implementation of the algorithm is not confined to a special convergence schema, the authors implement it based on the algorithm by Fruchterman and Reingold [23]. Dogrusoz et al. define the term of a *graph manager* as $M = (S, I, F)$ with S being the *graph set* $S = G_1, G_2, \ldots, G_l$, I the inter graph edge set, $F = (V^F, E^F)$ a rooted nesting tree. The underlying simulated physical system comprises electrically charged particles and springs of a desired ideal length, of which the first are associated with the nodes and the latter with the adjacency edges of the graph. For simplicity, nodes are considered to repel each other only in the case that they are located in the same graph. Additionally, a gravitational force towards the center of each subgraph is introduced, which is intended to keep the components together. This force is defined to be independent of the node size or its distance to the graph center. The ideal length is computed for each edge separately, involving the size of the adjacent nodes. In this model, a minimal total energy state is searched for. In the process, a simulated annealing technique is applied, which means that a temperature value, decreasing with every iteration, limits the movement of the nodes. A node with a nested graph is called *expanded*. In the algorithm, an expanded node and its nested graph are represented as a single entity, which the authors compare with a cart. The cart is supposed to be elastic in its boundaries, ready to adapt them to the current size of the nested graph. Hierarchy crossing edges are subject to a special treatment. The basic idea is to split the edges into two parts. The first is the part that is located in the subgraph containing one of the nodes. It administrates forces meant to keep the node close to

the boundary of the subgraph. The other is the part leaving the subgraph, acting as a regular spring. As this requires heavy computation, this solution is approximated by manipulation of the desired edge length. An amount proportional to the sum of the depth of the end nodes, calculated from their common ancestors, is added to the ideal length value. The authors state that application specific constraints can be introduced as additional forces. As an example they refer to PATIKA, which handles orientation flow and compartment-constraints.

The algorithm is working in several steps:

INITIALIZATION: Nodes are set to initial size, a threshold for the convergence is set, a random initial node positioning is calculated, trees are moved from the graph, the graph *skeleton* remains.

STEP 1: The spring embedder algorithm runs on the skeleton of the graph with gravitational forces and application specific constraints deactivated

STEP 2: The spring embedder algorithm runs in multiple iterations with gravitational and application specific forces activated - the trees are introduced back into the graph level by level.

STABILIZATION: The layout gets "polished".

The spring force $e = (u, v)$ is calculated as:

$$F_S = \frac{(\Lambda - \|P_u - P_v\|)^2}{\eta} \overrightarrow{P_u P_v}$$

with Λ being the ideal edge length, η the elasticity constant of the edge and P_u and P_v the positions of the nodes u and v.

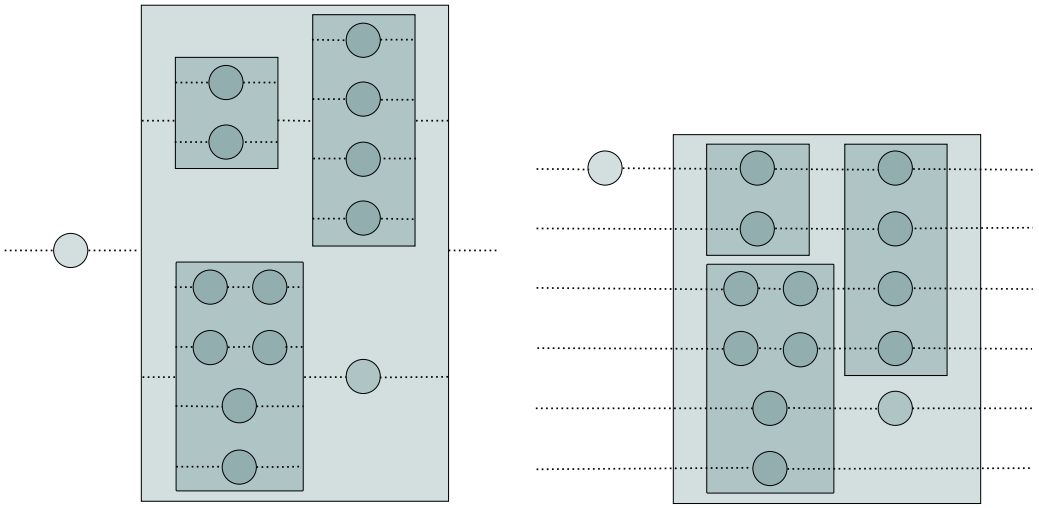
The repulsion force is defined as:

$$F_r = \frac{\alpha}{\|P_u - P_v\|^2} \overrightarrow{P_u P_v} \ .$$

The node positions and sizes are calculated bottom up, the compound nodes propagate back to their children for an update of bounds. The authors give a good documentation in pseudocode. Note that according to a statement of the authors, the algorithm's performance decreases dramatically when the nesting depth increases.

▶ 3.2.2 Compound Graphs in the Layering Approach

For the layout of directed compound graphs the literature provides approaches of the layering method. This method is based on dividing the drawing space into different layers, into which the nodes are sorted in a way that all edges point into the same direction. As this is not possible for graphs that contain cycles, the drawing algorithms by this approach include a phase for the removal of cycles. Afterwards, nodes are sorted into the layers. The algorithm also has to find an ordering of the



(a) Local layering. (b) Global layering.

Figure 3.12. The difference between local and global layering (top to bottom).

nodes in each layer that supports the goal of minimizing edge crossings. Actual node placement and edge routing are further prerequisites to the final drawing of the graph. The methods to draw compound graphs in the layering approach can be roughly divided into methods that use *local layering* and methods that employ *global layers*. Local layering means that one distinct set of layers is employed for each compound node. Global layering on the other hand employs one set of layers for all nodes, which means that compound nodes have to span more than one layer. This is mostly implemented by representing the compound node with the help of several dummy nodes. Forster [20] gives a good overview of the advantages and disadvantages of the two methods. The advantage of the local layering method is a generally good computation time, while the advantage of the global layering is that it leads to more compact drawings. It can additionally be observed that local layering needs several passes of a general layering algorithm, while global layering does not.

Figure 3.12 shows the differences of the two approaches. The global layering shown in 3.12(b) is more tightly arranged than the local layering pictured in 3.12(a).

Local Layering

Sugiyama and Misue I. Sugiyama and Misue [43] present an algorithm for the automatic drawing of compound directed graphs based on the hierarchical layer layout method. This algorithm was considered, but not chosen as a basic idea for implementation in this thesis. The reason was a general decision in favour of global layering, see The algorithm is restricted to graphs without adjacency edges between nodes in an ancestor-descendant relationship. It is partitioned into four steps: In the

first step, the *hierarchization* takes place, in which the inclusion tree gets annotated with compound level assignments for each node. The second step, the *normalization*, takes the assigned compound digraph and yields a proper assigned compound digraph. This means that all adjacency edges lead downward with respect to the compound level assignment of the nodes and associate only nodes with parents of the same compound level. Edges violating these criteria are replaced with appropriate dummy vertices, dummy inclusion edges and dummy proper adjacency edges. The next step consists of the *vertex ordering* in purpose of the minimization of edge crossings and edge-rectangle crossings as well as the achievement of closeness between adjacent vertices. The algorithm works on subgraphs, fixes the positions of nodes adjacent to nodes of other subgraphs in border positions, and performs the ordering of other vertices based on barycentric methods. A fourth step is dedicated to the determination of the *metrical layout*. It consists mainly of the vertex positioning, including the determination of horizontal and vertical positions as well as width and height of the bounding boxes. The edge routing as the second part of metrical layout is derived from the vertex positions, since proper adjacency edges are drawn as straight lines, while nonproper adjacency edges are routed with the help of dummy and virtual vertices originating from step two by assigning them zero width. The routing is not orthogonal.

In more detail, the four steps comprise the following actions.

Step 1. In the context of the assignment of compound levels, the notion of a derived graph is used, in which all adjacency edges between vertices of different levels in the inclusion tree are replaced by edges representing $<$ and $\leq$ relations of nodes of the same level. In this derived graph, a heuristic is employed to eliminate feedback edges in the graph in order to resolve cycles, preferring edges of the derived type and among those the ones representing $\leq$ relationships. Then the assignment of compound levels is based on the structure of the cycle free derived compound graph.

Note that the algorithm works with local layers—each node spans only one layer, while the interior of compound nodes is divided in a set of layers of its own.

Finally the focus is back on the newly assigned compound digraph, in which every edge whose direction is not downward in the compound level hierarchy gets reverted. This reversion is undone in the last drawing step.

Step 2 Nonproper adjacency edges are replaced with the help of dummy nodes. The dummy vertex with the highest assigned compound level is placed in the rectangle of the lowest leveled subgraph containing both start and end node. This means that the edges are routed outside the bounding box of any subgraph that does not contain both nodes associated by the edge. Therefore the edges tend to leave the lower surrounding rectangle at the side.

Step 3. In order to attain closeness between adjacent vertices of different subgraphs, for each node the difference between the number of adjacencies to

subgraphs to the left and right in the ordered compound graph is calculated. If it is not zero, the node is placed to the accordant border of the rectangle and fixed there, multiple nodes of this kind in order of the calculated excesses of connections to subgraphs on the side in question. The edge crossing problem is treated by the heuristic two-level barycentric method [45]. The ordering of the nodes of the second level according to their upper barycenters is alternated with the ordering of the nodes of the first level according to their upper barycenters. A barycenter is calculated as the sum of the positions of adjacent edges in the complementary level divided by their number. The number of the edge-rectangle crossings is minimized by another heuristic called insertion BC. Into each edge between nodes of one layer, a dummy node is inserted, which is positioned in the layer above. Then the layer of the dummy nodes is ordered with regard to the lower barycenters, followed by an ordering of the original nodes according to their upper barycenters. The algorithm employed is an extension of the two level barycentric methods to an n-level hierarchy. It iterates over the levels. First, the iteration is carried out from level one downwards, performing upper level barycentric ordering, insertion of upper level dummy nodes, and repetition of upper level barycentric ordering (down step). Second, the iteration is reversed starting with level $n - 1$, inserting dummy nodes to the lower level, and ordering according to the lower barycenters. The whole cycle is repeated a fixed number of times.

Step 4. The procedure for the metrical layout accepts an ordered digraph and the width of leaves as input and calculates the widths of all other nodes as well as their global horizontal positions. The procedure employs a heuristic method called PR method. It works on a customized local hierarchy and moves the nodes positions according to their barycenters without disturbing the order of the level.

Sugiyama and Misue II. In 1996, Sugiyama and Misue [44] gave a short introduction of the compound graph visualizer D-Abductor. Features are:

- The author's drawing algorithm for compound graphs
- Graph editing
- Collapse and expand operations
- Display with animation (preservation of mental map)
- Network communication among multiple set of D-Abductor possible
- Interface for applications (Simple)

A sample screen is shown in Figure 3.13.

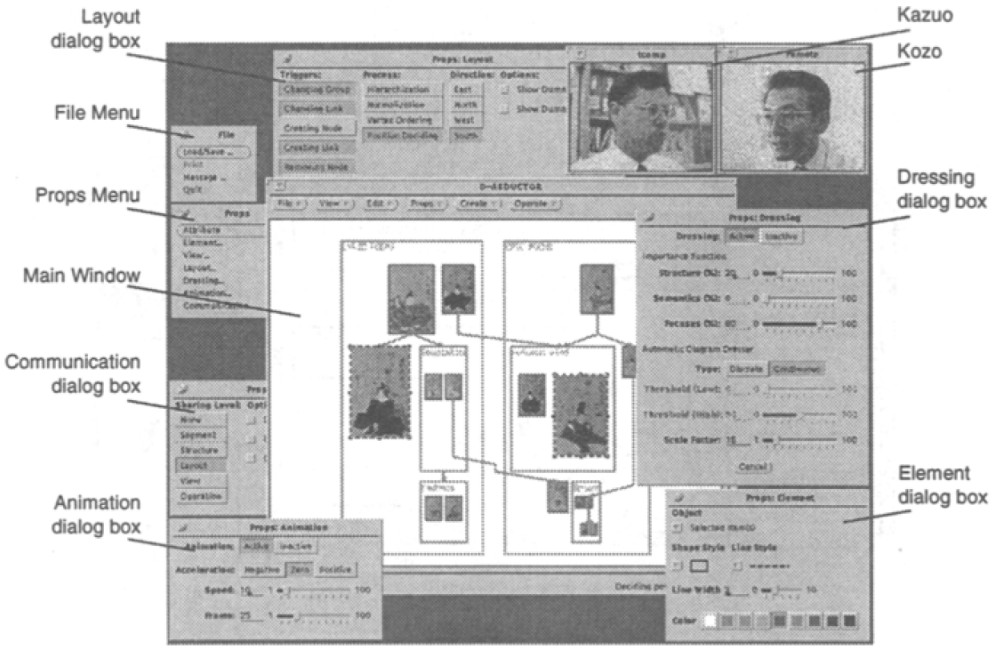
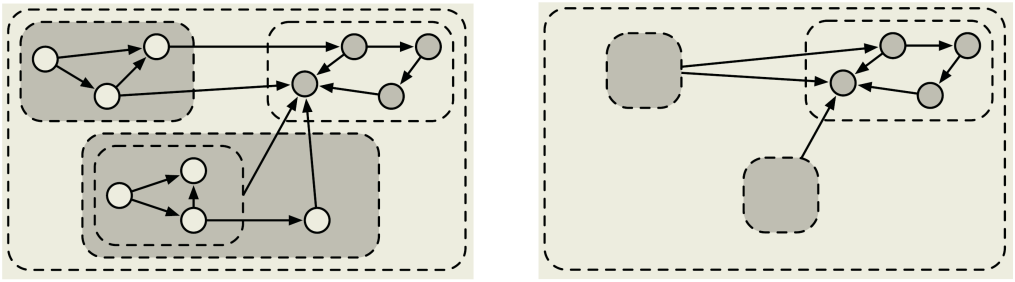


Figure 3.13. A sample screen of the D-Abductor-tool. Figure by [44].



(a) Compound graph, expanded view.

(b) Compound graph, contracted view.

Figure 3.14. Expanding and contracting compound nodes. Figure by [34]

Raitner [34] presents a method for creating views for a graph drawn with the method of Sugiyama and Misue [43, 44]. Goal is to be able to expand or contract compound nodes in the drawing of the graph without the need to redraw the whole graph. Figure 3.14 shows an example of an expanded and a contracted view of a compound graph. The author states that reducing the recalculation to the affected part of the drawing reduces calculation time and helps to preserve the mental map of the user. The redrawing follows the four steps of the algorithm, determining the exact part of the graph for which calculations have to be redone expanding a node:

HIERARCHIZATION: The level assignment can be restricted to the subgraph induced

by the children of the node affected by the navigational operation. All old nodes stay on their levels.

Normalization: The node in question is expanded which comprises the adding of the induced adjacency edges and appropriate inclusion edges, while the edges connected to the meta-node in contraction are removed. Newly introduced improper adjacency edges are made proper. The method is exactly that of Sugiyama and Misue [43, 44].

Vertex Ordering: The vertex ordering algorithm is applied to the subtree rooted at the expanded node. Positions of dummy nodes of induced edges[5] are reused. Dummy nodes of all edges inheriting from the replaced induced edge are treated as one block, which is positioned according to the dummy node of the edge replaced. Goal of this method is to preserve the mental map by routing the expanded edges along the same course as the old edge.

Metric Layout: The local coordinates are adjusted for the expanded node and all its ancestors. The Step of summing up the absolute coordinates in traversal of the hierarchy remains unaltered.

The author states that contraction of a node that has been expanded reverts the mentioned steps in a straightforward way, while contracting a node that has not been expanded before is the critical case. He explains that a derived edge of the type $\leq$ in the initial layout can lead to massive changes in the level assignment, which is harmful to reaching the goal of preservation of a mental map. His proposal is either not to allow contraction in nodes that have never been expanded or only to allow derived edges of type $<$, thus accepting that the initial view might be less compact.

Global Layering

Sander I [38] presents a layout method for general directed compound graphs. It is based on hierarchical layered layout and similar to the algorithm by Sugiyama and Misue [43]. Sander's method provides the basic idea for the implementation part of this thesis. Like in this method, layers are assigned to the nodes. To achieve a proper hierarchy that contains adjacency edges only between nodes of adjacent layers, dummy nodes and dummy edges are inserted to replace edges spanning several layers.
The layers of this algorithm are global layers, so there is only one set of layers for all nodes of the graph; in particular no compound graph has its own internal layers. It might span several global layers instead.
In difference to the method of Sugiyama and Misue [43], the level assignment is based on the concept of the *extended nesting graph*. In the nesting graph, each node

[5] Induced edges are edges connecting a meta node to other nodes because of an adjacency of children of the meta node.

without leaf-quality with respect to the inclusion tree is represented by two dummy nodes for its upper and lower border, denoted by $u^{(+)}$ or $u^{(-)}$ respectively. The nesting graph contains these dummy nodes, the leaves, and the following edges:

1. an edge $(u^{(-)}, v)$ for each (u, v) in the inclusion tree (E_T) with u being an inner node of the inclusion tree $(\in S)$ and v being a leaf node of it $(\in B)$,
2. an edge $(u_1^{(-)}, u_2^{(-)})$ for each $(u_1, u_2) \in E_T$,
3. an edge $(v, u^{(+)})$ for each $(u, v) \in E_T$ with $u \in S, v \in B$, and
4. an edge $(u_2^{(+)}, u_1^{(+)})$ for each $(u_1, u_2) \in E_T$ with $u_1, u_2 \in S$.

It can be viewed as the inclusion tree and its mirror image joined at the leaves. This nesting tree is successively extended by edges representing adjacency edges, if they do not introduce a cycle. The rank assignment is derived from the topological order of the extended nesting graph. The rank of the upper dummy nodes is afterwards corrected with the help of the converse topological order to keep the upper border of a subgraph near to the position of contained leaf nodes.
In the following, cycle-producing edges are analyzed with respect to the question whether the choice of adequate border nodes as anchor points of the edge would prevent the need to revert the edge to achieve uniform edge orientation. This is especially helpful in the case of edges from descendant to ancestor, and for that reason, this step is not found in the method by Sugiyama and Misue [43] method, which is restricted to graphs without edges of this type. Edges whose adverse direction cannot be solved by appropriate anchor point placement are reverted. The original direction is revived in the final drawing step. Next, empty layers are removed, before the splitting of long edges that cross several layers is performed. The latter is done with the goal of reducing the span of each edge to one layer. This is achieved by inserting dummy nodes and dummy edges. If edge (w_1, w_2) is to be replaced, all dummy nodes are assigned at least to the subgraph that contains w_1 and w_2. In contrast to the solution of Sugiyama and Misue [43], this method routes the edges inside of border rectangles as far as possible. This means that all dummy nodes for an edge (w_1, w_2) will be routed inside subgraph u, if w_1 or w_2 are inside u and $R_{min}(u) \leq R(dummy) \leq R_{max}(u)$ holds with R, R_{min}, R_{max} returning the rank, rank of lower border node, and rank of upper border node, respectively. Consequently, the edges tend to leave border rectangles at the top or bottom instead of on the sides. Compared to the method of Sugiyama and Misue [43], the phase of crossing-reductions follows another strategy as well. Though both methods use barycentric heuristics, the application is per subgraph in the approach of Sugiyama and Misue [43], while Sander performs the node ordering globally. First, the crossing reduction with barycenter weights is done for the complete levels of the graph with

$$W_p(v) = \frac{1}{|pre(v)|} \sum_{w \in pre(v)} P(w)$$

as the barycenter weight for top down traversal with $pre(v)$ returning the direct predecessors of v with respect to adjacency and $P(w)$ returning the actual position of a w. Accordingly, the barycenter weight for bottom up traversal is

$$W_s(v) = \frac{1}{|suc(v)|} \sum_{w \in suc(v)} P(w)$$

with $suc(v)$ returning the direct successors of v. The achieved node ordering is used as a starting point for repositioning phase, in which the nodes are placed such that border rectangles can be drawn around subgraphs. This requires

1. that the nodes of a subgraph of the same rank must be placed in an unbroken sequence in their layer, and

2. that nonnested subgraphs must not be intertwined across the layers, which means that all nodes of one subgraph must be exclusively left or right of all nodes of the other in all levels.

The basic idea of this reordering employs the notion of *average subgraph position* calculated as the sum of the positions of all leafs within the subgraph divided by their cardinal number. If the average subgraph position of subgraph u_1 is below that of u_2, it is expected that many nodes in u_1 are positioned left of nodes of u_2. Accordingly this average can be calculated for one layer alone, taking only the leaf nodes in this layer into account. The nodes are sorted bottom-up by setting the average subgraph position of a leaf node to its position and the cardinal number of its leafs to 1 and calculating the average subgraph positions layer-wise. The author states that the final solving of intertwined subgraphs can be done afterwards with the help of the complete average positions of subgraphs directly, but that this would lead to unnecessary reorderings. He introduces the concept of a subgraph ordering graph instead, which consists of all nodes and an edge (w, w') if there is a subgraph containing both w and w' and directly neighboured nodes of a common layer v and v', of which v belongs to w and v' is included in w'[6]. Cycles in the subgraph ordering graph are broken at the node with smallest total average subgraph position. If the subgraph ordering graph is sorted topologically, it yields an ordering λ_O denoting which subgraphs are left of other subgraphs (under the assumption that layering takes place from top to bottom). The algorithm performs a sorted traversal of all layers according to this order. If during the calculation of the order $\lambda_{O,i}$ for layer i there is a possibility to choose between possible next nodes, the node u with smallest barycentric weight

$$W_{p,i}(u) = \frac{1}{|\{v \in B | R(v) = i, u \rightarrow_T^* v\}|} \sum_{v \in B, R(v)=i, u \rightarrow_T^* v} W_p v$$

with $\rightarrow_T^*$ denoting inclusion.

[6]Note that inclusion is defined to be given in case of identity.

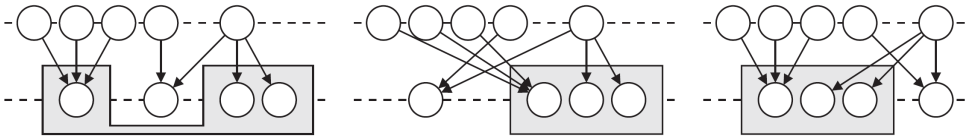


Figure 3.15. Introduction of unnecessary edge crossings by Sander's approach [38]. From left to right we see the layers before reordering, after reordering with Sander's method, and the optimal result. Figure by [20]

After this repositioning, the crossing reduction phase ends and the final positioning of nodes and edges takes place. To prevent overlaps of rectangles, sequences of border dummy nodes are inserted to the left and right of each subgraph, one dummy node per layer. Invisible edges are inserted between the dummy nodes of adjacent layers. A method to place these dummy nodes in straight vertical lines is introduced by Sander [37]. The author uses this method to place and route the edges.

Forster [20] intents to enhance the method by Sander [38] in the realm of crossing reduction. The method expects a proper layered clustered graph as input.
Sander already respects in his algorithm the two restrictions:

1. All nodes of the same compound node that are placed in one layer are to be drawn in an unbroken sequence, that means placed next to each other without nodes of other compound nodes placed in between them.

2. Compound nodes keep their relative position the same across all layers.

This is done by resolving violations of these rules after the actual crossing reduction by sorting the compound nodes with a barycentrical method. Forster criticizes this method for its tendency to introduce unnecessary crossings. An example for this is shown in Figure 3.15. He proposes a strategy in which the restrictions are considered right from the start.
It regards the first constraint by computing the child order of all compound nodes independently by introducing the concept of a so called *crossing reduction graph* G'_h which is associated to one compound node h of the graph G to be laid out. G'_h consists of two layers, of which the upper layer B'_1 is taken from G without modifications and the lower layer B'_2 comprises exactly the children of h in the corresponding layer hierarchy. The latter consists of all nodes that span the layer in question. For each edge $e = (s, t)$ in G incident to a successor of h, a 1-weighted edge (s, c) is substituted, where c is the child of h containing t. If this edge already exists[7], its weight is increased by 1. A conventional algorithm for weighted 2-layer crossing reduction is applied to this crossing reduction graph. The resulting order of the lower layer is used as an order for the children of h. Such, the hierarchy tree is traversed.

[7] Which might be the case if c contains more than one successor of h with incident edges,

We use the concept of crossing reduction graphs in the implementation part of this thesis. In order to respect the second rule, Forster proposes two alternative methods. In the first, a 2-layer crossing reduction method with constraint support is employed. Constraints are added between nodes with the same parent in the layer hierarchy tree. The alternative method is called *heavy edge method*. It does not call for a crossing reduction method with constraint support, but assigns very high weight to edges incident to a compound graph. The author points out that this method has the drawback that it does not always yield acceptable results when used with a non-optimal reduction algorithm, leading to the possible need of a reordering phase. Furthermore Forster explains that this method leads to less edge-rectangle crossings but more crossings between adjacency edges.

▶ 3.3 Work Presenting an Overview of the Topic

Brockenauer and Cornelsen [31] present a summary of visualization methods for clustered graphs and compound graphs. First, they introduce intrinsic clustering goals like the *Min-Cut k-Way Partition*, minimizing the partition cut weight[8] in a graph with edge-weights, or the *Ratio Cut Partition*, which combines this with cluster size balance. As both problems are NP-hard, heuristics concerning them are cited. Another heuristic is referred to that approaches the goal of clustering with respect to graph connectivity making use of the eigenvalues of the Laplacian matrix of the graph. All heuristics in this section are concerned with a fixed number of clusters, which is different for clustering methods of the following section which perform structural clustering by agglomerative clustering, starting with an n-way partition to iteratively construct a k-way partition. Different properties can be specified to be the critical criterion to be approximated, like vertex degree, diameter or k-edge-connectivity. Some other approaches are recited in compact manner, for example based on successive pattern matching or methods producing unconnected components.

In the following section, the works of Feng, Eades et al. concerning planar drawings of hierarchical clustered graphs [11, 12, 13, 17, 18, 19] are presented. They are based on the notion of c-planar hierarchical clustered graphs, which are defined to be drawable without crossing edges and edge-region-crossings. Another constitutive cognition is the following theorem:

Theorem on the Characterization of c-Planar Graphs: *A connected hierarchical clustered graph $C = (V, E, T)$, with $G = (V, E)$ is c-planar, if and only if there exists a planar drawing of G, such that for each node v of T all vertices of $V - V(v)$ are in the outer face of the drawing of $G(v)$.*

It is used to construct c-planar embeddings. Subgraphs are embedded recursively. The conditions of the theorem are guaranteed by inserting additional vertices and

[8] taking the weight of edges with nodes in different clusters into account

edges for the edges leaving the cluster.
The later works concern straight line drawing with convex shaped clusters, and orthogonal drawings with rectangular shaped clusters. Even a multilevel visualization of clustered graphs is introduced, showing both adjacency and inclusion relations by displaying each level in the inclusion tree in a plane of its own.
Next, the authors introduce the algorithm for hierarchical representation of compound graphs by Sugiyama and Misue [43], see Section 3.2.2 of this thesis for a summary of this work.
Furthermore, the authors refer to some methods for force-directed layout of clustered graphs, reaching from simple introduction of dummy nodes for each cluster over an expanded spring model with differentiated forces for nodes of the same or different clusters and for virtual nodes keeping a cluster together by inserting virtual spring forces of all nodes belonging to that cluster, see also [27]. Another method introduced is that of Wang and Miyamoto [47]. They use metalayouts working on a coarsened structure of the graph, treating the clusters as nodes in one phase of the algorithm in the sense of a divide-and-conquer-approach. Wang and Miyamoto's method of taking layout constraints into account, namely absolute and relative position constraints for vertices, is also presented.
Finally, the authors deal with a method by Eades [10]for the drawing of extremely large graphs, of which only a part can be known at a time.

Sander II [39] is concerned with graph visualization in context with compiler construction, which involves the displaying and interactive navigation of very large graphs. Sander gives an overview over force directed placement and the layered approach in general, together with remarks about their application in compiler construction. With regard to compound graphs, he explains the concept of recursive layout and gives a short summary of the algorithm by Sugiyama and Misue [43] to introduce the nondividing method in comparison. The following part of the paper is devoted to the browsing of large graphs, in which linear views and fisheye views that enlarge special parts of the graph and or distort it are explained.

▶ 3.4 Miscellaneous

Huang et al. Huang *et al.* [27] present a method for *Online Graph Drawing*. This term refers to the drawing of huge graphs that are partially unknown and can only be displayed on screen in tiny parts, such as hyperlink graphs of the websites in the WWW. Though this does not specially refer to the problem of drawing clustered or compound graphs, the authors' method of using a queue of focus-nodes as representation for subgraphs might offer interesting analogies. The thought has not been elaborated for this thesis though. The authors define an exploration of such a graph $G = (V, E)$ as a sequence of logical frames $F_1 = (G_1, Q_1), F_2 = (G_2, Q_2) \ldots$. Each of them consists of a connected subgraph $G_i = (V_i, E_i)$ of G and a queue Q_i of *focus nodes*, which represents the sequence of subgraphs viewed by the user. Additionally the term of *distance-d neighborhood*, denoting the subgraph characterized by a single

node v and a nonnegative integer d, comprising all nodes whose graph-theoretic distance from v is at most d, shortly written $N_d(v)$ and $N(v)$ when $d = 1$. For a queue $Q = (v_1, v_2, \dots v_s)$, the subgraph induced by the union of $N(v_1), N(v_2), \dots N(v_s)$ is called a *logical frame* with its focus nodes $v_1, v_2 \dots v_s$ Each neighborhood is divided in the following two parts:

1. Common part $C(v)$: Graph induced by the nodes of $N(v)$ which also occur in the neighborhood of another focus node:

$$C(v_j) = \sum_{i=1, i \neq j}^{s} N(v_j) \bigcap N(v_i)$$

2. Local part $P(v)$: Graph induced by the nodes of $N(v)$ that do not belong to the neighborhood of any other focus node:

$$P(v_j) = N(v_j) - \sum_{i=1, i \neq j}^{s} N(v_i).$$

A node and its local part form the *local regions*. The authors state that in a drawing of the graph, the local regions should never overlap to help the user follow the addition or deletion of a focus node. The graph is explored by visualizing a sequence of logical frames, of which frame is obtained from its predecessor by the addition of a focus node and its neighborhood—this new focus node is selected by the user—and deletion of at least one other focus node. The operation follows a deletion policy, which can be for example FIFO or the *largest k-distance rule*, which requires the node with the largest graph-theoretical distance from the new focus node to be selected.
The authors suggest an *"in-betweening"* animation technique to preserve the mental map of the user. It consists of a sequence of drawings laid out with a modified spring embedder algorithm, in which the nodes' positions differ only slightly. The goal is to create the appearance of the new focus node sliding slowly towards the center of the page.
In the modified spring embedder, with $F_i = (G_i, Q_i)$ as the current logical frame and $G_i = (V_i, E_i)$, the total force applied on node v is:

$$f(v) = \sum_{u \in N(v)} f_{uv} + \sum_{u \in v_i} g_{uv} + \sum_{u \in Q_i} h_{uv}$$

where f_{uv} is the force exerted on v by the spring between u and v and g_{uv} and h_{uv} are gravitational repulsions exerted on v by one of the other nodes u in F_i. For each of this forces, a strength value can be set to modify their behaviour. The extra gravitational force aims to minimize overlaps among the neighborhoods and keeps the layout of the queue of focus nodes close to a straight line, helping to understand the direction of exploration.
Note that the paper contains a detailed description of the mathematical equations needed for the calculation of an equilibrium force configuration.

▶ 3.5 Summary

Approaches to the drawing of clustered graphs in the force directed and other approaches have been presented as well as layered and force directed methods for the layout of compound graphs. Table 3.1 presents an overview of the approaches in the order of their introduction. Works of summarizing character and works presented in Section 3.4 are omitted.

▶ 3.6 Relations to the Work Presented in this Thesis

The implementational part of this thesis extends the KLay Layered algorithm to layout compound graphs with hierarchy crossing edges. Accordingly, only approaches suitable for this kind of graphs came into view as suitable basic concepts. Additionally the approach proposed had to be of the layered kind. We detected only two approaches that fulfil these requirements, the approach by Sugiyama and Misue [43] and the method of Sander [38]. The comparison of the two shows that one basic choice had to be made, whether local and global layering would match the characteristics of KLay Layered best. The approach of global layering was chosen for its ability to yield compact drawings and for its compatibility with the structures of KLay Layered, especially given by the fact that only one run of the complete algorithm is needed, see also Section 5.1. Another argument is that the approach of Sugiyama and Misue does not support descendant edges, while the method of Sander does.

The layout of compound graphs in KLay Layered does not consist of a simple implementation of Sander's paper though. The initial point is not Sander's algorithm, but the KLay Layered algorithm for flat directed graphs. However this would not be possible without the flat representation of the compound graph with the help of dummy nodes, which is according to Sander's idea. We extended the representation by introducing dummy nodes that serve to represent ports and thus broaden the application range of the algorithm to include port containing compound graphs. I am not aware of any algorithm but KLay Layered using this expanded concept.

I did not adopt the first two phases, the cycle breaking phase and the layering itself, from the approach of Sander but took them over from the KLay Layered algorithm for flat graphs without alteration. This induces two requirements. The first is the insertion of dummy edges for the layering phase, which is mostly done during graph import. The directed dummy edges reflect the desired assignment order for the layering phase and express an "should be put in a layer left of"-relationship between two nodes. The second addition consists of a preprocessing step of the cycle breaking phase that dissolves cyclic dependencies of compound nodes, which cannot be handled by an arbitrary cycle breaking heuristic. The concept of the cycle removal graph is introduced, which simplifies the information of compound node adjacency dependencies by propagating the adjacency edges to the outermost compound nodes involved. While the notion of propagating edges up to ancestors of the source and/or target nodes is not new—it is for example used by Bertault and Miller

authors	*graphs*	*approach*	*specifics, tool*
Wang, Miyamoto	clustered, undirected	force	constraints +, slow, quality constraint-dependent, LYCA
Eades, Huang	clustered, undirected	force	change animation, cluster overlaps -, DA-TU
Frishman, Tal	clustered, undirected	force	edges between clusters allowed, incremental
Bourqui, Auber	clustered, undirected, weighted	force, Voronoi	extends GRIP
Itoh et. al.	multiple category, undirected	force, tree map, space filling	clustering by adjacency + category
Balzer, Deussen	clustered, undirected	implicit surfaces	level-of-detail-views (opacity)
Eades, Feng	clustered, undirected	force or layered	multilevel visualization, skeletons must be triconnected for force approach
Bertault, Miller	compound, directed or undirected	force	hierarchy crossing edges are substituted, nuage
Dogrusoz et al. I	compound, undirected	force	constraints +
Dogrusoz et al. II	compound, undirected	force	constraints +, varying node size +
Sugiyama, Misue I, II	compound, no descendant edges	layered, local layers	fast, drawing space -
Raitner	compound, no descendant edges	layered, local layers	extends Sugiyama, Misue, expanded, contracted views
Sander	compound, directed	layered, global layers	slow , drawing space +
Forster	compound, directed	layered, global layers	enhancement of Sander, edge crossing
This work	compound, directed	layered, global layers	modular, ports (+), slow

Table 3.1. Overview for the presented related work.

in the replacement of hierarchy crossing edges and also by Sugiyama and Misue in the creation of their so called derived graph structure—the concept of cycle removal graphs has not been used before in automatical compound graph drawing as far as I am aware of. Furthermore, to my knowledge, dummy edges have not been used before to enable correct layering of a flat compound graph representation by an arbitrary layering algorithm. The crossing minimization in KLay Layered for compound graphs however follows Sander's approach as far as he proposes a remedy for the intertwining of subgraphs across the layers. The other main challenge in the crossing minimization of compound graphs is to keep nodes of the same subgraph in contiguous sequence in each single layer. This task is not handeled as Sander proposes, but works on the idea of crossing reduction graphs as introduced by Forster [20]. Instead of using Forster's own constraint respecting crossing minimization heuristic however, we implemented the crossing minimization phase structure such that the choice of a crossing minimization heuristic is independent and left interchangeable. The reservation of drawing space for the border segments of compound node drawings again follows Sander's approach [38], which was strongly motivated by the fact that the node placement algorithm of KLay Layered already was an implementation of Sander's method to place node segments in straight horizontal lines [37]. This placement strategy is reused to place dummy nodes and edges marking the side lines of compound nodes. The edge routing of KLay Layered could remain unaltered.

4

At Home in KIELER: The Working Environment

This chapter presents a short introduction to the working environment for the implementational part of this thesis, as it belongs to the KIELER research project[1] and the implementation had to blend in an existing layout algorithm that is part of this project, the KLay Layered algorithm. After we have introduced some additional terms in Section 4.1, Section 4.2 leads into KIELER and its project KLay, while the topic of Section 4.3 is the algorithm KLay Layered, which was to be extended by this work.

▶ 4.1 Terms

As we will concentrate on layer-based layout in the following chapters, we introduce some additional terms related to it. Without loss of generality, we presume in the following that layering takes place from left to right, if not stated otherwise. We assume that the layers are numbered $0, \ldots, n$ from left to right. For a node v that has been assigned to a layer, $l(v)$ denotes the number of the layer, v is assigned to and $ord(v)$ refers to the position of this node in the ordering of the layer, which will only be defined during the node ordering phase. We assume that ordering positions are numbered $0, \ldots, n$ from top to bottom.

▶ 4.2 KIELER and KLay

KIELER is a research project that aims to provide amendments to graphical modeling. It offers subprojects in the domains of the syntax, semantics, and pragmatics of the graphical model-based system design. The KLay project is situated at the heart of the pragmatics section. It is in charge to support one main concept of KIELER, which is to employ automatically generated layout to support the creation and maintenance of diagrams as well as modern dynamic visualization techniques. While an-

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

other subproject, the KIELER Infrastructure for Meta Layout (KIML), deals with the diagram layout specifications on an abstract level, KLay provides the algorithms that compute the concrete layout information. For this purpose, KLay procures a set of different layout algorithms implemented in Java. One of this algorithms is KLay Layered, which is introduced in the next section.

▶ 4.3 KLay Layered

▶ 4.3.1 Architecture

KLay layered is a layer based layout algorithm. Its structure, as introduced by Schulze [41], is modular and consists of the following three kinds of elements.

1. An interface to the users of the algorithm. To support a certain graph format, a module responsible for the import of graph information to the internal data structure of KLay layered and for application of the calculated layout information is requested. Currently, there exist the KGraphImporter and the CompoundKGraphImporter to connect KLay Layered to KIML for flat and compound graphs.

2. The five core phases of the layout algorithm—cycle removal, layer assignment, crossing minimization, node placement, and edge routing. Each of them handles a modular task of the graph layout.

3. Intermediate processors, which can dynamically be inserted into slots before or after the layout phases according to the layout specifications for the actual run. The intermediate processors prepare, influence and postprocess the layout phases, for example by adding or removing dummy nodes or edges.

Figure 4.1 shows this structure.

▶ 4.3.2 Dummy Edges and Dummy Nodes

KLay Layered makes use of dummy edges and dummy nodes. These do not belong to the original graph structure, but are inserted additionally. The reason for the insertion is either that a phase of the algorithm demands a certain quality of the graph structure that can only be provided with supplementary dummy nodes and/or edges, or that a certain effect can be provoked by the insertion. An example for the first case is the creation of a proper layering, i.e. a layering in which no edge spans more than one layer. A layering can be turned into a proper layering by sectioning long edges by the insertion of a dummy node in each layer the edge crosses. Layering itself can be influenced by the insertion of dummy edges. In general it will lead to node u being placed in a layer left of v if we add a dummy edge $d = (u, v)$.

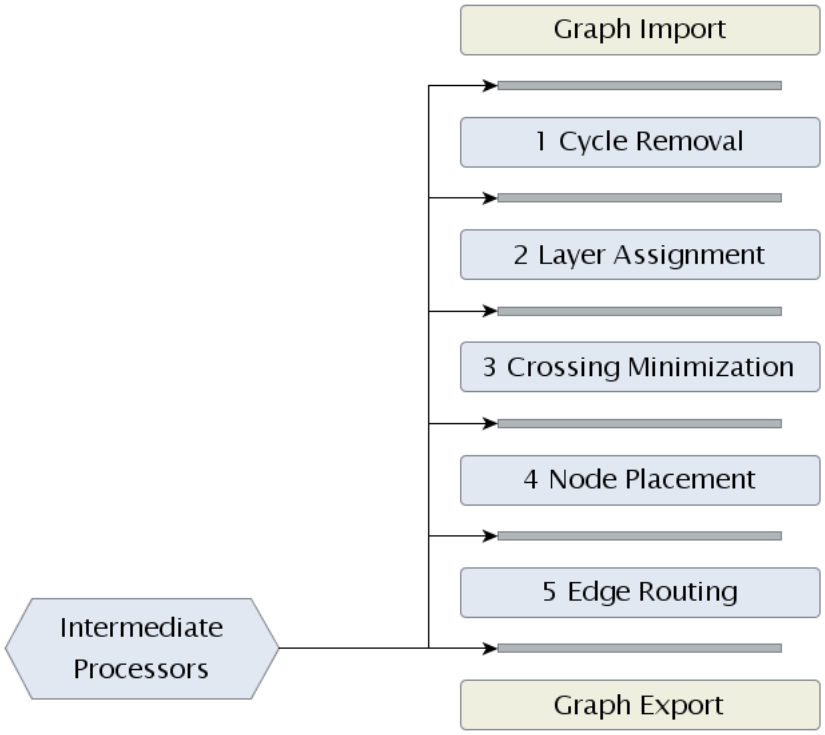


Figure 4.1. The structure of KLay Layered. Figure by [41].

▶ 4.3.3 Phases of the Algorithm

Layers are to be assigned to the nodes in a way that all edges point in the layout direction. Obviously, this is not possible in a cyclic graph. To cope with this problem is the charge of the *cycle removal* phase. It has to guarantee that the graph does not contain any cycles when passed to the actual layer assignment. Each cycle is broken by reversing one of the constituting edges. The optimal choice of the set of edges to be reversed is NP-hard, see Karp [30], so the cycle removal is handled by a heuristic. In contrast to the removal of the edges, which would also render the graph acyclic, the reversion enables us to account for these edges in the following layout phases. It is for example respected in the minimization of edge crossings and it is subject to edge routing. The inversed edge is restored directly before the final layout application by an intermediate processor, called *ReversedEdgeRestorer*.

The *layering* phase assigns nodes to layers. The nodes have to be assigned in a way that successors of a node are placed in a layer to its right. This makes all edges of the graph point to the same direction. Since nodes might carry constraints with regard to the layers they may be placed in, the layering phase in KLay Layered has to be able to respect them or to use a special intermediate processor, the *LayerConstraintHandler*. The layer assignment implies the rough assignment of x-coordinates for the nodes, as the nodes of one layer are placed vertically in line.

The *crossing minimization* or *node ordering* phase strives to calculate an ordering for the nodes in each single layer that minimizes the crossings between adjacency edges. Finding the minimum number of crossings is NP-complete even for two layers, see [29]. This means that a heuristic is to be employed. At the time, KLay Layered uses a barycenter heuristic. This heuristic processes two layers at a time, one of them is regarded as fixed, while the order of the other, called *free* layer, is to be determined. For each node of the free layer, the heuristic takes all nodes of the fixed layer into account that are connected to the node with edges. The average of the position numbers of all these nodes is considered as the nodes' barycenter weight. Finally the heuristic orders the nodes by their barycenter weights. This heuristic is applied in several sweeps across the layers in both directions until the number of edge crossings does not decrease.

During the *node placement* phase, the y-coordinates of the nodes are determined. As dummy nodes that are inserted to split up long edges are placed as well, this phase is relevant for the number of edge bends.

The last core phase is the *edge routing* phase, in which the routing of the edges is determined and the final x coordinates of the nodes are assigned.

▶ 4.3.4 Data Structure

The internal datastructure of KLay layered is the *LayeredGraph*. LayeredGraph, *LNode*, *LEdge*, *Layer*, and *LPort* are the main elements of the LayeredGraph, subclassing *LGraphElement*. In contrast to the data structure of KIELER, the *KGraph*, the graph itself is not represented by an LNode, but by a LayeredGraph. The LayeredGraph owns a list of layerless nodes for nodes that have not been assigned to a Layer and a list of Layers, which themselves each own a list of LNodes. Each LEdge is connected to its nodes by LPorts, there are no portless LayeredGraphs. This means that we have to store additional information to know whether an LPort is a representative of a port in the original graph structure or not. The most interesting difference between the KGraph and the LayeredGraph in the context of this thesis is that the LayeredGraph has no notion of hierarchy. We solve this problem by storing a list of LNodes called CHILDREN and an LGraphElement called PARENT as properties with the LNodes to be able to store inclusion information.

▶ 4.3.5 Intermediate Processors

Intermediate processors pre- or postprocess the core layout phases. Therefore they are dedicated to a special slot before or after one of the layout phases. There can be dependencies to other intermediate processors in the same slot. Furthermore, the intermediate processor is defined by its preconditions and postconditions. Intermediate processors are used in KLay layered for example for port handling, long edge splitting and the treatment of self loops. See Schulze [41] for further information. For the layout of compound graphs, the following five additional intermediate processors are introduced.

1. The *CompoundCycleProcessor*, in charge of solving cyclic dependencies of compound graphs,
2. the *CompoundDummyEdgeRemover*, which removes the dummy edges that have been used for layering,
3. the *CompoundSideProcessor*, inserting dummy nodes and edges to reserve drawing space for upper and lower compound node rectangle sides,
4. the *SubGraphOrderingProcessor* implemented to disentangle subgraphs that have gotten interleaved in the node ordering phase, and
5. the *CompoundGraphRestorer*, used to prepare the final layout application.

5

Layout of Compound Graphs in KLay Layered

The goal of the implementation part of this thesis is to provide the KLay Layered algorithm with the capability to draw compound directed graphs with hierarchy-crossing edges. This general problem was restricted in some respects, as Section 1.3 explains, mainly in the domain of port handling. This chapter explains several tasks that compound graph layout encompasses and debates different approaches to address them. For each task, the approach chosen for implementation is described. First, the basic choice of the layering approach is considered. Next, problems are discussed according to their order of appearance during the algorithm's layout process from the graph import, passing the five phases of the algorithm to the actual layout application. As long as nothing else is declared and without loss of generality, the following explanations are given under the assumption that layering takes place from left to right. We write $l(v)$ for the layer number of the node v with $l(v) \in \mathbb{N}$, if v is assigned to a layer, else $l(v) = \text{undef}$.

▶ 5.1 Choosing Between Global and Local Layering

We selected two approaches as candidates for the general strategy to draw directed compound graphs with the layering approach, the one by Sugiyama and Misue [43] and Sander's alternative solution [38]. These are introduced in Section 3.2.2 and both offer fairly general solutions. The main difference between the two is that while Sugiyama and Misue employ one set of layers per compound node, a method which is called local layering, Sander uses global layering. As discussed in Section 3.2.2, the latter means that the whole compound graph is laid out using one set of layers only. As Forster observes [20], drawing with local layers usually is superior with regard to computation time. On the other hand, global layers allow for more compact drawings.

The global layering method has one further advantage. As the layering takes place on a flat representation of the whole graph and is not performed separately for each compound node, one single pass of the layering phase is sufficient. As it is a goal of this thesis to implement the automatic drawing of compound directed

graphs with KLay Layered in one pass of the whole algorithm and with very little changes to the phases of the algorithm, the approach of Sander is chosen as the more adequate basis for the implementation. It must be noted though that this choice means trading computation speed for system compatibility and tighter arrangement. Please see Section 6.1 for the evaluation of the performance of the implementation.

▶ 5.2 Flat Representation of Compound Graphs

A *flat representation* of the compound graph opens the possibility to draw the graph by the KLay Layered algorithm without the need of passing the algorithm in its entirety more than once. In essence, the graph is represented by a plain directed graph. The key scheme to enable such a representation is the substitution of compound nodes by dummy nodes.

▶ 5.2.1 Dummy Node Representation

Leaf nodes are represented by single LNodes. They are imported as nodes in non-hierarchical layout, which includes the copying of the properties, size and label and the creation of LPorts to represent the node's ports.

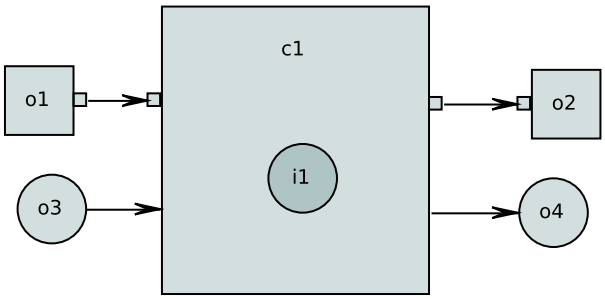
For compound nodes, a single representative cannot be sufficient, because they span more than one layer. Therefore, they are represented by dummy nodes that mark their left and right border. In its flat embodiment, a compound node consists at least of one *Left Compound Border (LCB)* dummy node holding its left border and one *Right Compound Border (RCB)* dummy node on the right border.

All edges ending directly at the compound node without having a target port are connected to the LCB, except for incoming descendant edges. Additionally, we set the LCB as the start node for outgoing descendant edges of the compound node, if they have no source port.

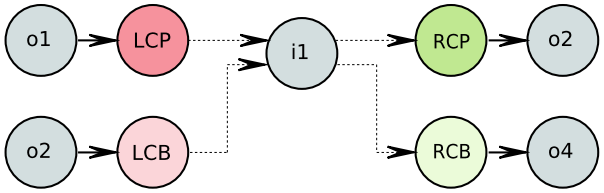
For the edges that originate at the compound node and have no source port, the RCB serves as source node. An exception are outgoing descendant edges. The RCB is the target node for incoming descendant edges that are no target port edges.

Figure 5.1 illustrates the dummy node representation.

Each port in the original graph is represented by a *compound port* dummy node. We connect all edges that are attached to the original port to this dummy node in the flat representation. To date, two port dummy types are implemented: the *LCP* and the *RCP*, which are placed together with the compound border dummies at the left respectively right side of the compound node. As a result, all ports are placed at the left or right side of the compound node in the final drawing, including those that were originally at the north or south side. This is due to the problem restrictions explicated in Section 1.3 and subject to future work. LCPs serve for outgoing descendant source port edges and incoming non-descendant target port edges. RCPs are inserted in case of incoming descendant target port edges or outgoing source port edges that are not descendant edges. Please note that once a representative dummy node for a port is created, it is reused for all edges



(a) A compound graph. Port leaf nodes are drawn as rectangles, other leaf nodes as round shapes.



(b) The internal representation of the compound graph. Dashed lines denote dummy edges. Left and right compound border nodes are labeled as LCB or RCB respectively, left and right compound port as LCP or RCP. Notice that LPorts are not explicitly shown, as the Layered Graph has source and target ports for every edge.

Figure 5.1. A compound graph and its internal flat representation.

connected to the original node, no matter if they are descendant edges and what direction they have. Thus, cases are solved in which a conflict in choosing between an LCP representative and an RCP representative would arise, for example in the case that descendant edges and non-descendant edges of the same direction meet in the same original port. To support a future optimization with respect to meeting port constraints in compound nodes, an LCP is chosen as representative of ports that are situated on the western side in the original graph, while we employ an RCP if the original port side was eastern. An additional advantage of this is that the routing of non-descendant edges will be uncomplicated, as the compound node does not have to be bypassed in the most cases.

We call LCBs and LCPs the *opening dummy nodes*, short $dumO(c)$ of a compound node c's representation and the RCBs and RCPs its *closing dummy nodes*, short $dumC(c)$. We refer to the set of opening and closing dummy nodes of a compound node c as $dum(c)$.

We insert a set of directed *compound dummy edges* D for the representation of compound node C:

$$D := \{(u,v) : ((u \in LCB \ \cup \ LCP) \wedge (v \in child(C))) \\ \vee((u \in child(c)) \wedge (v \in RCB \ \cup \ RCP))\}$$

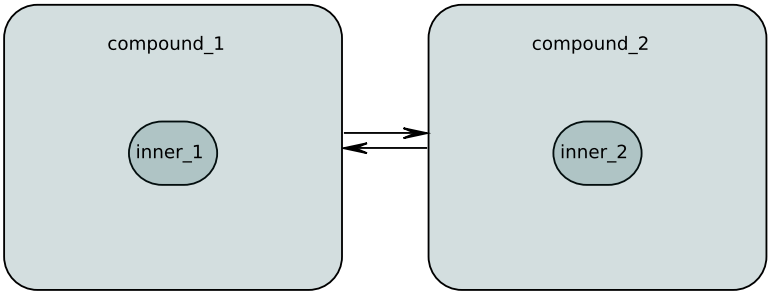
with LCB and RCB denoting the 1-element sets of upper respectively lower compound border dummy nodes representing C and LCP and RCP being the sets of upper respectively lower compound port dummy nodes, each part of the representation of C. This means that each child node of the compound node and each dummy node representing such a child node is connected with the opening dummy nodes of the compound node by an incoming edge and with the closing dummy nodes of the compound node by an outgoing edge. In a strict sense, this insertion is part of the preprocessing of the layering phase, because it is done to keep the representing nodes for C together and in the right order throughout the layering phase. It is presented in this context nevertheless as it is implemented as part of the graph import for reasons of simplicity and performance. The compound dummy edges are removed after the layering phase by an intermediate processor, the *CompoundDummyEdgeRemover*.

Figure 5.1 illustrates in Figure 5.1(b) the internal representation to which the compound graph shown in Figure 5.1(a) is imported for the KLay Layered algorithm. The leaf nodes have single representatives, labeled `o1` to `o4` and `i1`. The compound node `c1` is represented by four dummy nodes. The upper and lower compound border nodes are labeled `LCB` or `RCB` respectively. The portless edges are connected to the compound border nodes, the incoming edge to the upper compound border dummy, the outgoing edge to the lower compound border dummy. For each port of the compound node, a compound port dummy is inserted. The upper compound port node, inserted for the port of the incoming port edge, is labeled `LCP`, the lower compound port node is labeled `RCP` and represents the port of the outgoing port edge.

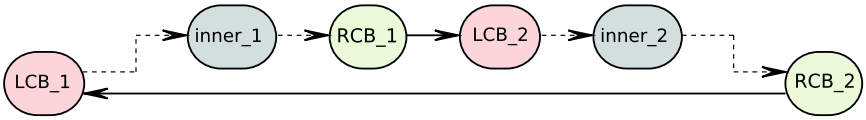
Due to the compound dummy edges, the dummy nodes of a compound graph are placed in different layers. Let i be the number of the leftmost layer in which one of the opening dummy nodes is placed, and j the number of the rightmost layer in which one of the closing dummy nodes is placed. We have to guarantee that the condition $i < j$ holds throughout the layout algorithm execution. We say that the compound node *spans* the layers $i, \ldots, j$.

▶ 5.3 Cyclic Dependencies and Descendant Port-Connected Edges

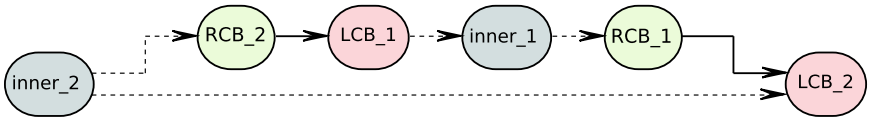
It is a crucial requirement for the drawing of compound nodes from flat representation that the dummy nodes and children of a compound node are layered in the right order, meaning that if $i = l(d_o)$ with $d_o \in dumO(c)$ for compound node c, $j = l(o)$ with $o \in des(c)$ and $k = l(d_c)$ with $d_c \in dumC(c)$, the following order has to be fulfilled: $i < j < k$. As we have seen in Section 5.2.1, this should be provided by the dummy edges inserted throughout the import. Unfortunately these dummy edges may not suffice in the case of a cyclic dependency between two compound nodes. The reason is that in the cycle breaking phase, a compound dummy edge may be chosen to be reverted instead of one of the adjacency edges that constitute the cyclic dependency. An example of this is shown in Figure 5.2. We see a very simple cyclic dependency between two compound nodes in Figure 5.2(a) and the internal flat representation of the graph before the cycle breaking phase in Figure 5.2(b). After the



(a) A simple cyclic dependency between two compound nodes.



(b) The internal representation before the cycle breaking phase. One of the adjacency edges (straight line) should be reverted to resolve the cycle. That would lead to correct layering.



(c) Instead, the cycle breaking heuristic chooses the dummy edge (dashed line) from LCB_2 to inner_2 to be reverted, which will result in the nodes to be layered in the wrong order as implied by the order in the image.

Figure 5.2. Cyclic dependencies in compound nodes.

cycle breaking phase, one of the dummy edges, which are drawn with dashed lines, is chosen by the cycle breaking heuristic to be reverted as shown in Figure 5.2(c). The layering phase will sort the nodes into layers according to a faulty order now. The LCB that is part of the representation of compound node compound_2 will be placed in a layer to the right of the RCB representing the same compound node and as well to the right of the leaf node inner_2. This will prevent compound_2 to be drawn correctly by the KLay Layered algorithm. This is exactly what happens if we run KLay Layered on this graph without special treatment of cyclic dependencies in compound nodes.

We would have a simple solution of this problem at hand, if we were in possession of a cycle breaking heuristic that supports constraints on which edges not to choose for edge reversion during the cycle breaking phase. Setting such a constraint for each compound dummy edge would yield correct layouts.

The GreedyCycleBreaker currently in use in KLay Layered does not support constraints of this nature. But it is worthwhile to take a closer look on what would happen, if we expanded it to respect them. In short, the GreedyCycleBreaker works in the following steps:

1. Remove all sources and sinks from the graph.
2. Chose a node v with $min(indeg(v) - outdeg(v))$.
3. Revert incoming edges of v.
4. Remove v
5. If there are nodes left, go back to the first step.

To enable the GreedyCycleBreaker to correctly handle compound dependencies, the choice of the node in the second step would be restricted to nodes without incoming compound dummy edges. Now we take a look at the example in Figure 5.3. Figure 5.3(a) shows a drawing of two compound graphs with a cyclic dependency. The adjacency edges are on different depth levels. Figure 5.3(b) shows the internal representation in KLay Layered.

An optimal decision for edge reversion would be to revert the edge from RCB_c2 to LCB_c1. For this graph, $min(indeg(v) - outdeg(v))$ is -1, which indeed holds for LCB_c1. Additionally, it has no incoming dummy edges. It follows that it is eligible to be chosen in step two of the cycle breaking heuristic. However, the same conditions hold for LCB_c2. The GreedyCycleBreaker is free to choose any of the two nodes for incoming edge reversion. In the case that LCB_c2 is elected, we cannot but get a suboptimal result in the number of edges reverted, because the graph is still cyclic. Furthermore, as soon as LCB_c2 is removed from the graph according to step one in the second iteration, LCB_m2 becomes eligible for step two in the second iteration, for then it has one incoming and two outgoing edges, the incoming edge being an adjacency edge. Suppose it to get chosen in the second iteration, the graph is still cyclic and a third edge has to be removed to resolve the cycle. So the handling of this constellation by a GreedyCycleBreaker with constraints could lead to three edges being reverted instead of one.

This observation motivated us to provide a preprocessing step that allows for the drawing of compound graphs with cyclic dependencies independently of the cycle breaking strategy currently chosen.

The *CompoundCycleProcessor* expects a layered graph as input and has no further preconditions. It is placed in the slot before phase 1. After processing, the graph contains no more cyclic dependencies between compound nodes. These are removed by edge reversion. Reverted edges are marked for later recovery. Basically, this preprocessor works in three steps.

1. Represent the cyclic dependencies of compound nodes in a *cycle removal graph*, comprising representatives of the edges that form cyclic dependencies.
2. Use an arbitrary cycle breaking heuristic to remove cycles from the cycle removal graph.
3. Revert the set of edges represented by the chosen edges of the cycle removal graph.

(a) Two compound nodes in cyclic dependency.

(b) The internal representation in KLay Layered. Dashed edges are dummy edges. Straight edges are adjacency edges.

(c) The corresponding cycle removal graph.

Figure 5.3. A cyclic dependency with adjacency edges on different depth levels.

The CompoundCycleProcessor creates representatives for adjacency edges, in which source and target nodes are dummies of different compound nodes or belong to different compound nodes. Dummy edges are not considered. To create the according representative, the adjacency edge is propagated up the inclusion tree, which means that source and target are replaced by their parent node until both source and target share the same parent. Formally put, for an edge $e = (u, v)$ the propagated edge is the edge $e_p = (u_p, v_p)$, where

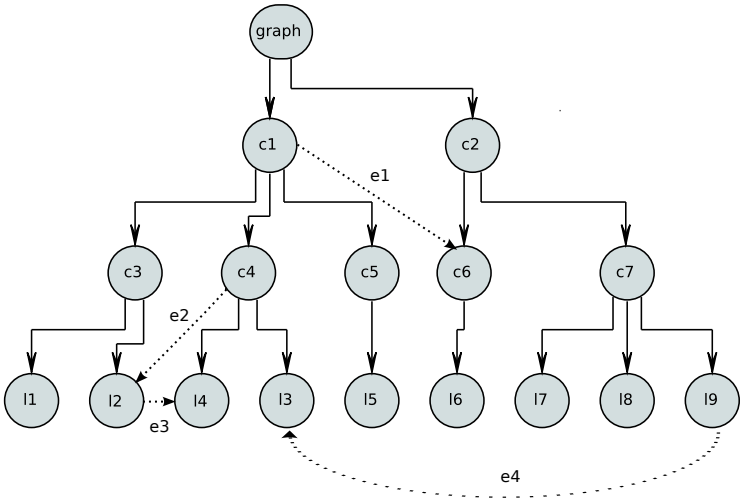
$$(u_p = u \ \vee \ u_p \in anc(u)) \ \wedge \ (v_p = v \ \vee \ v_p \in anc(v)),$$
$$u_p \neq v_p, \ parent(u_p) = parent(v_p).$$

When we refer to the *number of hierarchy levels crossed* by one edge in the following, we mean the sum $(dep(u) - dep(u_p)) + (dep(v) - dep(v_p))$.

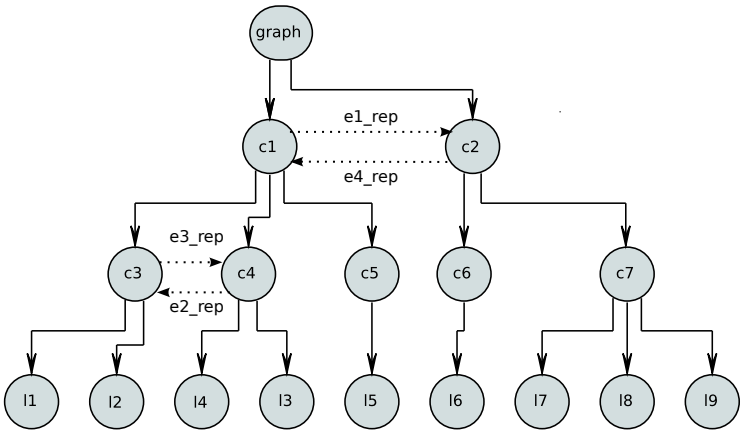
The general notion of propagating edges up the inclusion tree is used by Sugiyama and Misue [43] in the creation of a *derived graph* that is used for the assignment of compound levels to the vertices. In the context of this thesis the reason for the propagation of compound adjacencies is that u_p and v_p are the outermost compound nodes, whose internal representation is affected by the adjacency edge e. Only to consider the outermost compound nodes keeps the cycle removal graph simple and is sufficient to express the cyclic dependency without regard to the depth of the nodes connected by the adjacency edge. After propagating the edge, the CompoundCycleProcessor inserts agent nodes for u_p and v_p into the cycle removal graph as well as an edge between them that serves as representative of the adjacency edge in the cycle removal graph. After insertion of representatives for all concerned adjacency edges and their source and target agents, the cycle removal graph is cyclic if there are cyclic dependencies between compound nodes of the graph to be laid out. Any cycle breaking heuristic can be used to resolve these cycles. At the moment, the greedy cycle breaking heuristic, see Eades et al. [16], is employed in the implementation. It is possible to develop a heuristic specialized on this concrete problem in the future as a small and modular task.

For every edge of the cycle removal graph that is chosen to be reverted by the cycle breaking heuristic, we access the represented adjacency edge of the graph to be laid out and revert it. The original edge direction is restored before drawing with the help of the same intermediate processor that restores the edges reverted during phase 1, the *ReversedEdgeRestorer*. The employment of this intermediate processor is done simply by setting the boolean REVERSED property of the edges concerned. We see the concept of the cycle removal graph illustrated in Figure 5.4. In Figure 5.4(a), the adjacency edges of a compound graph are inserted into its inclusion tree. Figure 5.4(b) shows the propagated edges and the resulting cycle removal graph is pictured in Figure 5.4(c). The suffix `_rep` indicates that the object shown is representing the according graph element in the cycle removal graph. All three representations show that there are cyclic dependencies, so that two of the adjacency edges have to be reverted.

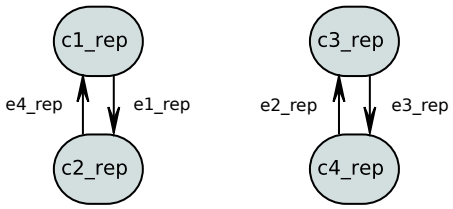
The reverting of edges that are connected to a compound dummy node involves finding and possibly creating the corresponding dummy node on the opposite side



(a) The inclusion tree of a compound graph with inserted adjacency edges (dotted edges).



(b) Representatives of adjacency edges derived by propagating source and target up the inclusion tree until they share the same parent.



(c) The resulting cycle removal graph with edge and node representatives.

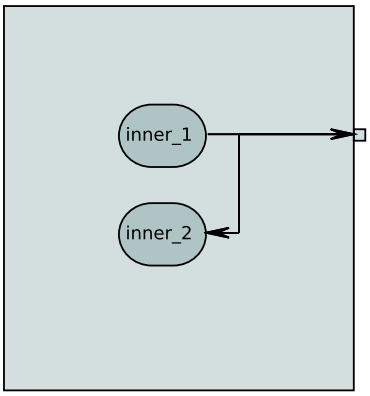
Figure 5.4. Deriving a cycle removal graph from a compound graph by propagating edges up the inclusion tree.

of the compound node. If an edge was connected to the LCB, it has to be connected to the RCB after reversion, if it was connected to an LCP, it changes connection to a RCP and vice versa, including the removal of no longer needed compound port dummy nodes. Please note that in the current implementation this process is not reverted during the restoring of edges before drawing, for the aesthetic appeal of the outcome is more satisfying that way. The reason is that edges do not have to be routed around the shape of the compound node. An example is Figure 5.2(a). The drawing is more readable with the two short edges than it would be with the edge from compound_2 to compound_1 routed around compound_1. In the case that port constraints have to be respected, the original port and connection of the edge have to be restored. As the support of port constraints for compound nodes is not part of this thesis, this is not implemented yet.

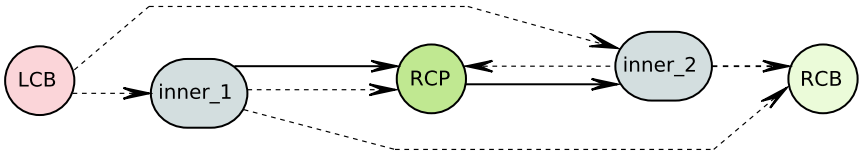
Coming back to the motivating example, we see in Figure 5.3(c) the cycle removal graph of the compound graph in 5.3(a). When we run the GreedyCycleBreaker with this cycle removal graph, it will immediatly chose the right edge to revert as c1 has three outgoing edges but only one incoming adjacency edge, thus being the only node with $min(indeg(v) - outdeg(v))$. Information formerly hidden from the GreedyCycleBreaker—namely that the cyclic dependency is caused by multiple edges in the direction c1 to c2— is now available.

Additionally we will see in Section 5.4, that for the insertion of additional dummy edges needed for the layering phase, the edge propagation process is necessary at the latest, so that the most complex work of the CompoundCycleProcessor cannot be spared even with a constraint respecting cycle breaking phase.

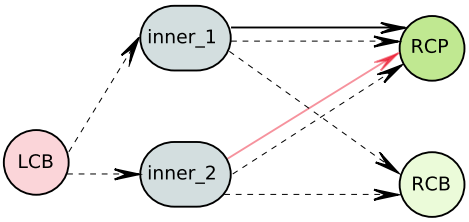
Another constellation that is producing cycles in the graph representation that cannot be solved by a constraint-free cycle breaking heuristic is that of ports with incoming and outgoing descendant edges. Ports of this kind are represented by one single compound port dummy node as explained in Section 5.2.1. This means that the compound port dummy node is connected to descendant nodes by incoming as well as by outgoing edges, possibly leading to its assignment to a layer in the midst of layers occupied by descendant nodes, which would be a faulty assignment. It can be observed that one of the responsible adjacency edges is forming a direct cycle with a compound dummy edge connecting the same compound port dummy and descendant representative as the adjacency edge, only in reverse direction. This adjacency edge should be reverted to solve both, the cycle and the general layering problem as Figure 5.5 shows. As we chose not to introduce constraints to the GreedyCycleBreaker, another solution is chosen here that completes the goal of keeping compound graph drawing independent of the choice of the cycle breaking heuristic. The adjacency edge forming a cycle with a compound dummy edge is reverted during the edge iteration of the CompoundCycleProcessor. More specifically, this is the incoming descendant edge in case of an LCP and the outgoing edge in case of a RCP.



(a) A compound graph with an incoming an an outgoing descendant edge.



(b) The internal representation. Dashed lines are dummy edges. Adjacency edge and dummy edge between the RCP and the descendant leaf node inner_2 form a cycle. If the dummy edge is to be reverted, the nodes will be layered in the faulty order implied by the image.

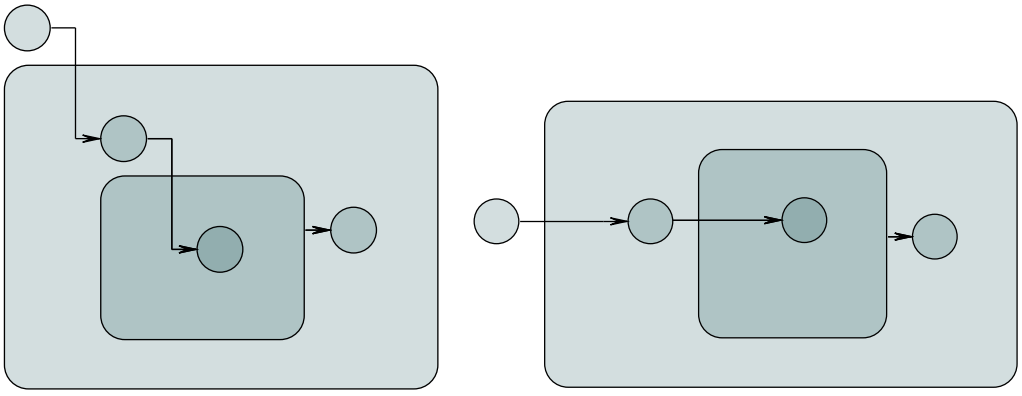


(c) Solving the cycle by reverting the adjacency edge instead allows for correct layering.

Figure 5.5. Cycle problem imposed by drawing ports with incoming and outgoing descendant edges.

▶ 5.4 Layering in a flat Compound Graph Representation

A major requirement in drawing compound graphs with the help of a flat representation is the preservation of the correct order of compound node dummies and compound node descendants. LCB and LCPs have to be layered left of representatives of descendants of the compound node, which themselves have to be placed in layers before the RCB and RCPs. As explained in Sections 5.2.1 and 5.3, the main means of meeting this requirement is the insertion of directed dummy edges that directly represent the constraints by expressing that the source node is to be layered



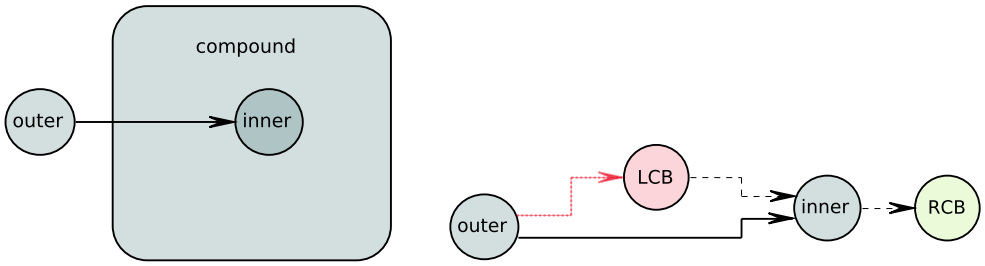
(a) Stair effect without additional dummy edges.

(b) Layout result with additional dummy edges.

Figure 5.6. Solving the stair effect problem for hierarchy crossing edges with the help of additional dummy edges for the layering phase.

left of the target node. The main problems not solved by this approach arise from situations that tempt the cycle breaker to revert one of the dummy edges instead of an adjacency edge to solve cycles in the internal flat graph representation. This is the case, if cyclic dependencies exist between compound nodes or if a port has incoming as well as outgoing descendant edges. These situations are disarmed by the intermediate processor CompoundCycleProcessor before the layering phase as described in Section 5.3. There remains a minor problem that does not affect the correctness of the drawings but their aesthetics and readability by introducing more edge bends than necessary. The problem arises in drawing hierarchy crossing edges and is rooted in the fact that the internal flat representation has no means yet to force the source nodes of such edges to be layered left of the LCB and LCPs of the compound node to which the target node belongs. For this reason we can observe that the source nodes appear at the top or bottom side of the compound node containing the target. As this leads to stair-like edge bends, we call this the *stair effect* of global layering with a flat graph representation. The effect is illustrated by Figure 5.6(a). Both hierarchy crossing edges have to be bent twice, because the source leaf nodes are placed in one layer with the opening dummy nodes of the compound nodes. Though this improves the compactness of the final drawing, it is not desirable, because the edge bends affect the readability. Drawings based on global layering tend to be compact in general for the reason that compound nodes are allowed to span multiple layers instead of spreading the width of one layer. We already profit from this advantage, so there is no need to trade readability for drawing space. The stair effect is therefore eliminated by the insertion of additional dummy nodes for the layering phase.

Figure 5.7 illustrates this process. It shows a very simple graph with a hierarchy crossing edge in Figure 5.7(a) for which the appearance of the stair effect would



(a) Directed compound graph with hierarchy crossing edge.

(b) Internal representation with inserted additional dummy edge (dotted). The straight line is the hierarchy crossing edge, dashed lines are dummy edges inserted during graph import.

Figure 5.7. Additional dummy edge inserted in the case of a hierarchy crossing edge.

be very irritating. To avoid this, we insert an additional dummy edge, which is displayed in Figure 5.7(b).

The insertion of this dummy edge results in the leaf node labeled outer to be put in a layer on the left of the LCB that is part of the representation of the compound node. The drawing in Figure 5.6(b) is laid out with the help of according dummy edges and does not show the stair-like bends, since source nodes are put left of the compound nodes instead of at their sides.

An interesting aspect is the insertion of dummy edges for hierarchy crossing edges that cross more than one hierarchy level. See Section 5.3 on how the number of hierarchy levels crossed is defined. In this case it is important to find the two distinct compound nodes that contain—or are identical with—source and target node and have the same parent node. We call them *source containing* and *target containing* compound node. Now we insert a dummy edge between the closing dummy nodes of the source containing compound node and the LCB of the target containing dummy node. This serves to guarantee that two compound nodes of the same depth connected by a hierarchy crossing edge are not placed side by side but span disjunct layers. Figure 5.8 illustrates the compound node level for which the layering constraint is to be inserted. We insert a dummy edge from the RCB of compound_1 to the LCB of compound_2 to realize this constraint.

We observe that the problem is closely related to propagating adjacency edges for the building of the cycle removal graph as described in Section 5.3. In view of the logical structure of KLay layered, the task of inserting additional dummy edges is one to be handled by an intermediate processor placed before phase 2, but it is striking that the propagation is already performed for each hierarchy crossing adjacency edge by the CompoundCycleProcessor, so that it is easily possible to let this intermediate processor handle the insertion of the additional dummy edges as well. Otherwise the same work would be done twice. For better performance, the dummy edge insertion is implemented within the CompoundCycleProcessor.

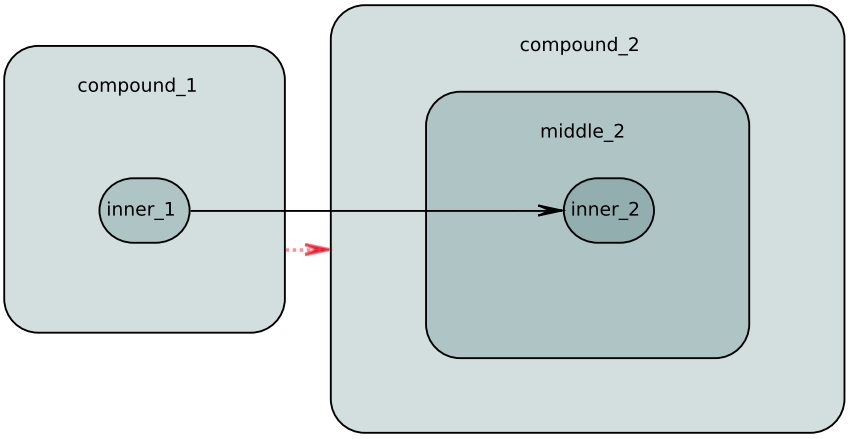


Figure 5.8. Eliminating the stair effect for edges crossing more than one level of hierarchy. The thick dotted edge illustrates the requested constraint for the layering phase.

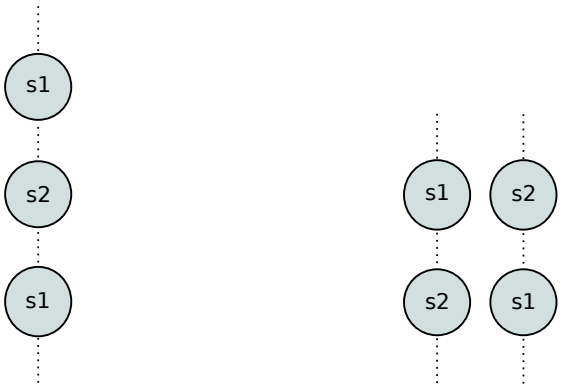
▶ 5.5 Node Ordering

The special demands compound graph drawing imposes on the node ordering phase can be stated as two restrictions that are not to be violated in the crossing minimization. Those restrictions are introduced in Section 5.5.1. The following sections present different possible approaches to meeting these restrictions and the actual implementation to KLay layered.

▶ 5.5.1 The Two Restrictions for Node Ordering in Compound Graphs

To reuse the crossing minimization phase of an algorithm for simple directed graphs for the layout of compound directed graphs might lead to a node ordering that prevents the compound nodes to be drawn correctly. There are two main restrictions that are essential. The first, in the following referred to as *restriction A*, focuses the single layer and demands that all nodes that belong to a subgraph or the representation of one of its nodes or edges have to be put in an unbroken sequence. Please note that we will regard long edge dummy nodes representing a hierarchy crossing edge to belong to the compound node containing the target and all subgraphs to which the target belongs. Thus restriction A means that nodes assigned to different compound nodes are not allowed to be mixed up within the layer.

The second restriction, which we call *restriction B* in the following, refers to the relative ordering of nodes of different subgraphs across all layers and claims that this ordering must be the same on all layers. This means that nodes of one subgraph must not be placed above nodes of another subgraph in one layer and below nodes of this other graph in another layer. In other words, subgraphs must not be interleaved across the layers. The reason for each of the restrictions is that its violation would render it impossible to surround all nodes of the given compound node by a bounding rectangle while leaving other nodes outside.



(a) Violation of restriction A. A broken sequence of subgraph nodes in one layer.

(b) Violation of restriction B. Intertwining of two subgraphs across the layers.

Figure 5.9. Violation of the two restrictions of node ordering in compound graphs. s1 and s2 denote nodes of different subgraphs.

Figure 5.9 illustrates both restrictions. Part 5.9(a) signifies a violation of restriction A. Nodes labeled s1 are of a different subgraph than the node labeled s2. As the two nodes of subgraph s1 are not placed in sequence, restriction A is violated and there is no way of drawing a bounding rectangle around subgraph s1 excluding subgraph s2. Figure 5.9(b) shows the meaning of restriction B. The two subgraphs s1 and s2 do not have the same order on both layers. In the left layer, the node of s1 is placed above the node of s2, while in the right layer, a node of s2 is placed above one of s1. None of the subgraphs could be surrounded by a bounding rectangle in the final drawing excluding each other.

▶ 5.5.2 Keeping Compound Node Contents in Contiguous Sequences

The first requirement to be met is to prohibit nodes belonging to different subgraphs to mixed up in one layer. It is important to keep in mind that this restriction is to be regarded with respect to hierarchy, which means that nodes belonging to the subgraph given by a compound node C have to be kept in an unbroken sequence, but the same condition is to hold for subgraphs given by any of its descendants. A result of this is that, with the exception of the highest depth level, we have to regard ordering node sequences instead of only single nodes. In this section first we explore an approach that was taken into account for solving this problem for KLay Layered, but not chosen in the end due to its restrictiveness. Then the approach by Sander is introduced, which was considered as well. Furthermore the basic idea for the current implementation is explained, which follows a method from Forster. Finally, some details about the implementation and a new class structure that resulted for the node ordering phase of KLay Layered are given.

A Road Not Taken: Constraint-Based Approach

One of my first ideas for meeting restriction A was to impose constraints on the node ordering phase of the algorithm that would prevent the nodes of a subgraph from leaving the boundaries of its sequence.

If we want to implement such an approach, additional dummy nodes that mark those boundaries would be a helpful device. Assuming that for each subgraph there is an opening sequence dummy node in the layer as well as a closing one, it is possible to express the constraint: For each node v that is part of subgraph S with opening border dummy node o_s and closing border dummy node c_s condition

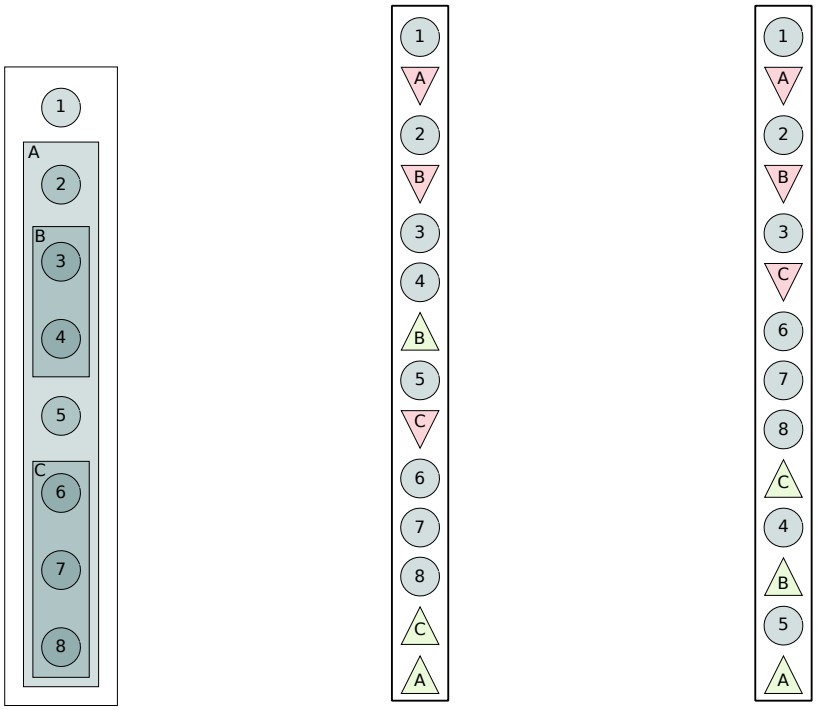
$$ord(o_s) < ord(v) < ord(c_s) \tag{5.1}$$

must hold.

The concept of using sequence dummy nodes is illustrated in Figure 5.10. Figure 5.10(a) shows a layer with eight leaf nodes. Leaf nodes two to eight belong to the subgraph given by compound node A, three and four also to the one of compound node B, and six to eight also to the subgraph induced by compound node C. Figure 5.10 shows the representation of compound node sequences with opening and closing sequence dummy nodes. In this representation, condition 5.1 holds. We can see however that this condition is not sufficient to guarantee the fulfillment of restriction A. If we take a look at another ordering of the same nodes as presented in Figure 5.10(c),we observe that constraint 5.1 still holds. Additionally, the sequences of the compound nodes A and C are still contiguous. In contrast, the sequence of compound node B is broken now by the insertion of the full sequence of compound node C. To solve this problem, another constraint would have to be introduced, which prohibits the faulty nesting of compound node sequences. In the example, a constraint such as

$$ord(o_C) < ord(o_B) \tag{5.2}$$

together with the constraint 5.1 would lead to a node ordering that fulfills restriction A. The problem with constraint 5.2 is that it fixes the order of the compound node sequences for compound node B and C. This implies that restriction A cannot be met by inserting constraints of this kind for the node ordering phase without at least fixing the order of sequences belonging to compound nodes that are siblings. Even their leaf-node siblings have to be included in the ordering constraints, which can be understood by examination of a node ordering in our example sequence, in which node 1 is moved within the sequence of compound node A: while constraint 5.1 is not violated, restriction A is. The only nodes that could be swapped in order freely would be leaf node siblings sharing the same constraint relations concerning their compound node siblings. This would handicap the crossing minimizer to an unacceptable extent. For this reason, we decided to drop the constraint based approach.



(a) A layer with three compound nodes, A, . . . , C, and eight leaf nodes, 1, . . . , 8. Compound nodes are drawn as rectangles. Leaf nodes are part of the subgraphs given by compound nodes that surround them.

(b) Compound nodes represented by opening (triangle on peak) and closing (triangle on base) dummy nodes. Condition 5.1 holds.

(c) Another order of the nodes. Constraint 5.1 still is fulfilled for all compound nodes, but restriction A is violated: The sequence of compound node B is broken.

Figure 5.10. Sequence based approach: Using sequence dummies to represent sequence borders of compound nodes.

Sander's Approach: Reordering according to Average Position

The idea of Sander [38] (for an introduction of his complete approach see Section 3.2.2) is to correct each step of barycenter oriented crossing minimization by resorting the layer according to average position weights of the subgraphs. For this purpose, only the nodes of one layer are taken into account. For this layer, a *reduced nesting tree* T' is built. This means that the inclusion tree is reduced to the nodes present in this layer and their ancestors up to their first common one. The building of T' is illustrated in Figure 5.11(b). Here, T' of the upper layer shown in Figure 5.11(a) is pictured.

Position weights $P(v)$ are denoted for the leaf nodes v of the reduced nesting tree. The weights are identical with the actual node positions in the layer. From these

weights, average position weights for the subgraphs are calculated. The average position weight $P_i(u)$ of subgraph u for layer i is calculated as:

$$P_i(u) = \frac{1}{|\{v \in V_L | u \rightarrow^*_{T'}\}|} \sum_{v \in V_L | u \rightarrow^*_{T'}} P(v)$$

with V_L denoting the set of nodes placed in this layer. In Figure 5.11(b) we can see the (average) position weights denoted on the reduced nesting tree of the example. The subgraphs in the layer and their belonging nodes are now reordered according to ascending average position weight. The result for our exaple layer is shown in Figure 5.11(c).

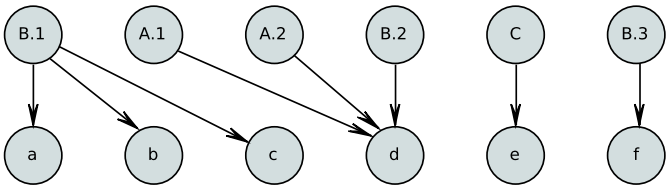
As we can observe by a comparison of this outcome and another reordering, which is both meeting restriction A and optimal in the number of edge crossings, pictured in Figure 5.11(d), this method tends to introduce unnecessary edge crossings. This was already observed by Forster [20]. In the example, four out of seven edge crossings are not necessary. To truly estimate the effect of this disadvantage, it might be rewarding to take a look at the behaviour of the whole algorithm with respect to more than one iteration. As we know, so far this has not been done by any critic of the approach. It might also be interesting to observe, whether another strategy of weight calculation, perhaps taking edge-induced barycenter values into account, would enhance the method.

We did not choose to follow this approach however, because in any case it is very intrusive with respect to the node ordering algorithm itself.

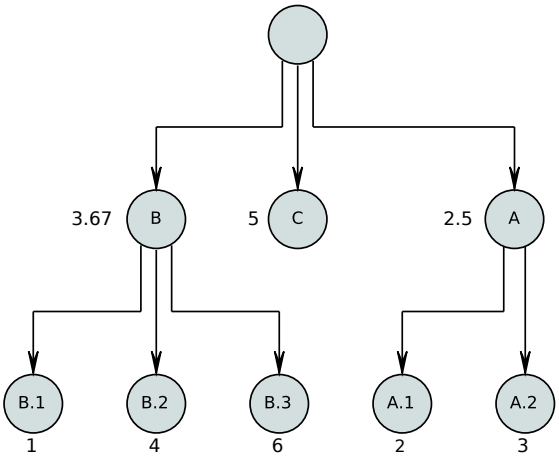
Forster's Approach: Crossing Reduction Graphs

Another approach to respect the hierarchy within one layer is that of Forster [20], see also Section 3.2.2. A main contribution of this work is the observation that computing the child order of all compound nodes independently does not result in losing quality. In consequence of that finding, Forster proposes to serve the nodes in accordant portions to a conventional algorithm for weighted 2-layer crossing reduction. As a means to handle that task, he introduces the Crossing Reduction Graph (CRG).

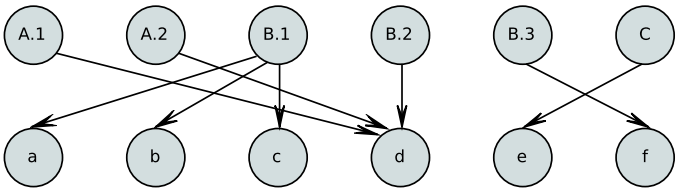
A CRG is a representation of two layers and is created for each compound node. In the fixed layer, the nodes remain unaltered. Let C be the compound node, for which the CRG is created. From the free layer, all nodes that are no descendants of C are removed as well as their incoming edges. For each compound child of C that is represented in the layer, all descendants are combined to form one single node that is to represent the compound child. All edges that were connected to the single descendants of the child are now connected to the new node. If multiple edges result, this is expressed by according edge weights. Now for each child of C there is at most one node in the free layer. The creation of the CRG is illustrated in Figure 5.12, which shows two layers of nodes with related compound nodes for the lower layer in Figure 5.12(a). In Figure 5.12(b) we see the CRG for the grey compound node.



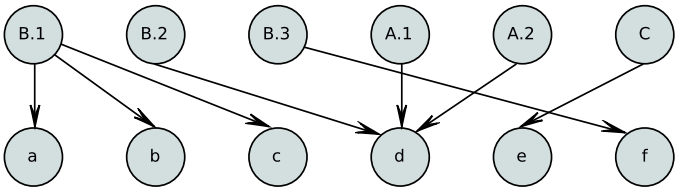
(a) Situation in two layers after a step of barycenter oriented reordering. We consider the upper layer. There are no edge crossings, but the nodes of different subgraphs are mixed up (Subgraphs indicated by labels A to C).



(b) The reduced nesting tree of the upper layer with annotated (average) position weights.

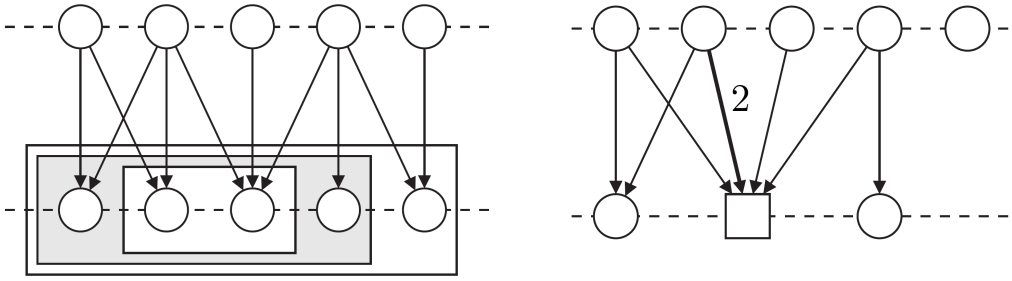


(c) The layers after resorting the upper layer according to average position weights of the subgraphs. Several edge crossings result, of which four are unnecessary as figure 5.11(d) shows.



(d) Optimal result of resorting the upper layer to meet restriction A.

Figure 5.11. Reordering of one layer according to average position to solve restriction A. The layers are displayed top to bottom instead of left to right for better readability.



(a) A clustered graph. (b) The CRG of the grey node.

Figure 5.12. Creating the CRG. Figure by [20]

If the CRG of a compound node is fed to a conventional algorithm for weighted 2-layer crossing reduction, the outcome can be used as the order for the children of C. This way, a child order for each compound node in the inclusion tree can be computed and the order for the layer nodes can be determined by traversing the inclusion tree.

Implementation

We chose Forster's approach, see Section 5.5.2, as the basic idea for our implementation. Two main reasons for this were that it does not yield the danger of loosing quality of crossing reduction and that it is uninvasive with regard to the crossing minimization heuristic itself. Implementing this approach does not hurt the modularity of KLay Layered in this respect. The crossing minimization heuristic can still freely be interchanged, given the expression of multiple edges does not alter. The fundamentals of the implementation were created by providing a new structure of the node ordering phase of KLay Layered that introduces the *CompoundGraphLayerCrossingMinimizer (CGCM)*, which implements the actual crossing minimization step for a given free layer. If the graph to be laid out is a compound graph, it creates CRGs for the compound nodes and hands them to the crossing minimization heuristic to calculate the seperate child orders, working its way from the innermost compound nodes to the outermost ones.

Figure 5.13 shows the structure of the node ordering phase. The class *LayerSweepCrossingMinimizer* is used to implement the performance of layer sweeps. It handles the layer iteration and the counting of edge crossings and uses a class that implements the interface *IPortDistributor* to determine the distribution of ports. The standard implementation of the IPortDistributor is the *NodeRelativeDistributor*, see Spönemann [42]. The LayerSweepCrossingMinimizer does not perform or even organize the actual crossing minimization however, but uses a *CGCM*, which manages the crossing minimization. For that purpose, the CGCM chooses an *ICrossingMinimizationheuristic*, of which the *Barycenterheuristic*, see Spönemann [42] and Schulze [41], is the standard implementation. The Barycenterheuristic does not handle con-

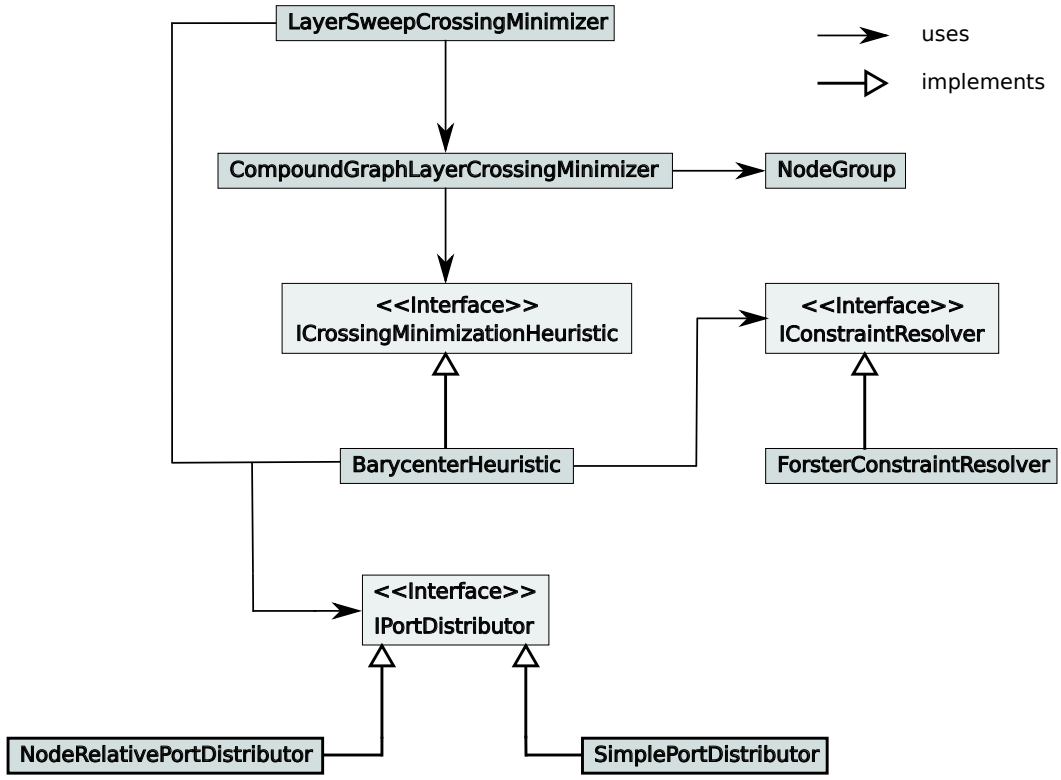


Figure 5.13. Architecture of the KLay Layered node ordering phase implementation.

straints by itself but uses an *IConstraintResolver* to resolve them. The interface is implemented by the ForsterConstraintResolver, see Schulze [41]. To calculate the node barycenters, the BarycenterHeuristic takes port ranks into account, see Spönemann [42], for which it uses an IPortDistributor.

It follows that for the implementation of compound graph node ordering, no changes to the actual layer sweep and to the crossing minimization heuristic have to be introduced. The additional means is added between the two as a kind of middle management deciding on which nodes and edges of the two layers to pass on in which order to the heuristic. The class CGCM is the only part of code working on the notion of hierarchy in the node ordering phase, while all others are left unaltered in their essence.

To build the CRGs, the CGCM makes use of the *NodeGroup* class, formerly called *Vertex*, see Schulze [41]. A NodeGroup can contain one or more nodes and is used in the node ordering phase to model sets of nodes that are placed next to each other. Now it is employed as well to envelop the child nodes of one compound graph to be able to express it as a single node group in the CRG of its parent. The class offers two constructors, one to generate a node group containing one single node, and another that merges two given NodeGroups into a new one. The latter

constructor also handles the weight, and barycenter calculation of the NodeGroup. The LayerSweepCrossingMinimizer already builds a single-node-NodeGroup for each node in the free layer and hands an according Map down to the CGCM. When nodes are grouped to represent a compound node in the CRG of its parent, the single-node-Node-Groups are merged by the CGCM to form a new NodeGroup. The order of the nodes will be preserved by this action in the nodes list of the NodeGroup. The CGCM passes a list of NodeGroups to the crossing minimization heuristic and will receive it back with the new ordering of NodeGroups. The CGCM exerts its only impact on the layout, if a compound graph is to be placed and the free layer contains any nodes. In the case of an empty free layer its main method would return directly and in case of a non-compound graph, it will pass the layer unaltered to the crossing minimization heuristic.

If the layered graph is a compound graph, two datastructures are essential to the algorithm. The first is the compoundNodesMap, in which for each compound node the layer nodes that are its children or its dummy nodes, together called *content* in the following, are stored in a list, represented by NodeGroups. It follows that in the beginning, the lists contain only single node NodeGroups. In order to build up the Map, the related compound nodes for nodes of the free layer are identified. This is the parent for leave nodes, the related compound node for dummy nodes. In the processing, we preserve the order of processed compound nodes in a list called compoundNodesMapKeys for to procure a documentation of the order of appearance.

The second data structure is a list of LNode lists called compoundNodesPerDepth-Level. This list has one NodeList for each depth level of the graph. At the beginning, we sort all compound nodes of the compoundNodesMapKeys into the lists indexed by their depth level. Now compoundNodesPerDepthLevel contains all compound nodes that are represented by own dummy nodes or by child leaf nodes in the layer.

The building and processing of CRGs takes place from the leafs of the inclusion tree up to the root. To manage this, the CGCM iterates the lists of compoundNodes-PerDepthLevel from the highest index to the lowest. For each compoundNode in the actual list, the content taken from the compoundNodesMap is passed to the crossing minimization heuristic.

Preserving the achieved order, the nodes are afterwards merged to form a node group, which is inserted into the content of the ancestor from the next depth level. This ancestor itself is inserted into the list of compoundNodesPerDepthLevel that is indexed by its depth. When the root of the inclusion tree is reached, the order of the nodes can be extracted from the remaining NodeGroups.

The pseudocode for the CGCM is shown below.

Listing 5.1. CompoundLayout

```
1 initialize crossing minimization heuristic: heuristic;
2 merge := NodeGroup.list -> Nodegroup
3 if layer = ∅ then
4     return;
```

```
5  if graph is not compound then
6      call heuristic(layer NodeGroups);
7      apply order;
8  else do
9      CNM := LNode -> NodeGroup-list;
10     K := LNode-list;
11     for each node V in layer do
12         C := parent(V);      // or related compound for dummy nodes
13         append V to CNM(C);
14         append C to K;
15     CNPDL := LNode-lists-list;
16     for each depth level i do
17         DL_i := LNode-list;
18         store DL_i at CNPDL(i);
19     for each node N in K
20         append N to CNPDL(dep(N));
21     while CNPDL ! = ∅ do
22         L := last element of CNPDL;
23         remove last element of CNPDL;
24         for each node P of L do
25             compoundContent := CNM(P);
26             call heuristic(compoundContent);
27             if P ! =root then
28                 NG := merge(compoundContent);
29                 append NG to CNM(parent(P));
30                 append parent(P) to CNPDL(dep(parent(P));
31             else do
32                 apply order;
```

▶ 5.5.3 Ordering of Subgraphs across the Layers

The CGCM ensures that the contents of different compound nodes are not mixed up in one layer, but it does not preclude that different subgraphs get intertwined across the layers. There are two main approaches to guarantee the compliance with restriction B. One is to respect it in the node ordering heuristic by inserting constraints or manipulating edge weights. The other approach consists of a postprocessing of the node ordering phase, in which the subgraphs are carefully disentangled. Both methods are introduced in the following.

Forster's Approaches: Constraint Method and Heavy Edge Method

Forster [20] aims to respect restriction B right from the start in the node ordering phase. He introduces the constraint method as well as the heavy edge method, both are illustrated in Figure 5.14. The figure shows compound nodes spanning two layers on the left, an illustration of the constraint method in the middle, and it shows the heavy edge method in the drawing to the right. Note that the figures show layers from top to bottom, not left to right. The constraint method depends on a

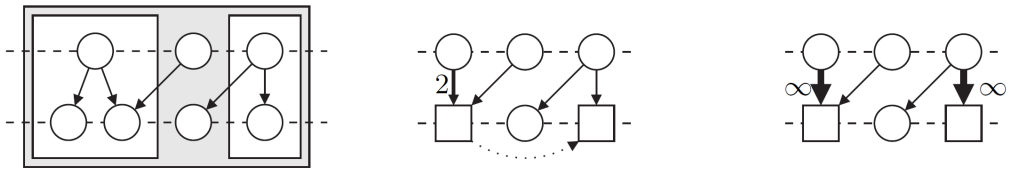


Figure 5.14. Forster's approaches to respecting subgraph ordering across the layers. The left drawing shows compound nodes spanning two layers. The constraint based approach is illustrated in the middle, while the heavy edge method is shown in the right figure. Figure by [20]

2-layer crossing reduction algorithm that supports constraints in the sense of predefined relative orders of some node pairs. Forster himself has proposed a constraint respecting crossing reduction algorithm in a subsequent paper [21], another is introduced by Schreiber [40]. The constraint method inserts constraints into the CRGs. The relative order of two sibling compound nodes in the fixed layer is sustained in the free layer by inserting an according constraint between their representing node groups in the CRG. The constraint is shown in Figure 5.14 as a dotted arc. The problem with this approach is that the order of compound nodes is determined by the first fixed order, which makes the crossing minimization too restrictive. Hence, it is not implemented in KLay Layered. The second proposal of Forster is the heavy edge method, which consists in modeling compound nodes as edges with very high weight. Figure 5.14 shows these edges labeled ∞. Forster explains that this method does not guarantee compliance with restriction B, so that postprocessing may still be necessary. Furthermore he points out that the heavy edge method penalizes adjacency edges that cross compound nodes at the expense of an increased number of adjacency edge crossings. For these reasons, the heavy edge method was not chosen for the KLay Layered implementation.

Sander's Approach: The Subgraph Ordering Graph

Sander [38] introduces the subgraph ordering graph as a means to disentangle subgraphs as a postprocessing of the node ordering phase, to see also Section 3.2.2. We recall the definition of the subgraph ordering graph:
The subgraph ordering graph consists of all nodes of the graph. It contains an edge (w, w'), if there are nodes v, v' and a subgraph u_0 with $l(v) = l(v')$ and $P(v) = P(v') - 1$ and $w \neq w'$ and $u_0 \rightarrow_I^* w \rightarrow_I^* v$ and $u_0 \rightarrow_I^* w' \rightarrow_I^* v'$ with $P(i)$ is the position of the node in the layer. The condition $P(v) = P(v') - 1$ means that Sander wishes to insert edges representing the relation *is left of*. To understand this it is important to keep in mind that Sander thinks of top to bottom layering, not left to right. The Subgraph Ordering Graph is shown in Figure 5.15. We can see in the example that there are disconnected graph components, one for each compound node in the inclusion tree.

There can be an arbitrary number of cycles in each of the components, indicating intertwined subgraphs. To achieve an ordering of the node that satisfies require-

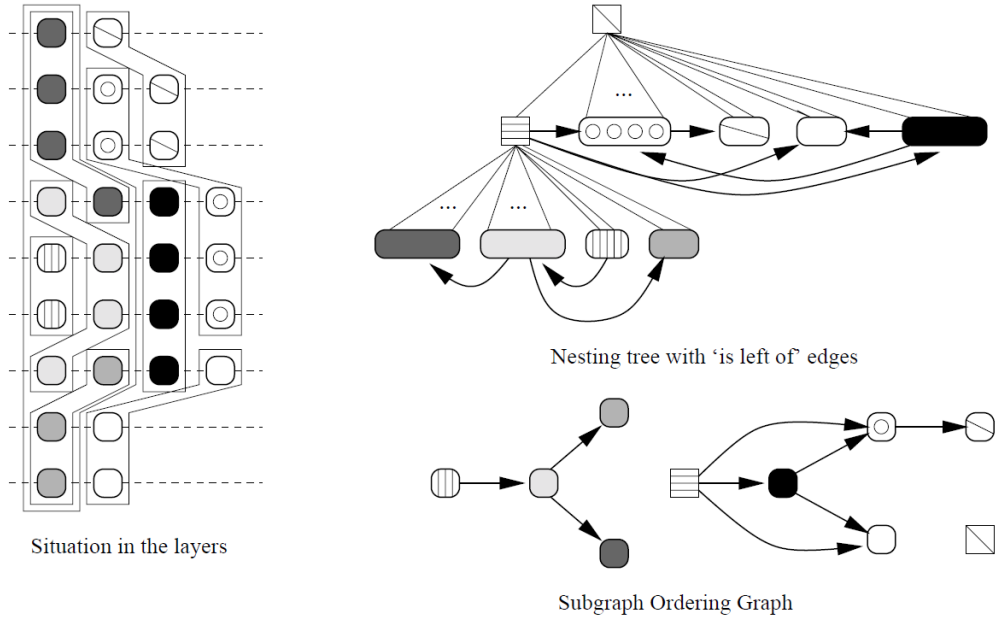


Figure 5.15. The subgraph ordering graph. Figure by [38]

ment B, the cycles in the subgraph ordering graph have to be dissolved. Then a correct node order is given by the topological ordering of the subgraph ordering graph. Sander's approach was chosen for implementation in KLay layered for two reasons. First, it does not affect the node ordering phase itself and is not dependent on heuristics that respect constraints or consider heavy edge weights. Second, its performance with respect to the number of node crossings is dependent on the cycle breaking heuristic employed, so this implementation yields a modular, compact task for future work. At the moment, the GreedyCycleBreaker heuristic is employed, leading to correct results. A future research topic might be to implement and use different heuristics and strive to enhance the layout with regard to the number of node crossings. One approach might be to cut the cycle at the node u with the smallest average position:

$$\overline{P}(u) = \frac{1}{|v \in B | u \rightarrow_I^* v|} \sum_{v \in B, u \rightarrow_I^* v} P(v)$$

with B being the nodes that are no compound nodes and $P(v)$ being the position of node v in the layer, as Sander proposes.

Implementation

The compliance with restriction B is the task of the intermediate processor Subgraph Ordering Processor (SOP) of KLay Layered. It assumes that the node ordering has

taken place and that the nodes in one layer that belong to the same compound node are placed according to restriction A. Its postcondition is that all subgraphs have the same relative order on all layers. The SOP is an implementation of the approach of Sander, hence its basic steps are:

1. Build a subgraph ordering graph.
2. Break the cycles in the subgraph ordering graph.
3. Retrieve the order of the nodes for each layer by a topological sorting of the subgraph ordering graph.

As we assume without loss of generality that layering takes place from left to right, we think of an *is below*-relationship of nodes in the layer rather than of an *is left of*-relationship. We start with an empty subgraph ordering graph, iterate each node in each layer and update the subgraph ordering graph in the case that nodes are neighbors in one layer that are of different compound nodes and are no leaf nodes of the highest depth level. To find the correct graph component to insert the edge, we propagate it in the way described in Section 5.3, until two (compound) nodes with the same parent are reached. Representatives for those nodes are inserted into the subgraphOrderingGraph as well as the connecting directed *is below*-edge. As explained before, the greedy cycle breaking heuristic is employed to remove the cycles from the subgraph ordering graph. If the cycle breaking heuristic detects no cycles, the order of the nodes remains unaltered, because restriction B is already satisfied. In any other case the layers are reordered such that the node ordering is in accordance to the topological sorting of the subgraph ordering graph. During the recursive order application, all compound nodes have to be processed, even if not all nodes have been inserted in the subgraph ordering graph. Merging those nodes into the order given by the subgraph ordering graph has to respect the layer node order determined by the crossing minimization phase as far as possible.

▶ 5.6 Drawing Bounding Rectangles

During the node ordering phase we have respected the restrictions of not intertwining subgraphs across the layers and of keeping nodes of one subgraph in an unbroken sequence with regard to the single layer. This was already a prerequisite to drawing compound nodes as rectangles surrounding the shapes of all the node's descendants, leaving out all other nodes. Furthermore, the opening and closing dummy nodes of the compound node reserve the drawing space needed for the left and right sides of the bounding rectangle plus any additional drawing space needed, as for insets etc. What is not taken care of by these methods is to prepare the drawing of the upper and lower side lines of the bounding rectangle. As we wish to draw straight lines between the subgraphs placed above each other in the layers, we have to enable the node placement phase to arrange the nodes such that drawing space is reserved accordingly. One problem in this is that the subgraphs will in most cases have a

varying number of nodes in the different layers. Without interference, the node placing algorihm would assign node positions that would allow for the drawing of curves rather than for straight side lines. Figures 5.16(a) and 5.16(b) illustrate this problem. As compound node c2 owns one node in the first layer, two in the next and one in the third, special effort has to be taken to ensure that straight side segments of the border rectangles can be drawn between both subgraphs.

▶ 5.6.1 Compound Side Dummy Nodes and Edges

To reserve drawing space for the upper and lower rectangle borders of compound nodes, we insert two *compound side dummy* nodes per layer for each compound node represented in the layer. This is inspired by the approach of Sander [38], who uses dummy nodes for the representation of side segments as well. The nodes are placed at the beginning and end of the contiguous sequences of the compound node's content in the layer. We call a side dummy placed at the beginning of a sequence *upper* compound side dummy and one placed at the end of the compound node's sequence a *lower* compound side dummy. Additionally, directed compound side dummy edges are inserted. If i resp. j is the number of the leftmost resp. rightmost layer compound node c spans, u_k resp. l_k are the upper resp. lower side dummy nodes inserted for c in layer k, then we insert compound side dummy edges such that $u_i, \ldots, u_j$ and $l_i, \ldots, l_j$ are paths in the set of compound side dummy edges. Figure 5.16(c) shows the representations of the two example compound graphs with inserted side dummies and side dummy edges. The insertion of side dummy nodes and edges is performed by an intermediate processor in the slot before the node placement phase, the *CompoundSideProcessor*.

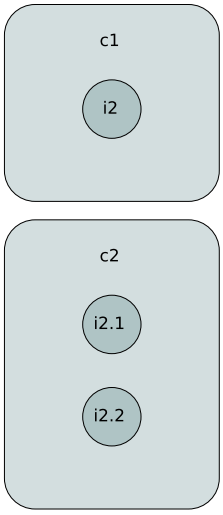
▶ 5.6.2 A Side Job for the LinearSegmentsNodePlacer

Having inserted compound side dummy nodes and edges as explained in Section 5.6.1, we can take advantage of a main ability of the node placement algorithm currently implemented in KLay Layered, the LinearSegmentsNodePlacer. It is an implementation of the algorithm introduced by Sander [37]. Sander defines the term of *linear segments* as a maximal sequence of nodes $w_1, \ldots, w_m$ with:

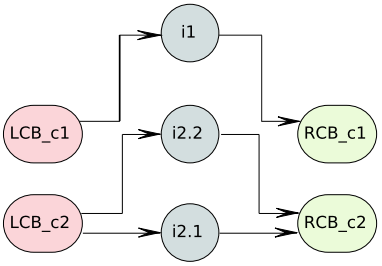
1. $l(w_i) = l(w_{i+1}) - 1$,
2. $indeg(w_1) \leq 1, outdeg(w_m) \leq 1, suc(w_1) = \{w_2\}, pre(w_m) = \{w_{m-1}\}$, and
3. $pre(w_i) = \{w_{i-1}\}$ and $suc(w_i) = \{w_{i+1}\}$ for $i \in 2, \ldots, m-1$.

The algorithm strives to draw linear segments and the edges connecting the nodes as a straight line, horizontal in case of left-to-right layering[1]. The background of this goal is the need to draw long adjacency edges that cross several layers and are split

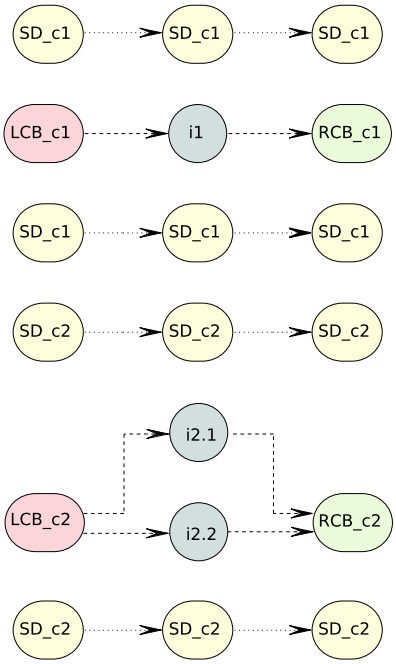
[1] Note however, that Sander's explanations are based on the notion of top-to-bottom-layering, thus referring to "vertical" lines.



(a) Two compound nodes layered above each other.



(b) The internal representation without side dummy nodes and edges.



(c) The internal representation with side dummy nodes and edges. Dotted edges are side dummy edges, dashed edges are compound dummy edges.

Figure 5.16. Drawing bounding rectangles of compound nodes with the help of side dummy segments.

up by dummy nodes to form a proper hierarchy. We simply extend the algorithm to regard the sequences of side dummy nodes as linear segments as well. Sander assigns each node to a linear segment, introducing naive segments with only one node—for nodes that do not have to be placed in line. He requires that edges of different linear segments do not cross after the crossing minimization phase. This prerequisite is easily met by the side dummy sequences of KLay Layered, which are inserted only after the crossing minimization phase in a way that compound side dummy edges do never cross. The algorithm orders the linear segments according to the relation "is below" and then uses a pendulum method Sander introduced in an earlier work [36]. The linear segments are modeled as the ball and the edges between them as the strings of a pendulum. If the uppermost segments are modeled to be fixed on the side opposed to the direction of gravity, the balls on the string swing to a balanced layout driven by their gravity. Gravity is regarded to be working in the direction of the layout, to the right in our case. Neighboured balls are modeled to influence each other to guarantee a minimum spacing between them. If all forces are balanced, the linear segments are placed in straight lines. Thus, the LinearSegmentsNodePlacer places the compound side dummy nodes and edges in straight lines, allowing for rectangle sides to be drawn in their place in the final layout. The minimal spacing guarantees the seperation of rectangle boundings. Additional drawing space, for example concerning insets, can be reserved by influencing the size of the compound side dummy nodes. It is important to notice that if the LinearSegmentsNodePlacer is to be exchanged for another node placing algorithm, this algorithm needs the ability to place the side dummy segments in a straight line.

▶ 5.7 Restoring a Compound Graph from Flat Representation

Before the layout application can be performed, all dummy nodes and edges that are still left in the graph are removed with the exception of the LCB of every compound node that serves as final representative. To prepare this, the size of the compound rectangle is calculated from the positions and size of the leftmost upper and lower side dummy nodes and the rightmost upper side dummy node. Port positions are derived from the position of the according dummy nodes and edges are transferred from these to accordingly created ports of the LCB. This work is done by the intermediate processor *CompoundGraphRestorer*, which is placed in the slot after phase 5 and assumes, that the splitting of long edges has been already been undone.

▶ 5.8 Summary

Drawing compound graphs in KLay Layered is done with the help of a special graph import that translates the compound graph into a flat representation. Cyclic dependencies between compound nodes and special port edge constellation demand preprocessing of the cycle removal phase, during which also special problems of compound graph layering are handled. However the most effort has to be made in the realm of node ordering to keep the children of compound nodes grouped in the

layers and to correct possible intertwining of subgraphs across the layers. For this purpose, a management class had to be inserted into the node crossing phase, which does not affect the central crossing minimization heuristic, but reflects hierarchy in the processing of the layer nodes. Additionally a postprocessing step was inserted to unify the subgraph order across the layers. Further challenges were the reservation of drawing space for the upper and lower side segments of bounding rectangles for compound nodes and restoration of the compound graph from the flat representation after phase 5. With the exeption of the node ordering phase, for which compound graphs need additional processing management, the tasks were handeled without interfering with the central phases of KLay Layered.

6

Evaluation

This chapter is concerned with experiments and tests that have been run on the implementation of compound graph layout in KLay Layered. Section 6.1 is concerned with the results of experiments with the topic of computation time performance, while Section 6.2 discusses the tests of the algorithm extensions and gives some drawing examples.

▶ 6.1 Computation Time

▶ 6.1.1 Hierarchical vs. Recursive Layout of Nested Graphs

In order to explore which effect the layout of hierarchy has on the computation time behaviour, we conducted an experiment to compare the layout of nested graphs by KLay Layered by recursive application or hierarchical layout. The recursive application means the application of KLay Layered for flat graphs to each subgraph recursively. Hierarchical layout refers to the approach presented in this paper. This is no comparison of methods that have the same capabilities, since recursive layout, in contrast to hierarchical layout, simply ignores hierarchy crossing edges. However, these measurements serve as an indication of the additional cost of the drawing of hierarchy crossing edges. The graphs were generated by random, the number of edges was fixed to 40 edges. The number of nodes is ascending from 28 to 246. Figure 6.1 illustrates the results of the experiment.

Due to the random generation, though with fixed specification parameters, the nesting structure of the graphs varies, which is in addition to the relatively small number of graphs the cause for some up- and downturns in the results.

We ran an additional experiment to find out what effect an increase of nesting level itself has on the computation time of hierarchical and recursive layout of nested graphs. We chose a small cyclic graph of five nodes and eight edges shown in Figure 6.2 and determined computation time results of both specifications of KLay Layered, with the graph situated in increasing nesting depth. The result is shown in Figure 6.3. It is obvious that hierarchical layout leads to a considerable increase in layout time. In this experiment, hierarchical layout exceeded the computation time

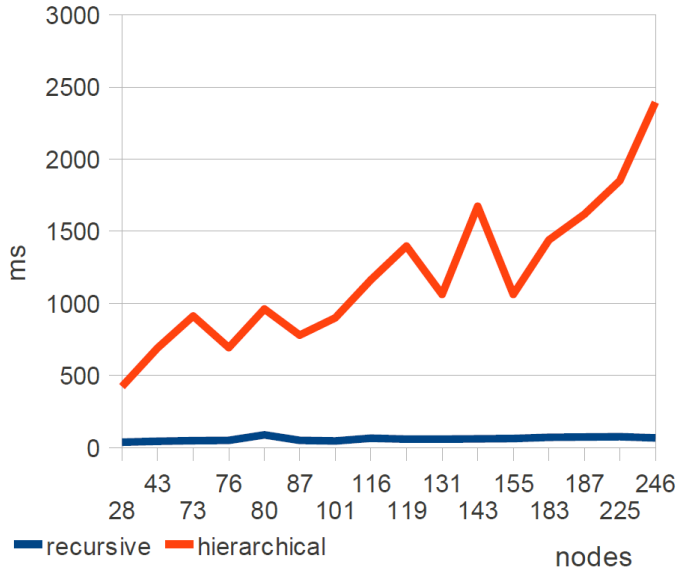


Figure 6.1. Experimental comparison of hierarchical and recursive layout with KLay Layered on nested random graphs. The number of edges is fixed, the number of nodes and the nesting structure varies.

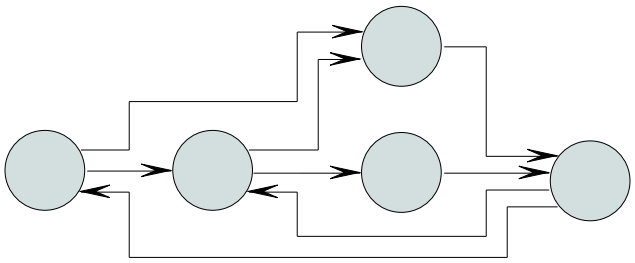


Figure 6.2. The small graph used in the experiment on the effect of increasing nesting depth

of recursive layout application to the single compound nodes by an average factor of 20. This motivated us to take a closer look on the layout times for the single phases and intermediate processors to determine which modules use the most time.

▶ 6.1.2 Computation Time of Submodules

While investigating which components use up what part of the computation time, we were also interested in the differences of computation times for each module, i.e. phases and intermediate layouters, for small and large graphs. This is motivated by the significance of the computation time difference of hierarchical and recursive layout for large graphs. We examined the component layout time of two sample graphs. One was a small graph with ten nodes, four of them compound nodes.

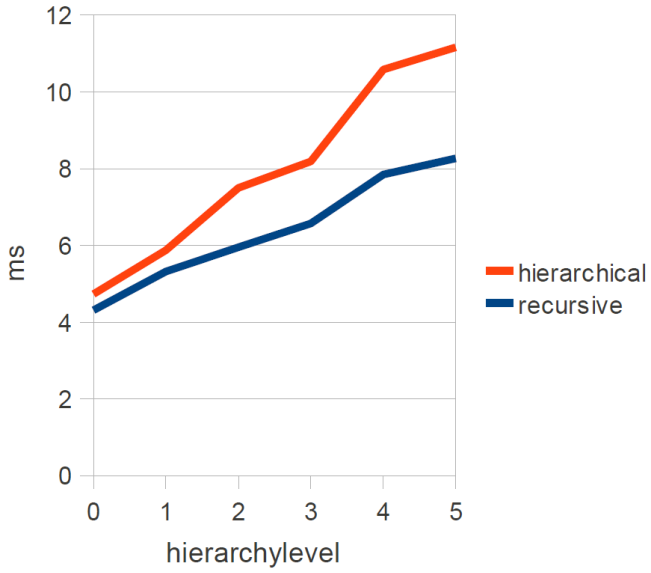


Figure 6.3. The effect of increasing nesting depth on the computation time of hierarchical and recursive layout for a small graph.

The graph comprises three levels of hierarchy and has a set of seven edges, one of them is a hierarchy crossing edge. The second graph consists of 150 nodes and 198 edges, almost all of them are hierarchy crossing. 96 of the nodes are compound nodes, and the graph has 7 levels of hierarchy. Figure 6.4 shows the outcome of the experiment, while Figure 6.5 especially illustrates the partition of computation times. Components that demand a very small amount of computation time are summarized in the drawing. The overall computation time was 613.794 *ms* for the small graph and 14870 *ms* for the large graph. The component computation times for the components in processing order are shown in Table 6.1.

While for the small graph two of the classical layout phases, the edge routing and the crossing minimization, use up the greatest part of the component time, it is a special compound graph drawing module that is predominant for the layout of the large graph. The Subgraph Ordering Processor (SOP) takes up about 83% of the component time. This explains why the computation time differences of recursive and hierarchical layout are more eminent in the domain of large graphs: The specialized module SOP, uniquely used in hierarchy based layout, gains a larger portion of computation time with the increase of nesting level and number of compound nodes. The SOP clearly emerges as bottleneck for the layout time of hierarchical layout in KLay Layered. Optimizing this module can be expected to exert great influence on the computation time behaviour of compound graph layout.

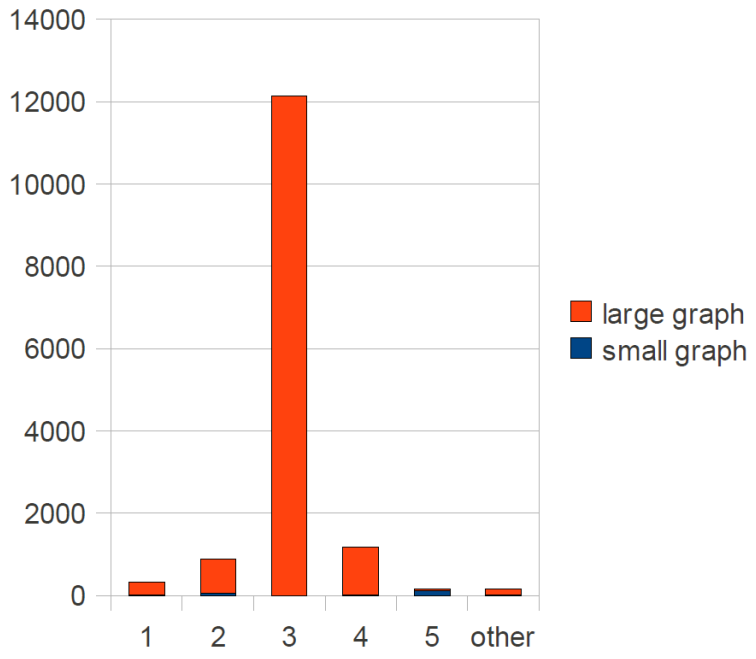
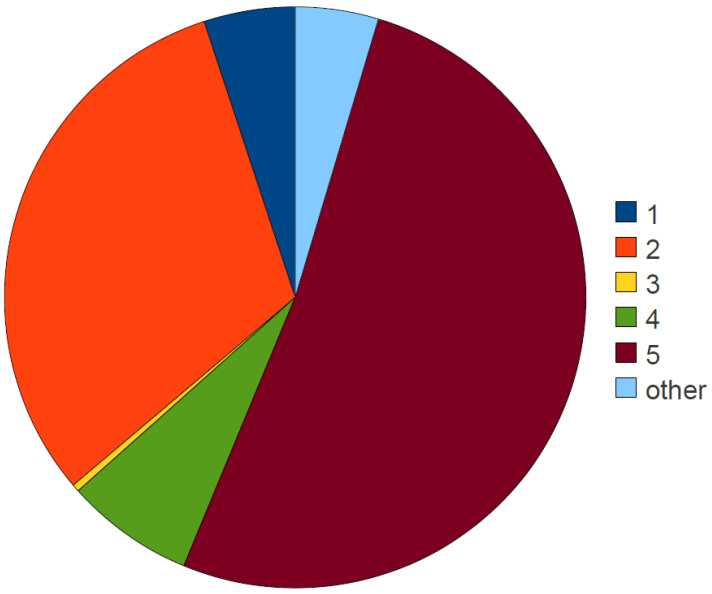


Figure 6.4. Component computation times on large and small graph. Legend: 1 = removal of cyclic dependencies, 2 = layering, 3 = order subgraphs, 4 = node placement, and 5 = edge routing. The other modules are summarized under the label other.

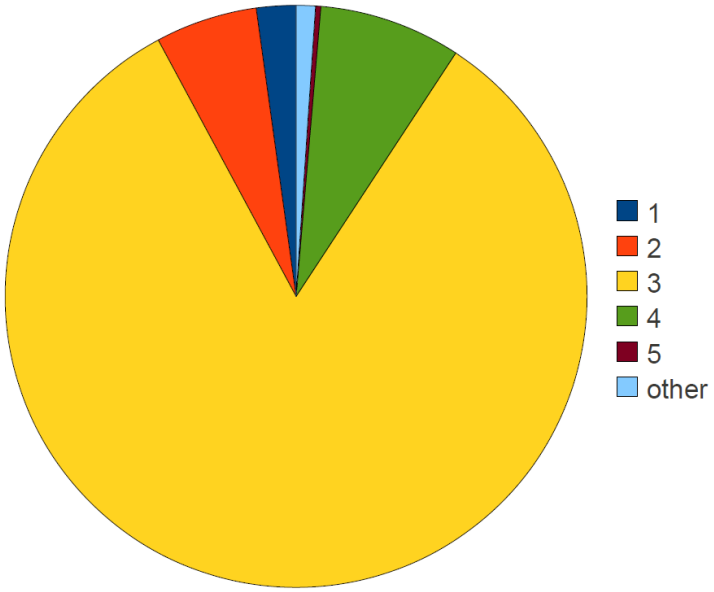
▶ 6.2 Testing and Examples

During development we built up a test suite of 45 test graphs by hand successively. They are small, partly minimal examples of special graph constellations. Each of them was created to test the implementation with regard to a special aspect as soon as the according kind of graph constellation had received special attention. After each development step, the whole test suite was run to ensure that no interference with former achievements had taken place. Amongst others the graphs represent the following problems. Of some, we show drawing examples laid out by KLay Layered.

- Nested nodes - different depth levels.
- Compound node with more than one child.
- Edge connected to port on only one side.
- Compound node with incoming port edges as well as incoming non port edges. See for example Figure 5.1(a).
- A compound node with outgoing edges of three types: hierarchy crossing, port edge, and normal edge.
- Descendant edges, incoming and outgoing, with ports or without.



(a) Distribution of computation time for a small graph (10 nodes, 7 edges, 3 hierarchy levels).



(b) Distribution of computation time for a large graph (150 nodes, 198 edges, 7 hierarchy levels).

Figure 6.5. Component time partition for graphs of different size. Legend: 1 = removal of cyclic dependencies, 2 = layering, 3 = order subgraphs, 4 = node placement, and 5 = edge routing. The other modules are summarized under the label other.

module	small graph (ms)	large graph(ms)
removal of cyclic dependencies	10.088	320.268
edge and layer constraint edge reversal	0.132	0.287
greedy cycle removal	1.150	1.629
layering	8.791	1.721
layer constraint application	0.184	0.180
removing compound dummy edges	0.796	2.673
long edge splitting	0.657	15.011
port side processing	0.190	2.580
port order processing	2.341	6.591
layer sweep crossing minimization	62.073	831.750
order subgraphs	0.806	12154.000
layer constraint edge reversal	0.127	1.225
port position processing	0.204	2.064
node margin calculation	0.714	7.161
set compound side dummies	1.306	89.832
linear segments node placement	14.238	1155.000
orthogonal edge routing	102.905	43.817
long edge joining	0.300	10.992
restoring of reversed edges	0.194	1.479
preparation of layout application	0.913	12.362

Table 6.1. Module computation times for a small and a large graph.

- Outgoing descendant edge to target with successor.
- Descendant edges crossing more than one depth level - both directions.
- Graphs containing hierarchy crossing edges. See for example Figure 2.1(c).
- Edge crossing more than one hierarchy level. See for example Figure 6.6.
- Hierarchy crossing edges with ports. See for example Figure 6.7.
- Compound nodes with northport or southport edges.
- Ports connected to more than one edge, even with descendant edges. See for example Figure 6.8.
- Simple cyclic dependency - two constituting edges on the same depth level. See Figure 5.2(a).
- A cyclic dependency with different depth levels at the source and target of the edges. See Figure 6.9
- Compound graph with a cyclic dependency formed by an edge without ports and a port edge. See Figure 6.10.

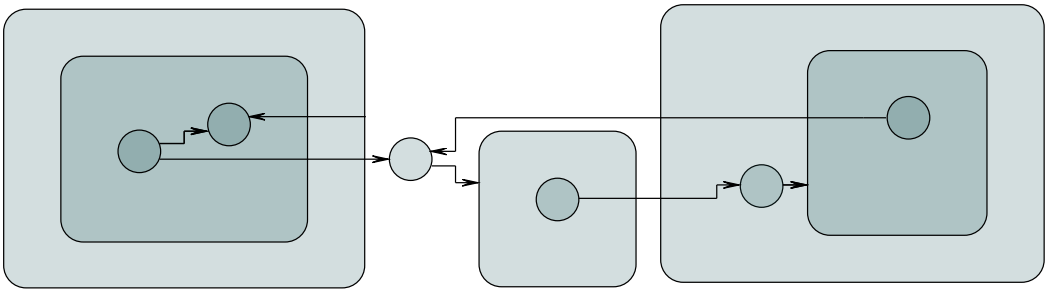


Figure 6.6. Indirect compound cycle via a leaf edge.

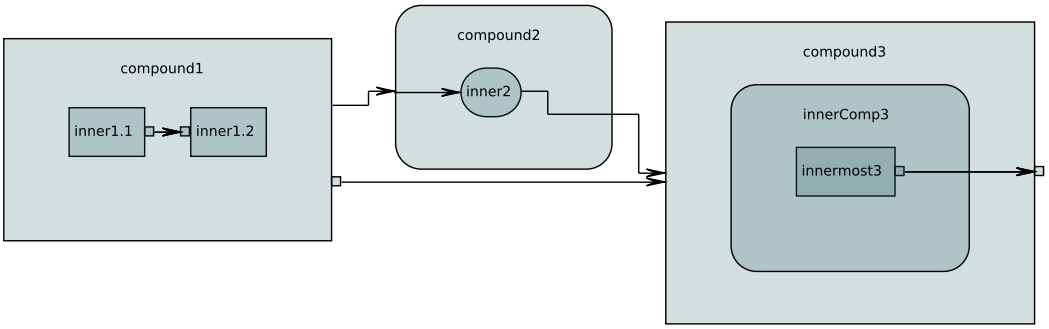


Figure 6.7. Graph with different types of edges and a hierarchy crossing descendant port edge.

- Indirect cycle of compound nodes via a leaf node. See for example Figure 6.6.
- A graph with hierarchy crossing edges and a cyclic dependency built by edges situated in different depth levels. See Figure 6.11.
- Graphs with potential for the stair effect. See for example Figure 5.6(b).
- Graph containing a cyclic dependency built by four edges, leading via a leaf node, and with edges of different depth level as well as edges with different depth levels of source and target. Additionally, the graph contains an outgoing descendant edge crossing more than one depth level. See for example Figure 6.6
- Graph with two cyclic dependencies, one with multiple edges, both indirect (constituted by more than one edge) and built by edges on different depth levels. See Figure 6.12.
- Graphs without nested structure.

Additionally, the algorithm was tested with 35 random graphs, almost all containing cyclic dependencies. A test graph example is given in Figure 6.13. The highest amount of nodes tested was 246, the largest number of edges 198. The highest nesting level tested with random graphs was seven. The largest graph tested that

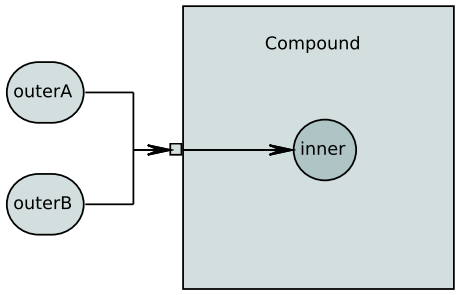


Figure 6.8. Port with multiple edges, including a descendant edge.

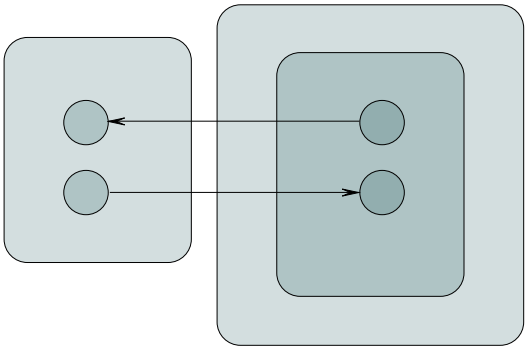


Figure 6.9. A cyclic dependency with different depth levels at the source and target of the edges.

contained hierarchy crossing edges consisted of 150 nodes, 198 edges and had seven hierarchy levels. We made the observation that node overlaps occurred in none of the graphs tested and that edges were connected correctly to the nodes. The drawings were duly compact, however the avoiding of the stair effect, see Section 5.4, causes distention in the layout direction. We can observe that the aspect ratio of graphs from papers is sometimes different from that of our results in this respect,

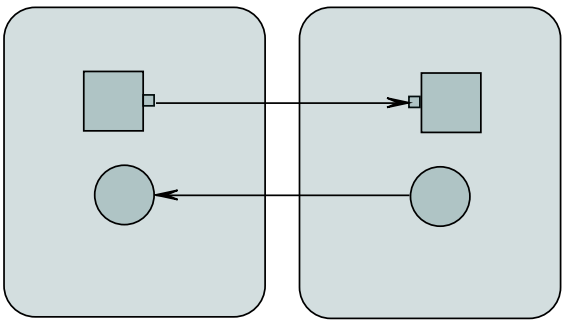


Figure 6.10. A cyclic dependency with port and no port connections of the edges.

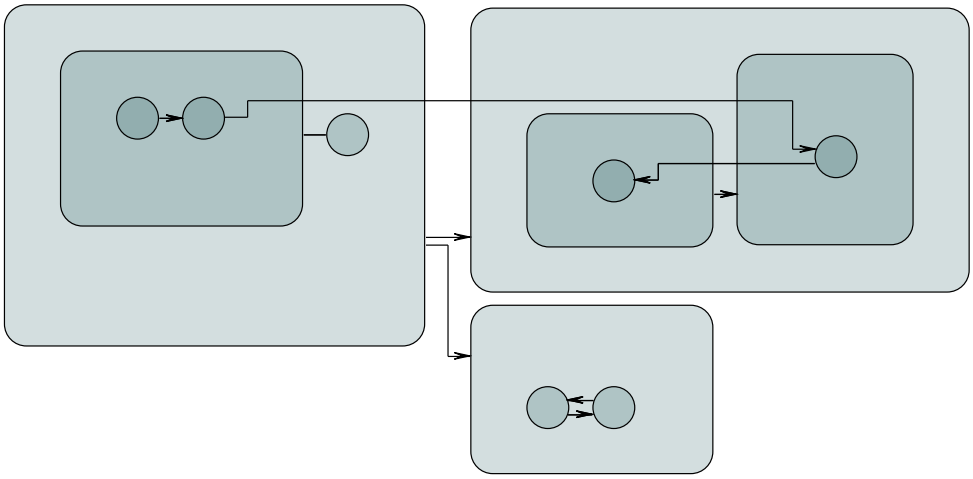


Figure 6.11. A graph with hierarchy crossing edges and a cyclic dependency built by edges situated in different depth levels.

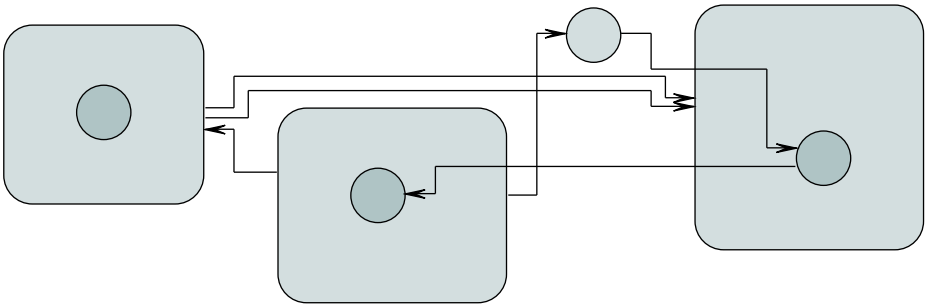


Figure 6.12. A graph with indirect cyclic dependencies and multiple edges.

an example is shown in Figure 6.14. Here we have to choose whether to prefer edge bend minimization or aspect ratio, as these criteria are in conflict. It would also be possible to insert according dummy edges in dependence of a layout option.

The number of crossings between adjacency edges as well as between nodes and adjacency edges is not yet satisfactory. This is due to the cycle breaking heuristic used in the SOP, which is not specialized in the problem. Subgraphs that are intertwined after the node ordering phase are untangled correctly, but the edge and node/edge crossings introduced could be reduced by implementation of a dedicated cycle breaking routine.

To improve crossing minimization, together with optimization of the SOP with regard to computation time, is one of the most pressing tasks for the future.

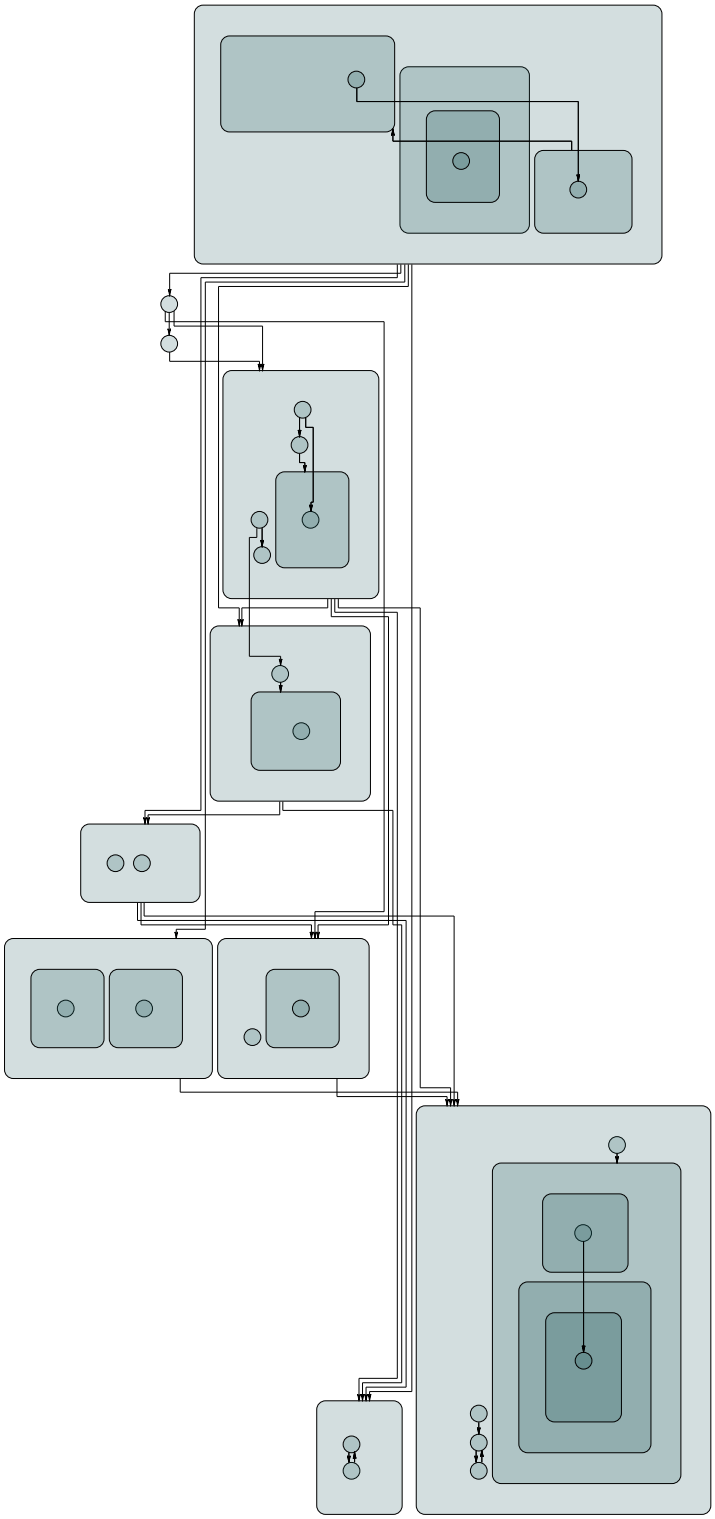
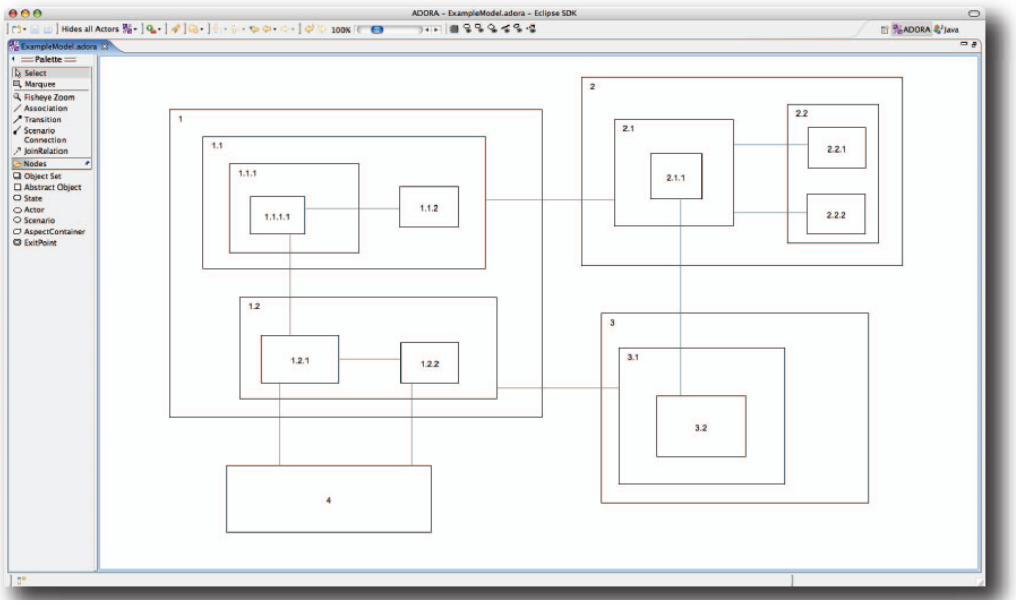
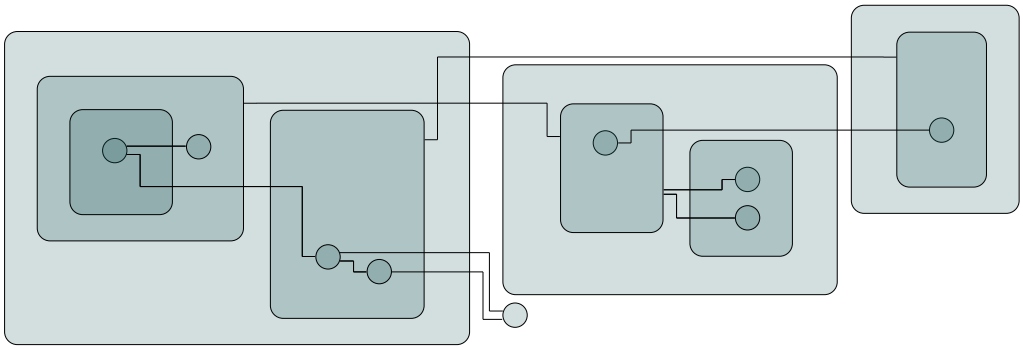


Figure 6.13. A test graph from the suite build with the help of a random graph generator.



(a) Original drawing from [35].



(b) According graph structure drawn by KLay Layered.

Figure 6.14. An example from literature drawn by KLay Layered.

7

Conclusion

This closing chapter gives a short survey of the preceding chapters (Section 7.1) and draws some attention to work to be done in the future (Section 7.2).

▶ 7.1 Summary

This thesis provided a rather extensive overview of the literature on automatic layout for clustered and compound graphs. Furthermore the KLay Layered algorithm of KIELER was furnished with additional modules to enable it to draw clustered graphs.

We connected our implementation to KIML and its KGraph structure by means of the CompoundKGraphImporter, which imports the graph to a flat representation in the LGraph format. This is rendered possible by the introduction of according dummy nodes, especially those marking the first and last layer a compound node spans. This flat graph representation and its global layering leads to a sensible penalty with regard to computation time. However it brings the advantage of compact drawing and the possibility to draw compound graphs with KLay Layered without severe changes to the structure and the main phases of the algorithm.

The problem of cyclic dependencies of compound nodes was solved by the introduction of the CompoundCycleProcessor, which preprocesses the cycle breaking phase. It constructs simplified representations of the dependencies by the propagation of edges up to the outermost compound nodes affected by them and decides on edge reversion with the help of these graph structures.

The actual layering of the flattened graph structure is managed with the help of dummy edge insertion and asks special treatment to avoid the stair effect, an occurrence of unnecessary edge bends.

The node crossing phase in contrast gave cause to the most complex changes and additions. As the nodes of different compound nodes have not only to be kept separate in each layer, but also in the same relative order over all layers, both an additional organization class for the node ordering, the CGCM, and the intermediate processor SOP were introduced. The CGCM presents the nodes and gathered node

groups to the crossing minimization heuristic in a way that the child order of each compound node is determined separately. Thus, nodes of a subgraph are kept in unbroken sequences. The SOP unknots mingled subgraphs over all layers.

The drawing of bounding rectangles is rendered possible with the help of dummy node and edge segments at the upper and lower sides. They are presented to the node placement algorithm as linear segments to place in a straight line.

The actual layout application is prepared by the CompoundGraphRestorer, an intermediate processor situated after the edge routing, and realized by the CompoundKGraphImporter, which closes the cycle of compound graph drawing in KLay Layered.

▶ 7.2 Future Work

There is some future work to be proposed. KLay Layered would profit from attention to the following aspects.

- Currently, north and south ports of compound nodes are moved to the left or right side of the node, according to their direction. They should be placed at the upper or lower side of the bounding rectangle instead, the edges routed in the way KLay Layered handles the problem for leaf nodes.
- Port constraints for compound nodes need support. This concerns also the restoring of reversed edges, which have to be reconnected to their original port dummy nodes.
- The drawing of regions, for example for the expression of concurrent execution in SyncCharts, is not yet accounted for.
- It should be explored if dedicated cycle breaking heuristics in the CompoundCycleProcessor and in the SOP can be implemented resp. developed to improve the quality of the drawings, especially with regard to adjacency edge crossings.
- Selfloops of compound nodes have not been regarded especially so far.
- Interactive layout, different layout directions and the separation of connected components should be supported.
- Label placement for compound graphs has not been object of concern yet.
- The SOP has to be optimized with regard to computation time.
- Dummy edges that connect dummy side nodes to the LCB and RCB during the node ordering phase might reduce the size of compound dummy nodes.
- It could be explore, how the side dummy node placement in relation to long edge dummies can be improved to reduce node size problems.

Especially the task of finding a favorable cycle breaker for the SubGraphOrdering-Processor should be an appreciable enhancement with respect to crossing minimization and therefore readability, while the optimization of the SOP can be expected to lower the overall computation time.

Bibliography

[1] M. Balzer and O. Deussen. Level-of-detail visualization of clustered graph layouts. In *Visualization, 2007. APVIS '07. 2007 6th International Asia-Pacific Symposium on*, pages 133–140, feb. 2007.

[2] François Bertault and Mirka Miller. An algorithm for drawing compound graphs. In Jan Kratochvíyl, editor, *Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 197–204. Springer Berlin / Heidelberg, 1999.

[3] R. Bourqui, D. Auber, and P. Mary. How to draw clusteredweighted graphs using a multilevel force-directed graph drawing algorithm. In *Information Visualization, 2007. IV '07. 11th International Conference*, pages 757–764, july 2007.

[4] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70:066111, Dec 2004.

[5] José Cruz-Lemus, Marcela Genero, M. Manso, and Mario Piattini. Evaluating the effect of composite states on the understandability of uml statechart diagrams. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 113–125. Springer Berlin / Heidelberg, 2005.

[6] José Cruz-Lemus, Marcela Genero, Mario Piattini, and Ambrosio Toval. An empirical study of the nesting level of composite states within uml statechart diagrams. In Jacky Akoka, Stephen Liddle, Il-Yeol Song, Michaela Bertolotto, Isabelle Comyn-Wattiau, Willem-Jan van den Heuvel, Manuel Kolp, Juan Trujillo, Christian Kop, and Heinrich Mayr, editors, *Perspectives in Conceptual Modeling*, volume 3770 of *Lecture Notes in Computer Science*, pages 12–22. Springer Berlin / Heidelberg, 2005.

[7] Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. A compound graph layout algorithm for biological pathways. In János Pach, editor, *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 442–447. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-31843-9_45.

[8] Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. A layout algorithm for undirected compound graphs. *Information Sciences*, 179(7):980–994, 2009.

[9] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[10] Peter Eades, Robert F. Cohen, and Mao Lin Huang. Online animated graph drawing for web navigation. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 330–335, London, UK, 1997. Springer-Verlag.

[11] Peter Eades and Qing-Wen Feng. Multilevel visualization of clustered graphs. In Stephen North, editor, *Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, pages 101–112. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-62495-3_41.

[12] Peter Eades, Qing-Wen Feng, and Xuemin Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In Stephen North, editor, *Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, pages 113–128. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-62495-3_42.

[13] Peter Eades and Qingwen Feng. Orthogonal grid drawing of clustered graphs. Technical report, 1996.

[14] Peter Eades and Mao Lin Huang. Navigating clustered graphs using force-directed methods. *J. Graph Algorithms Appl.*, 4(3):157–181, 2000.

[15] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. In *Proceedings of the First International Conference on Computational Graphics and Visualization Techniques*, pages 34–43, 1991.

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

[17] Peter Eades, Xuemin Lin, and Roberto Tamassia. An algorithm for drawing a hierarchical graph. *International Journal of Computational Geometry and Applications*, 6(2):145–155, 1996.

[18] Qing-Wen Feng, Robert Cohen, and Peter Eades. Planarity for clustered graphs. In Paul Spirakis, editor, *Algorithms — ESA '95*, volume 979 of *Lecture Notes in Computer Science*, pages 213–226. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60313-1_145.

[19] Qingwen Feng. *Algorithms for Drawing Clustered Graphs*. PhD thesis, The Department of Computer Science and Software Engineering, University of Newcastle, 1997.

[20] Michael Forster. Applying crossing reduction strategies to layered compound graphs. In Michael Goodrich and Stephen Kobourov, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 115–132. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36151-0_26.

[21] Michael Forster. A fast and simple heuristic for constrained two-level crossing reduction. In *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 206–216. Springer, 2005.

[22] Yaniv Frishman and Ayellet Tal. Dynamic drawing of clustered graphs. In *INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization*, pages 191–198, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software—Practice & Experience*, 21(11):1129–1164, 1991.

[24] Hauke Fuhrmann. *On the Pragmatics of Graphical Modeling*. Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.

[25] Pawel Gajer and Stephen Kobourov. Grip: Graph drawing with intelligent placement. In Joe Marks, editor, *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 104–109. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44541-2_21.

[26] Mao Lin Huang and Peter Eades. A fully animated interactive system for clustering and navigating huge graphs. In *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*, pages 374–383, London, UK, 1998. Springer-Verlag.

[27] Mao Lin Huang, Peter Eades, and Junhu Wang. On-line animated visualization of huge graphs using a modified spring algorithm. *Journal of Visual Languages & Computing*, 9(6):623–645, 1998.

[28] T. Itoh, C. Muelder, Kwan-Liu Ma, and Jun Sese. A hybrid space-filling and force-directed layout method for visualizing multiple-category graphs. In *Visualization Symposium, 2009. PacificVis '09. IEEE Pacific*, pages 121–128, april 2009.

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

[30] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

[31] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in LNCS. Springer-Verlag, Berlin, Germany, 2001.

[32] Atsuyuki Okabe, editor. *Spatial tessellations : concepts and applications of voronoi diagrams*, 2000.

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

[34] Marcus Raitner. Visual navigation of compound graphs. In János Pach, editor, *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 403–413. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-31843-9_41.

[35] Tobias Reinhard, Silvio Meier, and Martin Glinz. An improved fisheye zoom algorithm for visualizing and editing hierarchical models. In *Second International Workshop on Requirements Engineering Visualization*, pages 9–19. IEEE, 2007.

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

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

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

[39] Georg Sander. Graph layout for applications in compiler construction. *Theoretical Computer Science*, 217(2):175–214, 1999.

[40] Falk Schreiber. *Visualisierung biochemischer Reaktionsnetze*. PhD thesis, Universität Passau, Innstrasse 29, 94032 Passau, 2001.

[41] Christoph Daniel Schulze. Optimizing automatic layout for data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2011.

[42] Miro Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf.

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

[44] Kozo Sugiyama and Kazuo Misue. A generic compound graph visualizer/manipulator: D-abductor. In Franz Brandenburg, editor, *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 500–503. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0021834.

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

[46] William Thomas Tutte. How to draw a graph. In *Proceedings of the London Mathematical Society*, volume 3, pages 743–768, 1963.

[47] Xiaobo Wang and Isao Miyamoto. Generating customized layouts. In Franz Brandenburg, editor, *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 504–515. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0021835.

[48] Martin Wattenberg. A note on space-filling visualizations and space-filling curves. In *Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, pages 24–, Washington, DC, USA, 2005. IEEE Computer Society.

List of Figures