Node and Label Placement in a Layered Layout Algorithm

John Julian Carstens

Master Thesis
September 2012

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

Advised by Dipl.-Inf. Christoph Daniel Schulze

Abstract

In graphical modelling, the arrangement of diagram elements can be a tiresome and mechanic work. To free users from this, layout algorithms arrange the diagram elements automatically.

Different modelling domains require different layouts. A layout approach suitable for data flow diagrams is the layered layout approach. This approach is structured in several phases, of which one is called node placement. Present implementations are known to produce many edge bends. One task of this thesis is to employ a node placement algorithm which yields significantly less edge bends.

Another well known problem in layout algorithms is label placement. Present approaches rely on a post-processing by placing labels after the diagram has been laid out. The approaches presented in this thesis integrate label placement into the layout, resulting in more freedom and clean placements of labels.

Evaluation shows that both tasks can be fulfilled, but have to accept a trade-off which is a generally larger drawing of the respective diagrams.

Eidesstattliche Erklärung

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

Kiel,

Contents

List of Figures

List of Tables

Abbreviations

DAELP Discrete Admissible Edge Label Placement

GMF Graphical Modelling Framework

KAOM KIELER Actor-Oriented Modelling

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KIML KIELER Infrastructure for Meta Layout

KLay KIELER Layout Algorithms

OGDF Open Graph Drawing Framework

SAELP Side Aware Edge Label Placement

UML Unified Modelling Language

Chapter 1

Introduction

A difficult problem with computer systems is to make them accessible and usable to a large group of users. Especially the different knowledge background and the experience of users in working with computer programs often poses a great hurdle to developing usable designs. Here, a compromise is found by abstracting the task in a way which allows many users to work in an environment that is already known to them. To achieve that, graphical modelling and design may be a feasible solution. A graphical web site editor is considered more usable than an editor which relies on textual source code.

The same holds for engineering and the design of large and complex systems. Here, popular abstraction is to combine the used hardware or software components graphically, rather than using a textual programming or hardware description language.

But building on a common abstraction is far from being the only benefit of graphical modelling. Beside its usage in design and development, graphical models can also be used for documenting and describing existing systems. An example for that are some model types of the Unified Modelling Language (UML). Class diagrams, for example, can describe large object oriented software in a way that allows a quicker overview compared to studying the source code line by line.

When comparing textual and graphical design, a core problem of graphical design appears: while layout and arrangement in a textual environment is typically straightforward due to the few levels of freedom, layout and arrangement in a graphical environment can be a complex and time-consuming problem. When designing a diagram such as the one shown in Figure 1.1, a lot of time is used to arrange the diagram elements, a task that usually results in no semantic benefit and may drive users away from graphical design due to its monotony.

Automatic layout algorithms were developed to alleviate users from the need of having to layout their diagrams manually. With that, diagrams can be arranged automatically, to free the user of this time-consuming and annoying task. As one can easily imagine, the automatic layout of diagrams is a demanding problem which

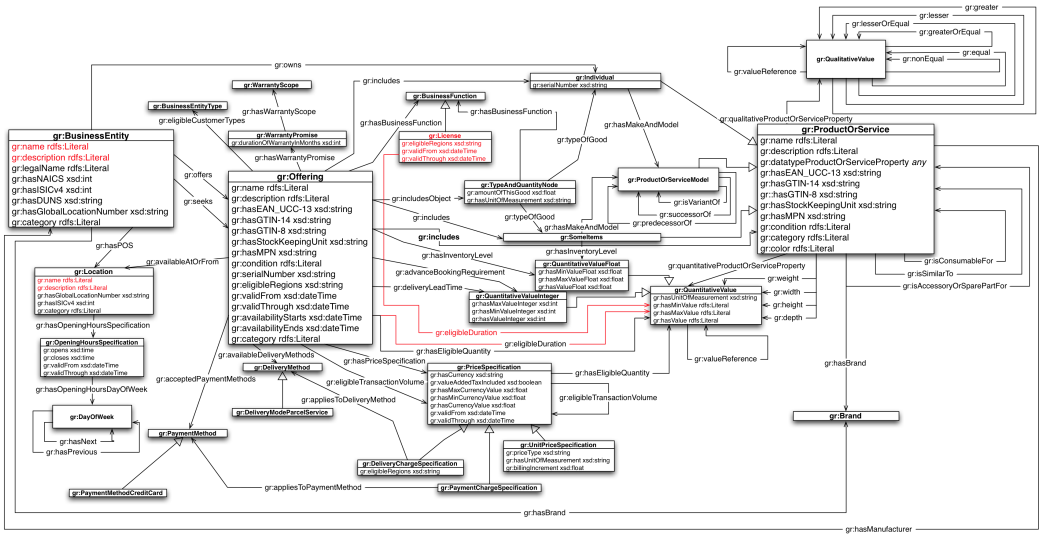


Figure 1.1. An UML diagram showing the structure and relations of the GoodRelations ontology.[1]

offers an almost infinite pool of possible approaches. Each of these approaches usually focusses on a certain diagram type or class of diagram types, because the requirements to an automatic layout differ from domain to domain.

The diagram type relevant to this thesis are *data flow diagrams*. With data flow diagrams, the system is represented by components, usually called actors, that process data and links that define how data is distributed between the actors. This distribution is also called the data flow. Data flow diagrams themselves were subject to a lot of research work in computer science over the last decades, and may have a different behaviour or semantics, depending on the definitions [Kah74, LM87]. An example for a data flow diagram is given in Figure 1.2, where the Fourier transformation of a waveform is computed and plotted.

As the name of this diagram type suggest, an important part of the drawings is the flow of the data. Hence, one goal of an automatic layout approach would be to arrange the drawing in a way that emphasizes the flow of data, resulting in a diagram that is easy to follow. An approach which deals with exactly this problem is the *hierarchical* or *layered* layout approach.

To be able to cope with a complex task such as automatic layout, algorithms are

[1] `http://purl.org/goodrelations/`

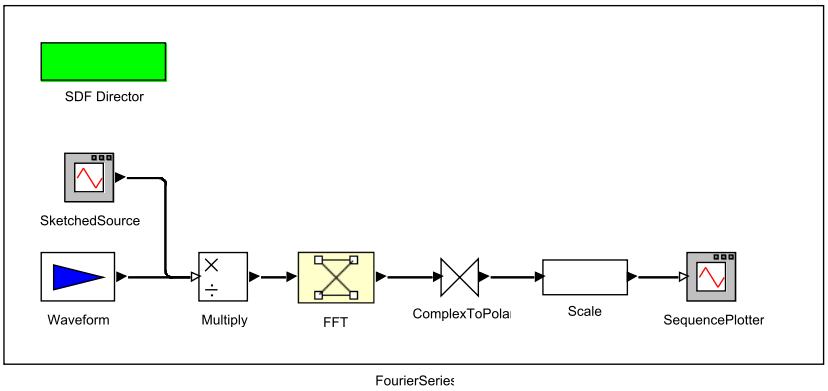


Figure 1.2. A data flow diagram from the Ptolemy II modelling tool [EJL+03], showing the computation of a Fourier series.

usually split into several subtasks or phases. One such phase of the layered layout approach is called *node placement*, which computes the vertical coordinates of actors given a horizontal arrangement computed by a previous phase. This subtask will be dealt with in this thesis.

Another subtask can be the placement of diagram annotations or *labels*. Usually, label placement is applied as post-processing, taking a laid out diagram and fitting the labels into the drawing. In this thesis, approaches are explored that integrate label placement into the actual layout algorithm, resulting in an easier placement due to the ability to explicitly reserve space for the labels.

To give a better idea of the scientific environment and other work on the topics of this thesis, the following section will present related work, investigate the groundwork for the approaches of this thesis, and distinguish other approaches that might look similar at a first glance.

1.1 Related Work

An approach for drawing directed, acyclic graphs, known as the hierarchical or layered approach, was proposed by Sugiyama, Tagawa, and Toda [STT81]. They split the problem of graph drawing into several sub problems to make it more manageable. The general idea of sorting vertices into layers that are placed below or above each other is realized in three steps. In the first step, the set of vertices is partitioned into layers. After that, the order of the vertices inside every layer

is changed such that the connections between the vertices of any two consecutive layers produce as few crossings as possible. In a final phase, the horizontal position of every vertex is determined, as the vertical position is indicated by the layering. As this thesis deals with vertex positioning (also referred to as node placement), the work of Sugiyama et al. is groundwork for it and implemented in an algorithm based on the Sugiyama approach that will be presented later in this section.

Several approaches to the node placement sub task were investigated. An approach by Sander uses the concept of *linear segments*, grouping vertices that are placed on a straight line [San96]. Linear segments are used mainly to make sure that edges spanning multiple layers are drawn straightly, while connections between actors are neglected.

A similar but more complex approach which creates larger vertex groups is proposed by Buchheim et al. [BJL01]. In the beginning, a virtual placement is created by placing vertices leftmost or rightmost. The average of this placement is used for thoroughly traversing the graph layer-wise until optimal sequences of vertices are found. With this, every edge has at most two bends if a straightforward edge drawing is used. Compared to the following approach, this approach consists of complex algorithms with larger run times.

A third approach is proposed by Ulrik Brandes and Boris Köpf [BK02]. Again, vertices are grouped into larger units called *blocks* whose vertices are placed on a straight line. This approach can be seen as being in the middle of the two other approaches. The blocks can also consist of vertices and edges, but are created in a best effort manner by traversing the graph in four directions, contrary to the complex sequencing of Buchheim. The approach of this thesis bases on the approach by Brandes and Köpf by extending it, especially by adding support for vertex sizes and connection points.

For the evaluation of results, a method for quantifying the quality of a layout is necessary. Basic aesthetics criteria were presented by Sugiyama et al. alongside with their layout approach.

A work that focusses on aesthetics of drawn graphs is presented by Helen Purchase [Pur02]. She presents aesthetics criteria that can be applied to decide on the quality of a graph drawing.

In terms of label placement, the root of the placement problems is found outside of the domain of computer science. The first approaches on this field were connected to cartography, dealing with the placement of city names or other landmarks on drawn maps.

First approaches to an automatic placement of labels in cartography were

investigated by Pinhas Yoeli and Eduard Imhof [Yoe72, Imh75]. They laid the foundation for most placement algorithms by introducing the approach of defining a set of candidate positions for each label, preferably close to the labeled element, and a set of quality rules to guide the selection of the best of the offered candidate positions.

While computer science has developed algorithms for a good cartographic labeling, the introduction of graphical modelling and layout algorithms created a new set of problems in the domain of graph labeling. Konstantinos Kakoulis and Ioannis Tollis did a lot of research in this area and developed algorithms for different labeling problems in graph drawing, most notably node label placement and edge label placement as a post-processing on an already drawn graph [KT03]. They apply the candidate position approach to the different placement problems. Label placement in this thesis is addressed in a similar way. However, instead of performing a post-processing on an already laid out graph, the placement is incorporated into the layout, thus adding the ability to explicitly reserve space for the labels.

An interesting approach to label placement is presented by Pak Wong et al. [WMP+05]. Instead of placing edge labels close to an edge, they use the label itself to represent the edge. Although this is certainly a creative idea, the practical usage is questionable since readability, especially with respect to edge crossings, is then an issue.

In the *dot* layout algorithm by Gansner et al., label placement is also incorporated into the layout algorithm and space for edge labels is reserved by modifying the edge routing [GKN02]. The label is then placed next to the center of an edge and can also be connected to the corresponding edge by a dashed line. This is a rather simple and inflexible method, thus a different approach is chosen in this thesis.

Castelló et al. have published research on label placement in a layered layout approach for state charts [CMT01]. For a feasible placement of edge labels, they introduce sub layers which are placed between the regular layers. The space created by these sub layers is used for the placement of edge labels. While this is a good approach for state charts where edge labels usually follow a uniform structure, some problems occur in data flow diagrams. Not every connection has a label, and the labels may be structured differently, resulting in having to reserve enough space for the largest label. The following approach, that also uses virtual structures introduced by the layout algorithm, such as dummy vertices and layers, might be a better compromise.

Although Eiglsperger et al. worked on label placement in an orthogonal layout

instead of a layered layout, one of their ideas can be used in the context of layered layout as well. In orthogonal layouts, dummy vertices that are not part of the original drawing are used to create ninety degree angles. Eiglsperger et al. use these dummy vertices to save space for labels [EKS03]. A similar approach is used in the context of layered layouts in this thesis.

In the environment of Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), a framework whose goal it is to improve the pragmatics of graphical modelling [FvH10], several researchers investigated and implemented layout algorithms for data flow diagrams. Most notable for the topics dealt with in this thesis is the work of Miro Spönemann and Christoph Daniel Schulze. Spönemann compared hierarchical and orthogonal layout approaches for their use in data flow diagrams, resulting in a modified implementation of Sugiyama's approach with feasible results [Spö09]. Schulze improved this implementation, especially with respect to edge connection points [Sch11]. Additionally, he restructured the algorithm to be more adaptive, allowing to introduce sub tasks between and after the main phases of the algorithm.

1.2 Research Goals

As mentioned in the previous section, the contributions of this thesis will be implemented into the KIELER Layout Algorithms (KLay) Layered layout algorithm. As this thesis consists of two separate problems in the same layout algorithm, the research goals for this thesis are presented in two sections.

1.2.1 Improving Node Placement

The node placement problem is a sub task of the layered layout approach. It decides on vertical positions of diagram elements, after a horizontal ordering was determined by an earlier phase. While it is not that important for the understanding of node placement, keep in mind that the final horizontal position is computed by the last sub task of the algorithm.

The current implementation in the context of KLay Layered has algorithms for each of the sub tasks of the layered layout approach. However, these algorithms are heuristics and may not always produce an optimal result with respect to certain criteria. As Christoph Daniel Schulze also pointed out in his diploma thesis, the results of the current node placement algorithms are not satisfying in some cases,

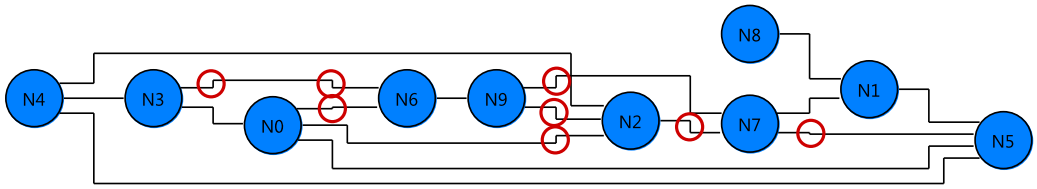


Figure 1.3. An example result of KLay Layered, with edge bends marked of which at least some may be avoided.

especially because unnecessary edge bends are introduced [Sch11]. An example for that can be seen in Figure 1.3.

The first goal of this thesis is the improvement of node placement. A different approach to node placement has to be found, focussing on the straight drawing of edges. As the main domain of applications are data flow diagrams, connections in diagrams will be drawn orthogonally, meaning that all angles in the drawings are ninety degrees. This results in many edge bends if connection points of edges are not assigned the same vertical coordinate. Thus, vertical coordinates have to be found in which as many connection points as possible are on the same level.

While a better node placement behaviour with respect to edge bends is the main goal of this thesis, other, connected new features may be introduced as sub goals. A sub goal could for example be the development of a priority mechanism which tries to give certain edges priority over others when it comes to deciding which edges to draw straightly.

1.2.2 Implementing Label Placement

The goals in terms of label placement are more open. This is, because there is no real label placement included in the current implementation of KLay Layered, resulting in a wide field of possible goals and approaches.

Label placement can refer to all kinds of annotations or text fields in a graph. Normally, these annotations are connected to certain elements of a diagram, for example, connectors. So the general task of label placement is to draw these annotations close to their connected elements. Furthermore, the labels should be drawn well away from other elements, to prevent any kind of overlapping or ambiguity as to which elements a label belongs to.

To keep the goals comprehensible, this wide field has to be reduced to a certain

subset. A lot of work has already been done in terms of placing labels as a post-processing step. Thus, the goal of this thesis is an investigation of placement approaches that can be integrated into a layout algorithm such that the placement is easier and better, because the whole layout can be changed to reserve enough space for a good label placement.

1.3 Overview

This thesis starts with an introduction of theoretical and mathematical concepts and a brief description of the working environment, the encapsulating tools, and existing algorithms in Chapter 2. After that, the topic of node placement is investigated in Chapter 3, starting with a detailed problem statement. A presentation of the chosen approach, its implementation, and a comparison of the results with the results of the old node placement algorithm based on linear segments is also included in this chapter. Chapter 4 moves on to the label placement problem. Again, a detailed problem statement is given, introducing and discussing a new sub problem. Approaches to an integrated label placement are presented, followed by a description of how some of these approaches were integrated into KLay Layered. A short evaluation of the implementation's results completes the label placement chapter. Chapter 5 concludes this thesis by summarizing it, discussing the achievements with respect to the posed goals, and by giving an outlook on tasks that remain for future work.

Chapter 2

Preliminaries

In this chapter, the foundations for understanding this thesis will be presented. The chapter starts with a section about terminology and definitions, clarifying any basic models, terms, or structures which will be used later on. Following that, a brief introduction of the working environment in which the contributions of this thesis will be used is given. Finally, the layered layout approach will be introduced and the specialities and features of the relevant implementation, namely KIELER Layout Algorithms (KLay) Layered, will be explained.

2.1 Terminology and Definitions

The basic model of thought when coping with automatic layout is a *graph*, which can be defined as follows:

Definition 2.1. A *graph* G is a pair (V, E), with a finite set of *vertices* or *nodes* V, and a set of *edges* $E \subseteq \{(u, v) | u, v \in V\}$.

A layout algorithm may also extend the set of vertices if necessary by inserting *dummy vertices* into V and removing them before the algorithm has finished. Then, V consists of two partitions, the regular vertices V_r and the dummy vertices V_d.

This rather simple graph definition can be extended or refined to cover different kinds of drawings or model. The first refinement which may impact a layout algorithm is the definition of a *directed* graph:

Definition 2.2. A graph $G = (V, E)$ is called a *directed* graph if all elements of E are ordered pairs which imply the direction of an edge.

An extension to the already defined graphs is a *port-based* graph, which adds connection points for edges to vertices. This connection points may be freely arrangeable on a vertex, may have a fixed side, a fixed order, a fixed ratio, or even a fixed position on the vertex. Its formal definition is given below:

Definition 2.3. A (directed), *port-based* graph G is a 4-tuple (V, E, P, v) with a finite set of vertices V, a set of edges $E \subseteq \{(p, q)|p, q \in P\}$, a finite set of *ports* P, and a function $v : P \to V$ for mapping elements of P to elements of V.

Any layout algorithm which applies to port-based graphs can be used for non-port-based graphs by simply resolving the mapping function v. Thus, the following definitions and algorithms will just be given for port-based graphs.

When working with the layered layout approach, a layering has to be applied to the graph at one point of the layout algorithm. The following definition deals with a *layered*, port-based graph:

Definition 2.4. A (directed), *layered*, port-based graph G is a 5-tuple (V, E, P, L, v), with V, E, P and v defined as in Definition 2.3, and L a finite ordered set $(L_0, ..., L_k)$, which is a partition of V into non-empty *layers*. Each element of V has to be part of exactly one layer.

For the better description of and navigation through graph elements, several relations and auxiliary functions are given in the following definition:

Definition 2.5. Given a (directed), layered, port-based graph $G = (V, E, P, L, v)$, with $t, u, w \in V$, $p, q, r \in P$, $e = (p, q) \in E$ and $i, j \in \mathbb{N}$, the following holds:

- $L(u) = i$, if $u \in L_i$
- $L_i(j) = u$, if u is the j-th vertex of L_i
- $pos(u) = j$, if u is the j-th vertex of L_i
- The number of vertices in a layer L_i is denoted as $|L_i|$
- e is called *short*, if $|L(v(q)) - L(v(p))| = 1$, otherwise it is called *long*
- e is called *inner segment*, if $v(p), v(q) \in V_d$, otherwise it is called *outer segment*
- e is called *in-layer*, if $|L(v(q)) - L(v(p))| = 0$
- The *forerunner* of $L_i(j)$ in a layer is $fore(L_i(j)) = L_i(j-1)$ if $j > 0$
- Likewise, the *follower* of $L_i(j)$ is $foll(L_i(j)) = L_i(j+1)$ if $j < |L_i| - 1$
- $pred(u) = \{v(p) : (p, q) \in E, v(q) = u\}$ denote the *predecessors* of u in a directed graph

▷ Likewise, $succ(u) = \{v(r) : (q,r) \in E, v(q) = u\}$ denote the *successors* of u in a directed graph

The definition of a layered graph can be extended by introducing the concept of a *proper* layering:

Definition 2.6. The layering of a (directed) port-based graph is called *proper*, if the following holds: $L(v(p)) = L(v(q)) + 1$, if $e = (p,q) \in E$.

A proper layering can be employed in any layered graph by inserting dummy vertices into edges which span more than one layer. The dummy vertices are inserted in the spanned layers and connected, such that the constraint of Definition 2.6 is met. With dummy vertices present, non-dummy vertices are also referred to as *regular* vertices.

Furthermore, an ordering can be defined on the vertices inside every layer of a layered graph:

Definition 2.7. Given a (directed), layered, port-based graph $G = (V, E, P, L, v)$ with $u, w \in V$, an *ordering* is a partial order $\prec$ on V, such that $u \prec w$ or $w \prec u$ if and only if $L(u) = L(w)$.

With this, a vertex can be denoted as $u_j^{(i)}$, given a layer $L_i = \{u_0^{(i)}, ..., u_{|L_i|-1}^{(i)}\}$ and an ordering $u_0^{(i)} \prec u_1^{(i)} \prec ... \prec u_{|L_i-1|}^{(i)}$.

The last extension to be made to the graph model is the addition of labels to the elements of a graph. Labels contain text, giving further information about the labeled elements.

Definition 2.8. A (directed), layered, port-based, *labeled* graph G is a 9-tuple $(V, E, P, L, \Theta, v, \theta_V, \theta_E, \theta_P)$, with V, E, P, L and v defined as in Definition 2.4. Θ is a finite set of labels, with $|\Theta| \leqslant |V| + |P| + 3 \cdot |E|$. $\theta_V : V \to \Theta$, $\theta_P : P \to \Theta$ and $\theta_E : E \to (\Theta \times \Theta \times \Theta)$ are mapping functions which assign the labels to elements of the graph. The 3-tuple θ_E maps to consist of a *head*, *center*, and *tail* label respectively.

This concludes the general definitions and terminology section. Now, the focus will lie on the working environment, in which a future implementation will take place.

2.2 Kiel Integrated Environment for Layout Eclipse Rich Client

Since the contributions of this thesis will be integrated and used in the research project Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1], a brief introduction of KIELER and its relevant parts follows.

The KIELER project deals with the improvement of graphical model-based design of large and complex systems. Its capabilities range from editing features such as *structure based editing*, which allows to change the model's structure directly instead of through diagram editing operations, to *focus and context*, a view principle which expands the currently important parts of a model (focus) and hides the rest of the model (context). A key enabler for these techniques is the also provided *automatic layout*, since the structure of the model will often change, requiring in its diagrammatic representation to be adapted and laid out to reflect the changes. Thus, automatic layout is a prerequisite for the practical use of the techniques described above. A much more detailed description of the KIELER project is given by Hauke Fuhrmann as a part of his dissertation [Fuh11].

As the "ER" part of the KIELER acronym shows, KIELER is implemented as an *Eclipse Rich Client*: The different parts of KIELER are implemented as plug-ins for the Eclipse[2] platform and thus are written in Java. Eclipse is a widespread platform originally developed by IBM in 2001. The concepts of KIELER described above fit well with the Eclipse environment, since there a several graphical editors which can be used in Eclipse. Furthermore, the popular Graphical Modelling Framework (GMF) allows the creation of custom graphical editors which can also benefit from the KIELER concepts.

Automatic layout in particular is a technique which graphical editors can immediately benefit from. One of the automatic layout algorithms shipped with KIELER is the KLay Layered algorithm. The contributions of this thesis will be applied to this algorithm, although the presented concepts may also be used within other layout algorithms. These layout algorithms can be used with any graphical editor, by creating a bridge which moulds the content of the editor in a data structure understood by the algorithms. The part of KIELER which provides this bridge for automatic layout is called KIELER Infrastructure for Meta Layout (KIML). A short overview of KIML will be given in the next section.

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

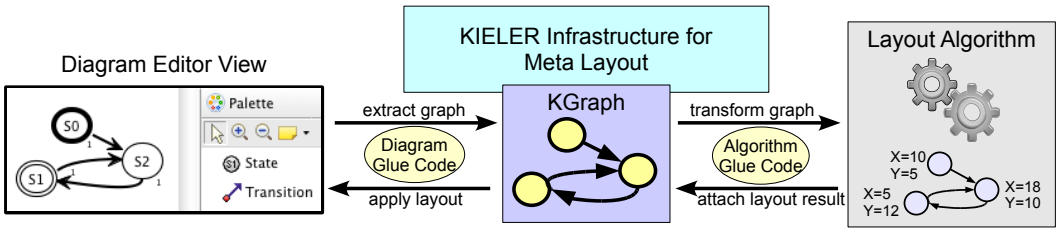


Figure 2.1. KIML structure overview [Fuh11].

2.2.1 KIELER Infrastructure for Meta Layout

As mentioned before, the topic of automatic diagram layout not only consists of layout algorithms themselves, but also of the problem of getting the diagram in a format readable by the layout algorithm and of applying the calculated layout back to the diagram. Also, it is desirable to solve these problems as generically as possible, such that as many algorithms as possible can be used for as many diagram editors as possible without or without much adjustment to any of the components.

The answer to that question in the KIELER project is KIML. It provides *meta layout*, as it is no layout algorithm, but offers a framework for connecting layout algorithms and diagram editors. KIML allows layout algorithms to provide *layout options* for the user to be able to influence the results, e. g., how much space should be left between vertices. An overview of the structure of KIML is shown in Figure 2.1.

A layout algorithm can be chosen individually for any supported editor. This in itself already proves very useful, since different kinds of diagrams require different layout approaches for good readability, and may even have a layout style "typical" for them, e. g., the orthogonal layout often seen in Unified Modelling Language (UML) class diagrams. To offer a set of already existing layout algorithms, bridges to libraries like Open Graph Drawing Framework (OGDF) [CGJ+07] and Graphviz [EGK+02] are included.

Apart from such existing layout libraries, KIELER also includes several own layout algorithm implementations which are grouped in the KLay project. Currently, there exist three layout algorithms in KLay, a *force-based* algorithm, a *planarization-based* algorithm [Cla10, Kut10], and a *layer-based* algorithm [Spö09, Sch11]. The latter will be the algorithm in which the contribution of this thesis will be integrated. Therefore, the next section will give an overview of the layered layout approach.

2.3 The Layered Layout Approach

The layered layout approach was proposed by Sugiyama, Tagawa and Toda [STT81] and is therefore often referred to as the *Sugiyama* layout. The original algorithm was limited to acyclic directed graphs and emphasized hierarchy and flow in the resulting layout. For this reason the approach is also called *hierarchical layout*, but is referred to as the layered approach in the KLay environment. This also helps to avoid confusion with another meaning of the term hierarchy, referring to the inclusion of separate sub-graphs into the graph with compound nodes.

The original approach of Sugiyama et al. proposed to divide the complex task of graph layout in the following four phases:

(i) **Find Hierarchy**: The vertices of a graph are assigned to layers (a hierarchy in the terminology of Sugiyama et al.) by following the direction of the graph. To get a *proper* layering, edges spanning more than one layer have to be broken by dummy vertices in each layer crossed by the edge.

(ii) **Crossing Reduction**: The number of edge crossings is reduced by switching the order of vertices inside a layer.

(iii) **Node Placement**: The exact position of each vertex inside its corresponding layer is determined, preserving the order calculated by the previous phase.

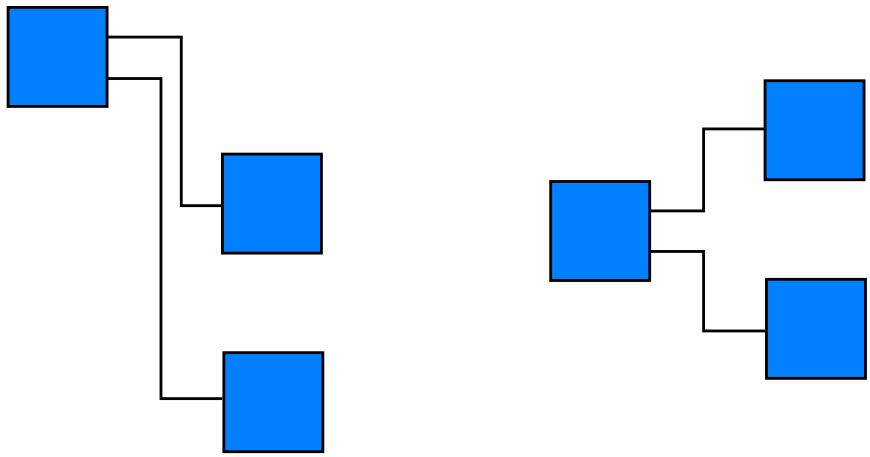
(iv) **Drawing**: Based on the layers and positions calculated in the previous step, the graph is drawn. Dummy vertices have to be removed and replaced by long edges.

For a better evaluation and discussion of layout results, Sugiyama et al. added criteria (called elements in the original paper) of readability to their description of a hierarchical layout [STT81]:

▻ **Criterion** A: The hierarchy of the layout, addressing the traceability of paths.

▻ **Criterion** B: The number of line or edge crossings in a graph, where less crossings are desirable.

▻ **Criterion** C: The straightness of lines with respect to an edge routing parallel to the layout direction, also improving the traceability. This criterion can be split into criterion C_1, the straightness of edges spanning only one layer, and criterion C_2, the straightness of edges spanning more than one layer



(a) A graph with an odd balance.

(b) The same graph with a balanced placement.

Figure 2.2. Balancing in the context of Sugiyama's aesthetics criteria.

- **Criterion** *D*: The closeness of connected vertices. Especially paths should become short.
- **Criterion** *E*: The balance of edges coming from or going into a vertex. This depends on the placement of the connected vertices, e. g., when having two connected vertices on the same side, whether one is placed above the original vertex and one below, which would result in a rather balanced placement, or whether both are placed above or below the original vertex, which would result in an odd placement, as can be seen in Figure 2.2. An equal balancing may improve the readability of branching and joining, but also most likely reduces the straightness of criterion *C*.

Since each phase posed by the approach can be solved by a separate algorithm, many different approaches for each of the phases were developed in addition to the original proposals [DETT99, ES90, GKNV93]. Going into the details of these techniques would exceed the scope of this thesis, at the very least because this thesis does not cover improvements for all the phases of layered layout.

Nevertheless, to give a better idea which will be the concrete environment within which the improvements presented in this thesis will be implemented, the following section will present the structure and workflow of the KLay Layered layout algorithm in some more detail.

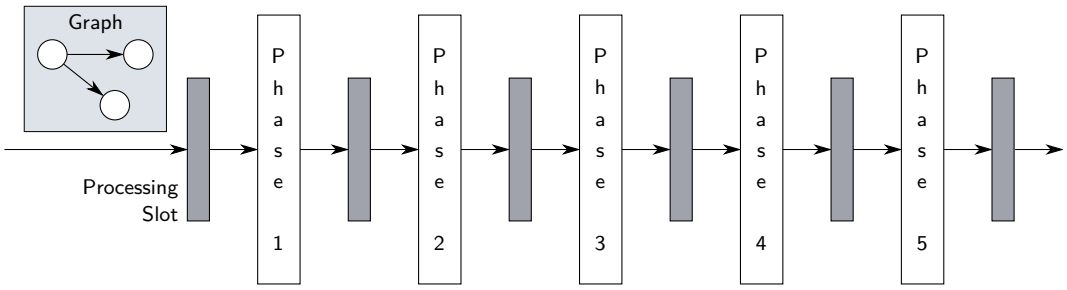


Figure 2.3. An overview of the KLay Layered architecture, showing the five main phases and the slots for intermediate processors [Sch11].

2.4 KLay Layered

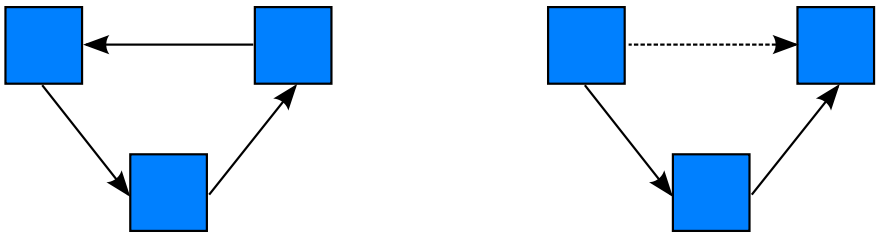
As mentioned before, the KLay Layered algorithm bases on the work of Sugiyama et al. and follows a similar approach. Yet, this basic approach was restructured and improved [Spö09, Sch11], resulting in a larger variety of graphs that can be laid out. KLay Layered has a very modular structure, allowing not only to replace the concrete algorithm used in each phase, but also to insert further algorithms between the phases if necessary. To make sure the modules work together, every phase and module has to satisfy preconditions and postconditions for the following phases and modules to rely on.

The basic structure of KLay Layered can be seen in Figure 2.3. The five main phases will be described in the following section. After that, a description of intermediate processors and a few examples will be given.

2.4.1 The Five Phases of KLay Layered

Contrary to the original four-phase approach of Sugiyama et al. to layered layout, KLay Layered is structured into five phases. The last three phases largely correspond to the original suggestion, with the difference of the last phase being edge routing instead of graph drawing, but since the problem described in the first phase of the original approach consists of two separate problems if the graph contains cycles, it was decided to split this phase into two for a better representation of these problems. Thus, the following five phases are the main elements of KLay Layered:

(i) **Cycle Removal**: Since the layered layout approach requires the graph to be acyclic, cycles have to be removed. This can be achieved by flipping edges,



(a) A graph with a cycle.

(b) The same graph without cycles by flipping the dashed edge.

Figure 2.4. A simple example of cycle breaking.

but not by simply removing them, as seen in Figure 2.4. This is, because the edge has to be present in later phases to be included in the final layout, especially in the edge routing phase. Finding a minimal set of edges to be flipped is an NP-complete problem [GJ79]; thus, a heuristic is used. The current implementation is based on *Greedy Cycle Breaking* which was inspired by Di Battista et al. [DETT99].

(ii) **Layer Assignment**: The goal of this step is to create a layered graph as defined in Section 2.1 from a directed, acyclic graph. To create a proper layering, an intermediate processor inserts dummy vertices as necessary to break long edges into short ones. An example layering is given in Figure 2.5.

An optimization goal for layering is to minimize width and height of the resulting graph, a problem which is, again, NP-complete [ES90]. Heuristics in the current implementation are the *Longest Path Layering* [Spö09] and the *Network Simplex Layering* [Döh10].

(iii) **Crossing Minimization**: In this phase, the algorithm tries to reduce the number of edge crossings by changing the vertex order inside the layers. Figure 2.6a depicts a graph with a crossing that can be avoided by switching the order of the nodes A and B, resulting in the graph shown in Figure 2.6b. But yet again, even if only two layers are considered, the problem of finding an order with minimal edge crossings is NP-complete [GJ83]. A heuristic was found with the *Layer Sweep Crossing Minimizer*. The approach here is to take two layers, to assume the order of the vertices in one layer to be fixed, and to use a heuristic to order the vertices in the non-fixed layer in a way that yields few edge crossings. The current implementation uses a *barycenter* heuristic

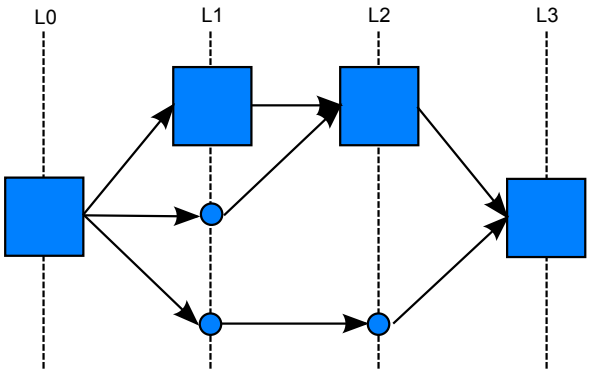
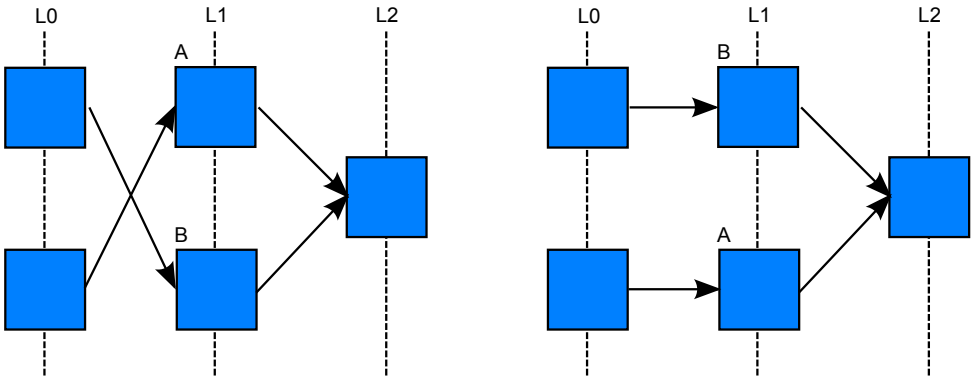


Figure 2.5. A properly layered graph with inserted dummy vertices.



(a) A graph with crossings.

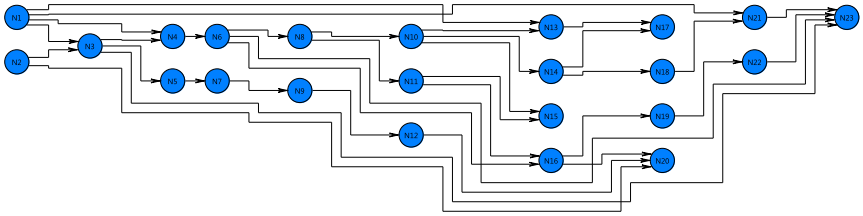
(b) The same graph with minimized crossings by flipping vertex order in layer L_1.

Figure 2.6. An example of crossing minimization.

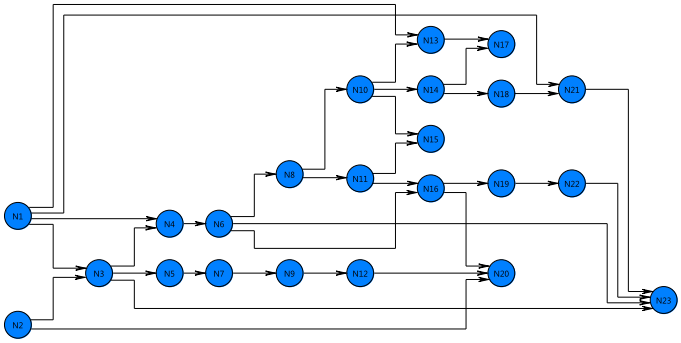
[Spö09] and also supports constraints on the order inside a layer, e. g., when a certain vertex must be placed directly after another vertex for any reason.

(iv) **Node Placement**: In this step, the order of the vertices inside their layers and the relations to other vertices is used to find a final y-coordinate for each vertex. Optimization goals here are a compact drawing of the graph or trying to minimize the number of non-straight edges. An example for the effects of node placement is shown in Figure 2.7.

The current approach, the *Linear Segments Node Placer*, tries to achieve this by



(a) Naive node placement which places vertices on the topmost possible position.



(b) Node placement of the same graph which tries to keep straight paths.

Figure 2.7. The effects of different node placement methods.

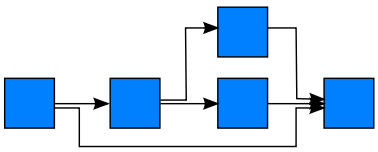
grouping nodes into linear segments which can be drawn straightly. Since the node placement problem is part of the contribution of this thesis, a more detailed problem statement will follow later.

(v) **Edge Routing**: The final step of the layout, if no post-processing is applied, is the edge routing. In this step, vertices are assigned their final positions, which were already roughly determined by layer assignment and node placement. After that, edges can be routed in different fashions. Examples of different kinds of edge routing are given in Figure 2.8.

The simplest way of routing edges is implemented in the *Polyline Edge Router*, which simply draws straight lines from vertex to vertex. A more complex routing is arrived by using splines in the *Spline Edge Router*, which uses the

(a) A graph with polyline edge routing.

(b) The same graph with spline edge routing.



(c) The same graph with orthogonal edge routing.

Figure 2.8. Examples of different edge routing methods.

space left between nodes and other edges to draw smooth and curvy edges [CRR10]. Another approach suitable especially for dataflow diagrams is the *Orthogonal Edge Router*. The current implementation is based on an approach proposed by Sander [San04].

These five phases represent the basic problems which are solved by KLay Layered. To have the ability of flexibly tailoring the algorithm to special needs, e. g., for new graph types or additional elements, and for having the main phases as pure as possible, the processing slots for intermediate processors were introduced. The following section will give a brief explanation of intermediate processors and examples of important processors.

2.4.2 Intermediate Processors

As could be seen in the decision to extend the original four phases of the Sugiyama layout into five phases, a major goal of KLay Layered was to keep the algorithms in the respective modules as pure as possible. Factoring out tasks which are not directly part of the phases into algorithms executed before or after the respective phase is the natural consequence of this idea. These algorithms, executed in the slots between the phases, are called *intermediate processors*.

A prominent example of an intermediate processor is introduced by the task of proper layering. As defined in Section 2.1, a layering is proper if edges only

connect vertices between neighbouring layers. Dummy vertices have to be inserted to break edges spanning more than one layer. If this would be done directly in the layer assignment phase, every layer assignment algorithm would have to include the same code for long edge splitting. Furthermore, a later phase, e. g., every edge router, would have to reconnect the splitted edges. With intermediate processors, one processor for splitting the long edges is placed in the slot after phase two and another processor rejoins the edges after phase five.

In general, every phase can specify which intermediate processors they require in which slot in order to work correctly. The order of intermediate processors inside a slot can be determined by dependencies between the processors, if any exist.

Chapter 3

Node Placement

The first main chapter of this thesis will cope with the node placement problem in the layered layout approach. The chapter will start with a problem statement, introducing the node placement problem in detail and giving the necessary definitions of the problem and possible optimization goals. After that, a possible approach to the problem will be presented and explained, and the implementation of that approach in the context of KIELER and KLay Layered will be described. The last section of the chapter will provide an evaluation of the implemented approach, comparing the new results with other approaches and checking the fulfilment of the posed goals and aesthetics criteria.

3.1 Problem Statement

As seen in Section 2.3, the node placement problem was already part of the approach of Sugiyama et al., where it was originally called *Determination of Horizontal Positions* (remember that Sugiyama et al. assumed graphs to be laid out from top to bottom, not from left to right as in this thesis). It was executed as the third step of a four step layout algorithm. In KLay Layered, as described in subsection 2.4.1, it is the fourth step of a five step algorithm and is called the *Node Placement Problem*. Although the goal is basically the same, the name has been changed to better reflect the generic nature of suitable algorithms.

To get a basic understanding of the problem, a closer look at the role of node placement within the whole layered layout algorithm might help. Figure 3.1 shows an overview of the five phases of KLay Layered, with the node placement phase highlighted. With the information about the first three phases given in subsection 2.4.1, the input to the fourth phase is an acyclic graph which is divided into layers and has a fixed order for the vertices within each layer. Intermediate processors might also have changed the graph, e. g., by splitting long edges to achieve a proper layering. The fifth phase expects a graph in which it can route all edges without having to cross other nodes or introduce more edge crossings than

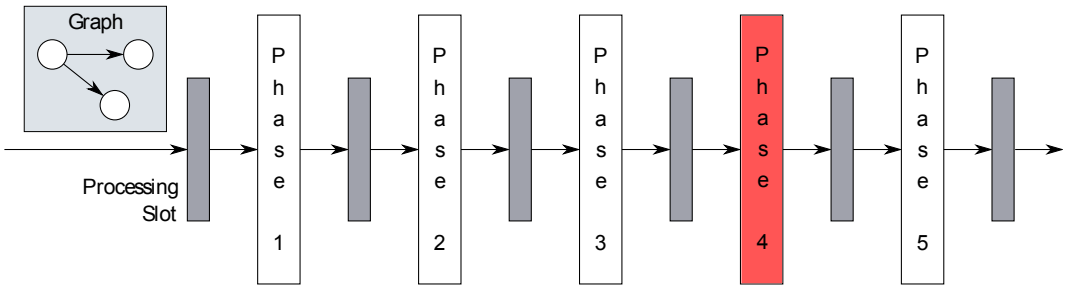


Figure 3.1. The position of the node placement phase in KLay Layered.

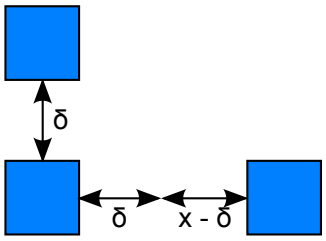


Figure 3.2. Three vertices, with δ denoting the smallest allowed distance between them.

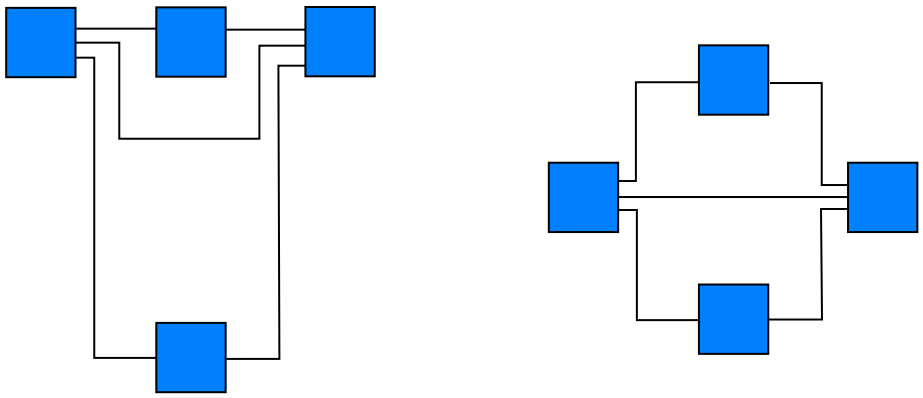
required with the vertex ordering the crossing minimization phase has yielded.

As mentioned before, the choice of layout directions only requires a geometrical transformation. Thus, node placement will take place vertically in the following without loss of generality. Speaking in coordinates, this means that the approximate x coordinate is indicated by the layer a vertex belongs to, while it is the task of the node placer to find suitable y coordinates. Additionally, KLay Layered also requires a layout algorithm to respect a *minimum separation constraint*, also called *object spacing*, a minimum distance between any two vertices, which will be denoted as δ. An example of the effects of a minimum separation constraint is shown in Figure 3.2. Note that δ can be different for different vertex pairs under certain conditions.

With this and the definitions given in Section 2.1, the node placement problem can be defined as follows:

Definition 3.1. In the *Node Placement Problem*, for a given (directed), layered, port-based graph $G = (V, E, P, L, v)$ with an ordering $\prec$, values for $y(u), u \in V, y : V \to \mathbb{R}$ have to be found, such that the following holds:

$$y(u) + \delta \leqslant y(w) \text{ if } w \in V \text{ and } u \prec w$$



(a) A valid, but space wasting solution to the node placement problem.

(b) A solution with a better balancing and space usage.

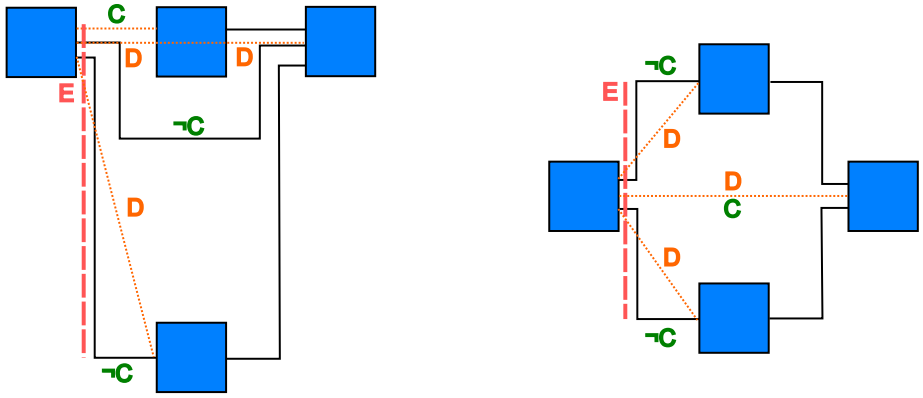
Figure 3.3. The effects of different node placements.

Table 3.1. Aesthetics criteria connected to node placement, as presented by Sugiyama et al.

Criterion	Explanation
C	Reflects whether an edge is drawn straightly, or not.
D	Distance between connected vertices.
E	Balancing of connected vertices, w.r.t. a given vertex.

As Figure 3.3 shows, a node placement which fulfils the requirements of the problem does not necessarily look aesthetic or is easy to read. For a better discussion of these problems, Sugiyama et al. presented aesthetics criteria, as seen in Section 2.3. Furthermore, they connected several criteria to each phase of their algorithm to show which aesthetic criteria are influenced by which phase [STT81]. According to them, node placement influences the straightness of long edges, the closeness of connected vertices, and the balance of the layout. A short reminder of the criteria and their names is given in Table 3.1. A modified version of Figure 3.3, shown in Figure 3.4, depicts how these criteria are influenced by different node placements, which can be considered good or bad with respect to the aesthetics criteria. In the figure, the criteria are only displayed for the leftmost vertex and its connected edges, to keep the diagram clear.

To get a better hold of the problem of creating a node placement that will be considered good with respect to the aesthetics criteria, Sugiyama et al. defined optimization problems [STT81]. For criterion *D*, the closeness of connected objects,



(a) The aesthetics criteria in the space wasting solution.

(b) The aesthetics criteria in the balanced solution.

Figure 3.4. The aesthetics criteria influenced by node placement.

an optimization problem can be formulated like this:

Definition 3.2. To achieve an optimal fulfilment of criterion D, the following term must yield the minimal value over all possible node placements complying with Definition 3.1:

$$\sum_{(p,q)\in E} (y(p) - y(q))^2$$

By solving this optimization problem, the edge lengths of the graph are minimized subject to the boundaries of the node placement problem.

Another optimization goal can be the criterion E:

Definition 3.3. To achieve an optimal fulfilment of criterion E, the following term must yield the minimal value over all possible node placements complying with Definition 3.1:

$$\sum_{w\in V} \left(y(w) - \sum_{u\in succ(w)} \frac{y(u)}{|succ(w)|} \right)^2 + \sum_{w\in V} \left(y(w) - \sum_{u\in pred(w)} \frac{y(u)}{|pred(w)|} \right)^2$$

With this term, a balanced layout is favoured by relating the vertex's placement to the placement of its neighbours.

The final aesthetics criterion influenced by node placement is criterion C, which concerns the straightness of edges. A possible definition can be given by means

(a) Unbalanced node placement with two edge bends.

(b) Balanced node placement with four edge bends.

Figure 3.5. The number of edge bends usually increases, when balancing is applied.

of a weight function $\omega(e), \omega : E \to \mathbb{R}$, which describes the importance of drawing edge e straightly:

Definition 3.4. To achieve an optimal fulfilment of criterion C, the following term must yield the minimal value over all possible node placements complying with Definition 3.1:

$$\sum_{e=(p,q)\in E} \omega(e) \cdot |y(p) - y(q)|$$

While $|y(p) - y(q)|$ determines the straightness of an edge, the weight function allows to foster the straightness of edges with certain characteristics, e. g., by choosing the length of an edge as a weight to have straight longer edges yield a smaller result of the term. When orthogonal edge routing is applied, as described in subsection 2.4.1, non-straight edges might have a larger impact on readability, since every non-straight edge will introduce at least two edge bends.

The simple example shown in Figure 3.5 shows that the goals of criteria C and E can contradict each other, resulting in trade-offs which may have to be made on either side.

The search for a minimal result for the three concerned criteria is a complex task. To achieve a feasible result in an acceptable run time, heuristics are applied. Usually, these heuristics favour certain criteria over others, resulting in rather different node placements. The node placement heuristic presented in this thesis is based on an approach by Brandes and Köpf. Their approach and performed extensions and modifications will be presented in the next section.

3.2 Approach

With the node placement problem in mind, this section will now deal with a heuristic that offers a solution which approximates an optimal solution of the optimization problems given in the previous section. The approach presented here is based on an approach introduced by Ulrik Brandes and Boris Köpf [BK02]. Hence, the first thing to do is to provide a detailed description of their algorithms that goes a bit further into the details than the original paper.

After that, the modifications and enhancements of the approach developed as part of this thesis are explained and discussed, showing how a larger variety of graphs can be laid out, how more power can be given to the user, and how the node placement phase can be made more robust.

3.2.1 The approach of Brandes and Köpf

The approach to node placement as proposed by Brandes and Köpf bases on the idea of aesthetics criteria as presented in Section 3.1, trying to give a heuristic which satisfies the three criteria mentioned above as well as possible. Especially the straightness of long edges is regarded, but also the closeness and the balancing of connected vertices are in the focus of the algorithm.

The terminology and calculations of the original algorithm assume a top-to-bottom layout. Since in the context of KLay Layered a left-to-right layout is assumed, the descriptions and algorithms in the following will be modified accordingly where necessary. Nevertheless, the naming of the parts of the algorithm will follow the original naming by Brandes and Köpf to avoid confusion when comparing or double-checking with the original approach. Mainly, the modifications will consist of switching x and y coordinates.

Brandes and Köpf assume a layered graph without ports. As mentioned in Section 2.1, a port-based graph can be used with any algorithm that deals with graphs without ports by introducing a mapping function $v : P \rightarrow V$. Nevertheless, ports introduce new challenges in terms of creating a good and readable layout, thus, modifications might be necessary to satisfy aesthetics criteria again. The approach by Brandes and Köpf will be explained assuming a portless graph. The necessary modifications to get good results for port-based graphs will be presented in a later section.

Basic idea

The basic idea of the algorithm is to align connected vertices by assigning the same y coordinate. To achieve that, the graph is traversed in four different directions to group connected vertices, each resulting in a separate candidate node placement. The edges between the members of these groups will be drawn straightly. Keep in mind that the basic approach assumes a vertex size of 0. If there are two or more possibilities for an alignment, dummy vertices of long edges will be favoured, to achieve a straight drawing of long edges. If the choice is between vertices of the same kind, the first vertex with respect to the traversal direction is chosen.

The final node placement is determined by taking the node placement which has the smallest height. This placement may now be balanced by creating an average with the other three results.

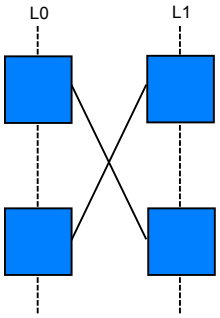
In the following, all four steps of the algorithm are described, starting with an introduction of terms and models used in the context of the algorithm.

Algorithm-specific terminology

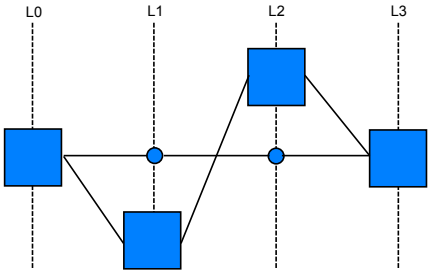
The first thing to notice about the terminology used by Brandes and Köpf is the classification of the different types of edge crossings, called *conflicts*, that might occur between vertices in adjacent layers. The definitions base on the concept of inner and outer segments as presented in Definition 2.5. Remember that inner segments are edges between two dummy vertices, while outer segments are edges connected with at least one regular vertex.

- **Type 0 conflict**: This conflict occurs when two outer segments cross each other, as shown in Figure 3.6a.
- **Type 1 conflict**: This conflict occurs when an inner and an outer segment cross each other, as shown in Figure 3.6b.
- **Type 2 conflict**: This conflict occurs when two inner segments cross each other, as shown in Figure 3.6c.

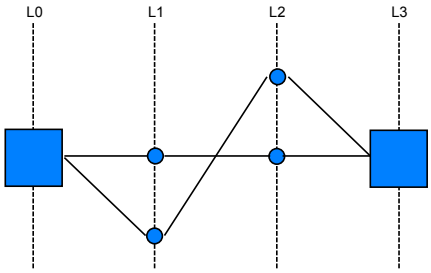
As mentioned before, the graph is traversed in different directions. For every iteration, a combination of a horizontal and a vertical direction is chosen, resulting in a total of four combinations. In the original paper of Brandes and Köpf, the direction names were chosen having a top-to-bottom layout in mind. The directions given here refer to a left-to-right layout and have to be mentally switched when



(a) Two outer segments crossing, resulting in a type 0 conflict.



(b) A type 1 conflict, an inner and an outer segment cross.



(c) A crossing between two inner segments creates a type 2 conflict.

Figure 3.6. The three different conflict types.

comparing algorithms from this thesis with the original approach. The four directions are defined as follows:

- **LEFT** and **RIGHT**: These refer to the order in which the layers of the graph are traversed. If the direction is **RIGHT**, and the layered graph consists of n layers, the layers are traversed from L_0 to L_{n-1}, and vice versa in the case of a **LEFT** direction.

- **DOWN** and **UP**: These refer to the order in which the vertices in each layer are traversed. If the direction is **DOWN**, the vertices are traversed from $L_i(0)$ to $L_i(|L_i| - 1)$, and vice versa in the case of a **UP** direction.

An overview of the possible iteration directions is shown in Figure 3.7.

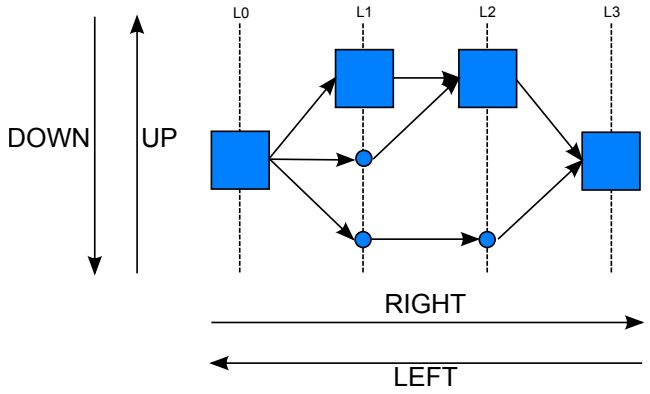
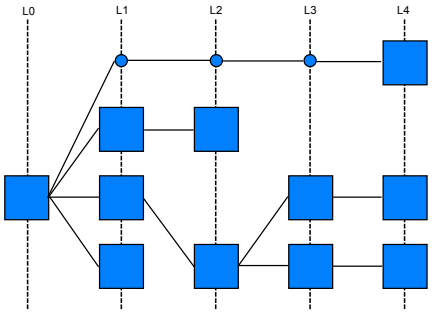


Figure 3.7. The iteration directions; a horizontal and a vertical direction are chosen in each iteration.

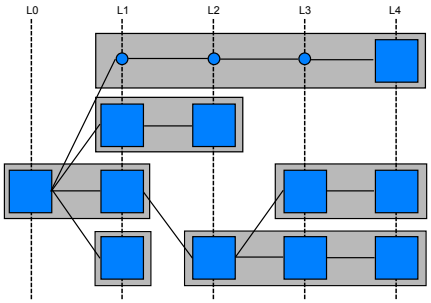
In the four iterations of the two main steps of the algorithm, the graph's vertices are partitioned into subsets of V for each of the direction combinations. The two possible elements which form a partitioning result are called *blocks* and *classes*:

- **Block**: Given n layers, a block consists of one to n vertices from subsequent layers, and represents the alignment which is determined by the algorithm. Only one vertex from each layer may be part of a block, limiting in the maximal block size to n. Depending on the direction of the current iteration, the first vertex in the alignment block is called the *root* of the block. If the direction of the respective iteration was **RIGHT**, the root vertex is part of the leftmost layer of all vertices in the block. In the case of a **LEFT** direction, the root vertex is in the rightmost layer of all vertices in the block. All vertices in a block will be assigned the same y coordinate. Furthermore, a block guarantees that all edges between vertices inside a block will be drawn straightly. Figure 3.8a shows a graph, Figure 3.8b gives the partitioning of the graph into blocks.

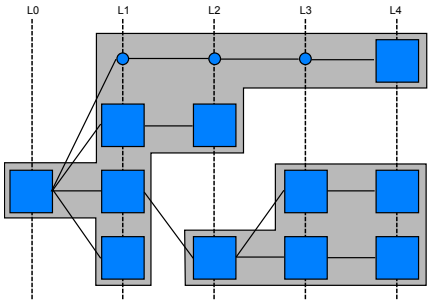
- **Class**: The partitioning into classes is derived from the partitioning into blocks by forming a *block graph* from it. This is done by connecting the vertices of a layer with their antecedent vertex in the layer ordering (if present) in iteration direction (**DOWN** or **UP**) with directed edges and merging the vertices of a block into a single, large, and possibly layer-spanning vertex. Now, the class of a block is determined by merging blocks in vertical iteration direction, until a reachable block is found, who is topmost or bottommost in its layer's vertex order and whose root vertex is in the leftmost or rightmost layer, depending on horizontal



(a) A properly layered graph.



(b) A partitioning of the given graph into blocks.



(c) The corresponding partitioning into classes, if layout direction was **UP**.

Figure 3.8. Different partitionings of a graph.

iteration direction. For example, the block partitioning from Figure 3.8b is transformed into a block graph. Then, classes are formed, resulting in the partitioning given in Figure 3.8c. As one can easily see, the class graph can be compacted by moving the lower right class and its containing vertices up.

With these terms and concepts in mind, a closer look at the original approach of Brandes and Köpf, with the before mentioned adjustments, can now be taken.

A final remark before diving deeper into the algorithms of the approach: The algorithms given in the following are slightly modified in comparison to the algorithms given in the original approach by Brandes and Köpf. The most notable changes are indices starting with zero (instead of one) and the inclusion of code for all possible iteration directions, not only for the combination **RIGHT** and

DOWN. Furthermore, minor mistakes, which were also corrected by the authors,[1] are corrected here, too.

Mark conflicts

The first step of the node placement algorithm by Brandes and Köpf consists of a preprocessing, which investigates edge crossings and marks type 1 crossings for a special treatment following later. For this first step, the possible iteration directions can be ignored, since they play no role in the conflict type decision. The algorithm, given in Listing 3.1, traverses the layers from left to right with the current layer being L_i, starting with the second layer. This is because only inner segments cause type 1 conflicts, and inner segments can not exist between the first and second layer since the long edge would need a regular source vertex before the first layer. The traversal of the graph ends with the second last layer, for the same reason.

Listing 3.1. Step 1: The mark conflicts algorithm.

```
1  for i = 1 ,... ,|L| - 2 do
2    k_0 = 0; l = 0;
3    for l_1 = 0 ,... ,|L_{i+1}| - 1 do
4      if l_1 == |L_{i+1}| - 1 or v_{l_1}^{(i+1)} incident to inner segment between L_{i+1} and L_i then
5        k_1 = |L_i| - 1;
6        if v_{l_1}^{(i+1)} incident to inner segment between L_{i+1} and L_i then
7          k_1 = pos(pred(v_{l_1}^{(i+1)})[0])
8        end if
9        while l ⩽ l_1 do
10         foreach v_k^{(i)} : pred(v_l^{(i+1)}) do
11           if k < k_0 or k > k_1 then mark segment (v_k^{(i)}, v_l^{(i+1)});
12         end foreach
13         l = l + 1;
14       end while
15       k_0 = k_1;
16   end for
17 end for
```

The loop starting in line 3 iterates over the vertices of the next layer after L_i (L_{i+1}). It is then checked, whether the current vertex ($v_{l_1}^{(i+1)}$) is the last vertex in the layer or whether it is part of an inner segment between L_i and L_{i+1}. If that is the case, the auxiliary variable k_1 is set to the position of the last node in L_i.

[1] http://www.informatik.uni-konstanz.de/~brandes/publications/

Table 3.2. Variables used in the mark conflicts algorithm and their usage.

Variable	Usage
i	The current layer
k	A predecessor in the previous layer (see line 10)
k_0	Previous inner segment dummy in the following layer
k_1	Inner segment dummy in the following layer
l	Variable for iteration up to an inner segment dummy or the end
l_1	Inner segment dummy in the following layer or the last vertex in that layer

Additionally, if $v_{l_1}^{(i+1)}$ was part of an inner segment, the variable k_1 is set to the position of the predecessor of $v_{l_1}^{(i+1)}$. This can be done because if an inner segment between L_i and L_{i+1} is present, the vertex in L_{i+1} has exactly one predecessor.

Then, the while-loop starting in line 9 iterates over the vertices in L_{i+1} that come before $v_{l_1}^{(i+1)}$ in the layer ordering. For these vertices, the predecessors ($v_k^{(i)}$) are traversed. A possible crossing is now detected by comparing the auxiliary variables with the position k in layer L_i of $v_k^{(i)}$. In the first iteration, it is only relevant whether k is greater than k_1. This would indicate, that a vertex whose layer order position is above $v_{l_1}^{(i+1)}$ has an edge to a vertex which is below the single predecessor of $v_{l_1}^{(i+1)}$ in layer order of L_i, thus being part of a type 1 or type 2 conflict. The original approach assumes that type 2 conflicts are eliminated in an earlier phase, something that is not necessarily the case in KLay Layered. The handling of type 2 conflicts will be discussed in the implementation section.

The edge causing the conflict by crossing the inner segment of $v_{l_1}^{(i+1)}$ is marked by storing the information in a global list. The final step of the for-loop starting in line 3 is setting the auxiliary variable k_0 to the layer position of $v_{l_1}^{(i+1)}$. With this, future iterations notice in line 11, whether an edge between L_i and L_{i+1} has a source vertex below the predecessor of $v_{l_1}^{(i+1)}$ and a target vertex above $v_{l_1}^{(i+1)}$ in layer order. This would also introduce a type 1 conflict that requires marking.

After one run of this algorithm, which can obviously be done in linear time, all non-inner segments causing a type 1 conflict are marked for future use in a later step of the algorithm. This leads to the next step, the partitioning of the graph into blocks, called *vertical alignment*.

Vertical alignment

The second step of the algorithm is called vertical alignment by Brandes and Köpf and deals with the partitioning of the graph into blocks, as introduced earlier in this section. In this step, the choice of iteration directions becomes important, since several lines have to be modified to fit the selected direction. To symbolize this in the code given in Listing 3.2, the affected lines of code are marked with either (**LEFT**) or (**RIGHT**) and (**DOWN**) or (**UP**). This means that if for example the direction **LEFT** is chosen, then all lines marked with (**LEFT**) are included in the code, while all lines marked with (**RIGHT**) are excluded. The same applies to the markers (**DOWN**) and (**UP**), who are also mutually exclusive. This method of displaying the code will also be used in other algorithms in this section.

Listing 3.2. Step 2: The vertical alignment algorithm.

```
1  initialize root[v] = v, ∀v ∈ V;
2  initialize align[v] = v, ∀v ∈ V;
3  for i = 0,...,|L|−1 do (RIGHT)
4  for i = |L|−1,...,0 do (LEFT)
5    r = −1; (DOWN)
6    r = ∞; (UP)
7    for k = 0,...,|L_i|−1 do (DOWN)
8    for k = |L_i|−1,...,0 do (UP)
9      if v_k^(i) has predecessors u_0 ≺ ... ≺ u_d with d > 0 then (RIGHT)(DOWN)
10     if v_k^(i) has successors u_0 ≺ ... ≺ u_d with d > 0 then (LEFT)(DOWN)
11     if v_k^(i) has predecessors u_d ≺ ... ≺ u_0 with d > 0 then (RIGHT)(UP)
12     if v_k^(i) has successors u_d ≺ ... ≺ u_0 with d > 0 then (LEFT)(UP)
13       for m = ⌊(d+1)/2⌋ − 1, ⌈(d+1)/2⌉ − 1 do
14         if align[v_k^(i)] == v_k^(i) then
15           if (u_m, v_k^(i)) not marked and r < pos(u_m) then (DOWN)
16           if (u_m, v_k^(i)) not marked and r > pos(u_m) then (UP)
17             align[u_m] = v_k^(i);
18             root[v_k^(i)] = root[u_m];
19             align[v_k^(i)] = root[v_k^(i)];
20             r = pos(u_m);
21           end if
22         end if
23       end for
24     end if
25   end for
26 end for
```

In the beginning of the algorithm, the data structures for storing the blocks are initialized. They consist of two maps that include every vertex in the graph as key. In the beginning, each vertex points to itself. After the execution of this algorithm, every vertex in the root map will point to the root vertex of the block it belongs to. This alone is enough to represent the partitioning into blocks, but for more convenience in later algorithms, the align map is also created. In this, every vertex points to the next vertex in the same block, or to the first vertex of the block in case of the last vertex. Thus, a vertex in a block containing only one vertex points to itself.

The algorithm then traverses the graph with respect to the chosen directions, starting with either the first or the last layer. The second loop then starts with either the first vertex in layer ordering or the last one, again depending on the choice of direction. The variable r indicates the position of the most recently aligned vertex in the neighbouring layer the alignment takes place with, to make sure that blocks do not cross each other. Thus, the variable has to be set to -1 if the traversal of the layer starts with the vertex at position 0, or to ∞ if the traversal starts with the vertex with the greatest position number.

The alignment of the blocks themselves takes place in a look-back manner, e. g., if a node from L_2 is investigated and the direction is **RIGHT**, the predecessors are regarded for alignment. This means, vertices for alignment will be found in L_1.

Lines 9 to 12 check for the presence of the correct kind of neighbours (predecessors or successors, depending on iteration direction) and deliver them in the right order, again with respect to the choice of directions. Of all discovered neighbours, the median two are chosen for a possible alignment. Even if there is only one neighbour, the single neighbour not automatically chosen, but is further investigated as follows.

First of all, the vertex to align may not be part of another block, which is checked in line 14. The lines 15 and 16 use the list of marked type 1 conflicts created in step one to avoid alignment of type 1 conflicting outer segments, thereby solving a type 1 conflict in a way which would prevent a long edge from being drawn straightly. Additionally, the variable r is used to prevent crossing blocks, by disallowing alignment with a vertex which is before (or after, depending on direction) an already aligned vertex in layer ordering.

The lines 17 to 20 update the data structure as described above if a candidate fulfils all requirements for an alignment.

This algorithm is executed four times, once for each possible combination of **LEFT** and **RIGHT** with **DOWN** and **UP**. The graph now has four possible

partitionings in blocks and is ready for a further partitioning into classes in the next step.

Horizontal compaction

The third step of the node placement approach by Brandes and Köpf consists of a further partitioning of the four already present block partitionings and results in four proposals for final coordinates. The latter might be refined by balancing the vertices with respect to their edges by considering all four coordinate proposals.

The general idea is to place the blocks one by one, in an order which is given by the iteration direction. Later blocks have to avoid the already placed blocks by being placed far enough above or below the already placed blocks, preserving the vertex order inside each layer while doing this. During this process, it is checked whether any blocks may be merged to classes, such that the class partitioning can be used for further compaction.

Listing 3.3. Step 3: The horizontal compaction algorithm.

```
1  function place_block(v)
2    if y[v] undefined then
3      y[v] = 0; w = v;
4      repeat
5        if pos(w) > 0 then (DOWN)
6        if pos(w) < |L(w)| − 1 then (UP)
7          u = root[fore(w)]; (DOWN)
8          u = root[foll(w)]; (UP)
9          place_block(u);
10         if sink[v] == v then sink[v] = sink[u];
11         if sink[v] ≠ sink[u] then
12           shift[sink[u]] = min(shift[sink[u]], y[v] − y[u] − δ); (DOWN)
13           shift[sink[u]] = max(shift[sink[u]], y[v] + y[u] + δ); (UP)
14         else
15           y[v] = max(y[v], y[u] + δ); (DOWN)
16           y[v] = min(y[v], y[u] − δ); (UP)
17         end if
18       end if
19       w = align[w];
20     until w = v
21   end if
22 end function
23
```

```
24 initialize sink[v] = v, ∀v ∈ V;
25 initialize shift[v] = ∞, ∀v ∈ V; (DOWN)
26 initialize shift[v] = −∞, ∀v ∈ V; (UP)
27 initialize y[v] = undefined, ∀v ∈ V;
28
29 for i = 0,...,|L|−1 do (RIGHT)
30 for i = |L|−1,...,0 do (LEFT)
31   for k = 0,...,|L_i|−1 do (DOWN)
32   for k = |L_i|−1,...,0 do (UP)
33     v = L_i(k);
34     if root[v] = v then place_block(v);
35   end for
36 end for
37
38 for i = 0,...,|L|−1 do (RIGHT)
39 for i = |L|−1,...,0 do (LEFT)
40   foreach v : v ∈ L_i do
41     y[v] = y[root[v]];
42     if v == root[v] and shift[sink[v]] < ∞ then (DOWN)
43     if v == root[v] and shift[sink[v]] > −∞ then (UP)
44       y[v] += shift[sink[v]];
45     end if
46   end foreach
47 end for
```

(Again, several lines of the horizontal compaction algorithm given in Listing 3.3 are flagged with a mark for the different iteration directions to distinguish the lines to be used in the different cases.)

The first thing to notice is the auxiliary function `place_block(`v`)`, starting in line 1. It provides a first assignment of a y coordinate to a block, which may be changed later on, if the class containing the block is moved due to compaction. The block is placed by starting with v, iterating over all vertices in the block to find the y coordinate closest to 0, whose choice would not result in breaking order or spacing constraints.

To achieve this, the forerunner, or follower in case of **UP** placement, in layer ordering of the current vertex is taken for comparison, called u in the following. Since u might be part of a block which should be above the block of v, but was not placed yet, e. g., because the block of v starts in an earlier layer and the block of u in a later one, the block of u is placed with the `place_block` function first. After that, the block of v is added to the class of u in line 10, if v was not part of a class yet.

If v and u are from different classes, a possible compaction is calculated in lines 12 and 13. If however v and u share the same class, the block of v is placed before or behind the block of u, depending on iteration direction, with respect to the minimum separation constraint δ. When the block of v was the first block to be placed in all concerned layers, the y coordinate is set to 0.

The values of `sink`, describing the root vertex of the highest or lowest block reachable in the block graph, and thus determining the class of a block, `shift`, a value by which a whole class can be moved for compaction, and `y`, the proposed y coordinates, are initialized before the first call of `place_block`, in lines 24 to 27.

After that, `place_block` is called for the root vertices of all blocks (and thus for all blocks) with respect to the iteration direction. The function ignores already placed blocks, which may occur due to recursive call inside `place_block` itself.

The for loop in the lines 38 to 47 finally distributes the coordinates of a block to all of its containing vertices to determine the individual placement. The y coordinate of a block is stored in its root vertex, thus, the y coordinate of the root vertex is assigned to all vertices in the block. Furthermore, the compaction calculated before is applied if the current node is a root node and thus represents a block.

In the original paper by Brandes and Köpf, line 42 read "**if** `shift[sink[root[v]]]` $< \infty$ **then**". That could lead to multiple applications of the class offset and has to be corrected to the version given in Listing 3.3. As mentioned before, the authors also offered an erratum concerning this mistake.

With this algorithm finished for all the four direction combinations, four possible y coordinate assignments for an approximative solution to the node placement problem are present. The solutions normally have good results with respect to straightness and edge length, but are rather unbalanced due to the alignment to either **DOWN** or **UP**. If a more balanced placement is desired, the four solutions can be combined into a single, but balanced node placement.

Balancing

The final step of the original approach by Brandes and Köpf uses the four layout proposals calculated in the steps before to merge them into a single result, with a good compromise between balancing, edge length and edge straightness. To achieve this, the layout proposal with the smallest height is selected as a foundation for the combination. Then, the median position of all proposals is chosen for each vertex and is used to shift the vertices to a balanced position. This method, with a

possible algorithm given in Listing 3.4, provably preserves the order and separation constraints of the graph [BK02].

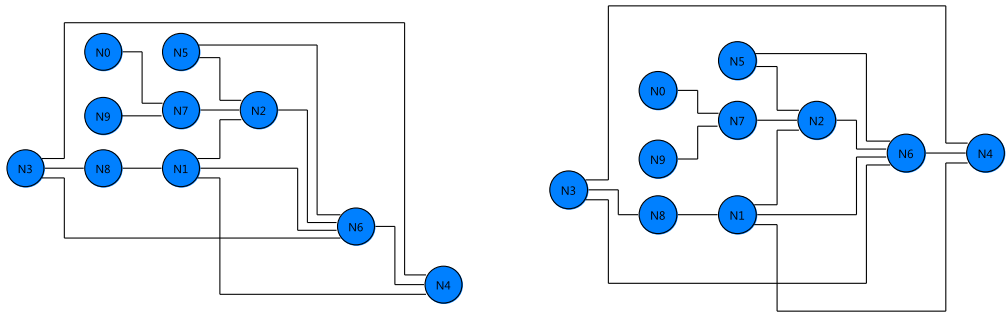
Let $l_{LD}, l_{LU}, l_{RD}, l_{RU}$ be the four proposed layouts, with L, R, D, U representing the four possible directions. Lines 1 to 6 discover the layout proposal with the smallest height and store the minimal and maximal y coordinate for each layout proposal. Lines 8 to 11 determine the diversion of all proposals from the chosen minimal proposal, by checking either the minimal or maximal y coordinate, depending on the compared proposal's iteration directions.

Lines 13 to 19 conclude the balancing algorithm by iterating over all vertices. For each vertex, the coordinates from every layout proposal are taken and the diversion from the minimal proposal is added. The median two coordinates are averaged and used as a new position for the respective vertex.

Listing 3.4. Step 4: The balancing algorithm.

```
1  foreach l : l ∈ {l_LD, l_LU, l_RD, l_RU} do
2    min[l] = min(l.y);
3    max[l] = max(l.y);
4    height[l] = max[l] − min[l];
5  end foreach
6  l_min = l with height[l] == min(height);
7
8  foreach l : l ∈ {l_LD, l_LU, l_RD, l_RU} do
9    shift[l] = min[l_min] − min[l];  (l_LD, l_LU)
10   shift[l] = max[l_min] − max[l];  (l_RD, l_RU)
11 end foreach
12
13 foreach v : v ∈ V do
14   foreach l : l ∈ {l_LD, l_LU, l_RD, l_RU} do
15         localY[l] = l.y[v] + shift[l];
16   end foreach
17   sort(localY);
18   y_bal[v] = (localY[1] + localY[2]) / 2;
19 end foreach
```

With this, the final node placement is given by the map y_{bal}, which maps every vertex to a y coordinate. The original approach by Brandes and Köpf is concluded with this step. An example result of the balancing, in comparison to the unbalanced layout of Figure 3.9a, is depicted in Figure 3.9b. (The given examples were created with the graph drawing tool of the KIELER tool.) However, for the approach to work properly, all vertices have to have the same size, an assumption, which is difficult to hold in a practical application. KLay Layered explicitly supports different



(a) A graph, laid out without the final balancing step, with the directions **LEFT** and **UP**.

(b) The same graph, with the final balancing step applied.

Figure 3.9. The effect of balancing.

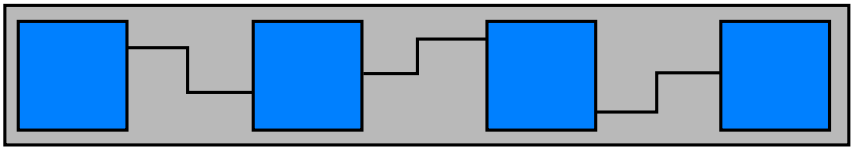


Figure 3.10. Perfectly aligned vertices inside a block, with unnecessary edge bends due to eccentric port positioning.

vertex sizes and thus, further modifications are needed to use this approach in the KLay Layered environment. Furthermore, the usage of port-based graphs poses even more requirements. The following sections will describe the modifications necessary for the support of such features.

3.2.2 Introducing vertex size and ports

One of the core aspects of the approach by Brandes and Köpf is the straightness of lines. As seen before, it is achieved by the alignment of vertices, which leads to having the start and end point of every edge in the block on the same y coordinate. In a practical application, several constraints are required for the approach to work. Vertex size and port positions would have to be uniform across all vertices—otherwise, problems such as unnecessary edge bends and even vertex overlaps

would occur. An example for unnecessary edge bends which occur although the vertices are perfectly aligned is given in Figure 3.10. The problem of vertex overlaps can easily imagined be by mentally enlarging any vertex in Figure 3.9 without modifying the rest of the graph.

It is not possible to allow constraints like a uniform vertex size or ports that are always placed centred on a vertex in KLay Layered. Furthermore, KLay Layered allows connected vertices in the same layer, e. g., in case of inverted ports who introduce a dummy vertex in the same layer. This would lead to vertices placed on top of each other if there are vertices from the same layer in the same block. These problems have to be solved to allow integrating this node placement algorithm into KLay Layered.

Fortunately, the latter problem can be solved rather easily by redefining the *predecessors* and *successors* terms.

Definition 3.5. $pred(u) = \{v(p) : (p,q) \in E, v(q) = u, L(v(p)) \neq L(v(q))\}$ denote the *predecessors* of u in a directed graph. Likewise, $succ(u) = \{v(r) : (q,r) \in E, v(q) = u, L(v(q)) \neq L(v(r))\}$ denote the *successors* of u in a directed graph.

Note that it is now disallowed to have predecessors or successors in the same layer. With that, alignment between vertices of the same layer is prevented, since lines 9 to 12 of Listing 3.2 use these relations to determine vertices for alignment in blocks. This is no problem for the general alignment, since the edges of inverted ports have to have a dummy vertex exactly above the vertex from which they were created and do not need to be aligned in any way with their source or target vertex. Beside that, they are treated as a regular vertex when it comes to alignment with vertices from other layers.

The problems of differing vertex sizes and port positions are more demanding, but it is possible to solve them by integrating an additional step into the node placement algorithm the task of which would be to determine the space needed by every block individually. Furthermore, it should investigate whether the vertices inside a block may be moved in a way that allows edges to be drawn straightly, even if eccentric port positions are present. In addition, the horizontal compaction step would have to be modified to take the results of the new step into account.

The first idea of a naive approach to this problem was to assume a fixed block size with relation to its contained vertices, for example twice the size of the largest vertex or the sum of all vertex sizes. With this, the blocks have enough space, which might also be used to shift the other vertices inside a block up or down to get the edges as straight as possible. Although this approach is very easy to implement,

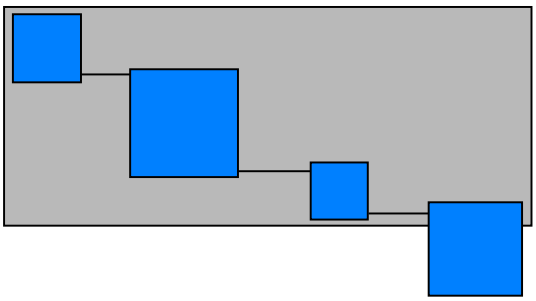


Figure 3.11. A block with double the size of its largest vertex that is still not large enough to reserve enough space for alignment, due to inauspicious port positions.

there are several disadvantages to it. On the one hand, the approach might lead to a lot of wasted space, since so many space should only rarely be required. On the other hand, if the port positions are really inauspicious, this approach might lead to vertices placed outside the bounds of the containing blocks, as the example in Figure 3.11 shows.

The naive solution to the latter problem, using the sum of all vertex heights as block size, would result in even more wasted space, and can thus be discarded.

A more feasible solution is to calculate the actually required space subject to vertex size and correct alignment. This is done by comparing the port positions of neighbouring vertices and shifting them such that the ports have the same y coordinates. The shift distance and the size of the vertices is then used to calculate the overall space needed by the block. This step has to be done for all the four iterations which result from the different combinations of layout directions, since the partitioning into blocks differs depending on the respective choice of direction.

An algorithm for solving the problem in this way is given in Listing 3.5. The two values to be calculated (shift and block size) are stored in maps, initialized in line 1 and 2. A so called *inner shift* determines the distance by which the vertices have to be moved relative to their calculated final y coordinate to have their port on the same height as the connected neighbour in the same block. The block size is associated with the root vertex of a block and is chosen to be as small as possible, with vertex sizes and shifting in mind.

The for loop starting in line 4 iterates over all root vertices, and thus over all blocks of the current partitioning of the graph. Initially, the height of the block stored in h_{Block} is set to the height of the root vertex. For accessing necessary values in the pseudo code, let $height : V \rightarrow \mathbb{R}$ deliver the vertical dimension of a vertex

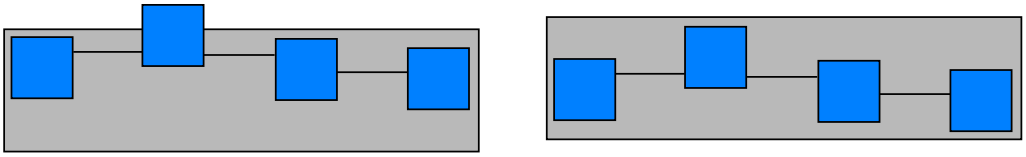
and $portPos : P \rightarrow \mathbb{R}$ the vertical position of the port, relative to the upper left corner of its vertex. The top border of the block, t_{Block}, is set to 0, the lower border, b_{Block}, is also set to the height of the root vertex.

Listing 3.5. Step 2.5: The inside block shift algorithm.

```
1  initialize inner_shift[v] = 0, ∀v ∈ V;
2  initialize block_size[v] = 0, v ∈ {root[u]|u ∈ V};
3
4  foreach v : v ∈ {root[u]|u ∈ V} do
5    h_Block = height(v);
6    t_Block = 0; b_Block = height(v);
7    x = v; y = align[x];
8    if ∃e = (p,q) : v(p) = x and v(q) = y and v(p) ≠ v(q) then
9      t_Block = portPos(p);
10     b_Block = height(x) − portPos(p);
11   end if
12   while y ≠ v do
13     e = (p,q), v(p) = x, v(q) = y;
14     inner_shift[y] = portPos(p) − portPos(q)+ inner_shift[x];
15     t'_Block = portPos(q);
16     b'_Block = height(y) − portPos(q);
17     if t'_Block > t_Block then t_Block = t'_Block;
18     if h'_Block < h_Block then h_Block = h'_Block;
19     x = y; y = align[y];
20   end while
21   if align[v] ≠ v then h_Block = t_Block + b_Block;
22   x = v; y = align[x];
23   if t_Block > portPos(p), e = (p,q) : v(p) = x and v(q) = y then
24     inner_shift[v] = inner_shift[v] + t_Block;
25     while y ≠ v do
26       inner_shift[y] = inner_shift[y] + t_Block;
27       x = y; y = align[y];
28     end while
29   end if
30   block_size[v] = h_Block;
31 end foreach
```

The auxiliary variables x and y are used for iterating over the block's vertices, starting with the root vertex and the possible next vertex in the block alignment. If the block only consists of one vertex, x is equal to y.

Before the iteration starts, lines 8 to 11 make sure that the first two ports connected to the root vertex and the first aligned vertex are taken into account for the calculation of the block size, if there are more vertices than the root vertex in



(a) A block with no post shift applied.

(b) The same block using post shift to "push" the vertices inside the boundaries of the block.

Figure 3.12. When no post shift is applied, vertices may extend the boundaries of a block.

the block. This is necessary because if the block only consists of the root vertex, the top and bottom border of the block are the same as the root vertex's boundaries. If shifting is applied, which is the case if there is more than one vertex inside the block, the boundaries of the block are relative to the port positions.

During the iteration, the edge that connects the vertices x (through port p) and y (through port q), is determined. Such an edge always exists since the vertices of a block, which are in neighbouring layers, have at least one edge that connects them. In case of a self-loop, the edge is ignored. The difference in vertical position between the two ports is calculated and stored as inner shift for y, with the inner shift of x as an additional offset. After that, lines 15 to 18 check the boundaries of vertex y and adjust the block's borders if necessary. Line 19 completes the iteration by moving the pointers forward. The loop stops once y points to the root vertex.

If the block consists of more than one vertices, the block size is determined in line 21 by adding the sum of the sizes of the top and bottom border. Finally, if the top border of the block extends higher than the root vertex, a post shift has to be applied since the placement of all vertices in a block depend on the root vertex. The block's vertices are moved down by the height of the top border. This is because the border was determined from the greatest distance between a port and a vertex's upper border. The post shift makes sure that this vertex is not placed outside of the block's boundaries, as shown in Figure 3.12.

With this method, vertex size and port positions are easily integrated into the node placement approach by adding the algorithm from Listing 3.5 as an intermediate step between the vertical alignment and the horizontal compaction steps. However, minor adjustments to the horizontal compaction algorithm have yet to be made to include the block size into the calculation of the concrete positioning of a block. Listings 3.6 and 3.7 show the lines from Listing 3.3, modified for taking the newly introduced block size into account.

Listing 3.6. Modifications to the class compaction calculation from step 3.

```
12 shift[sink[u]] = min(shift[sink[u]], y[v] − y[u] − blockSize[u] - δ); (DOWN)
13 shift[sink[u]] = max(shift[sink[u]], y[v] + y[u] + blockSize[v] + δ); (UP)
```

Listing 3.7. Modifications to the block position calculation from step 3.

```
15 y[v] = max(y[v], y[u] + blockSize[u] + δ); (DOWN)
16 y[v] = min(y[v], y[u] − blockSize[v] - δ); (UP)
```

With these modifications, the node placement approach by Brandes and Köpf is enhanced with the abilities of dealing with vertices of any size and of aligning vertices in a way which yields straight edges regardless of port positioning. The runtime performance of the additional algorithm is obviously linear, since there is only one iteration over every vertex, with respect to the block partitioning. Especially the ability of handling larger, differently sized vertices can be of great importance in a practical application. Some tools, e. g., KIELER, allow nested graphs, to express the complex interior of certain vertices. These vertices tend to become very large, to keep their content readable. With the presented modifications, diagrams of this kind can be used in connection with the approach by Brandes and Köpf. An example, taken from the KIELER Actor-Oriented Modelling (KAOM) editor, laid out with KLay Layered with this new approach already integrated, is given in Figure 3.13. The meaning of the diagram is of no further importance; what is of importance is, that many different vertex sizes can be handled.

Besides the enhancement of the functionality of the node placement approach, it is also possible to modify or enhance this implementation of a phase of the KLay Layered algorithm on a different level. The following section will deal with such meta modifications, coping with the effect of balancing on the number of edge bends and how to handle difficult and error-prone graph constellations.

3.2.3 Balancing and fault tolerance

Section 2.3 contains a list of criteria for evaluating different aspects of a layout, especially focussing on the aesthetics and readability of a graph. Optimizing a layout for all of these goals is difficult to say the least, since most of them contradict each other. In the following, the focus will lie on two such criteria: the straightness of edges and te balancing of outgoing connections. Section 3.1 already reasoned that there is a conflict between these two criteria: in unfortunate cases, the mathematical optimum of the two may result in a compromise which fulfils both only partly. The

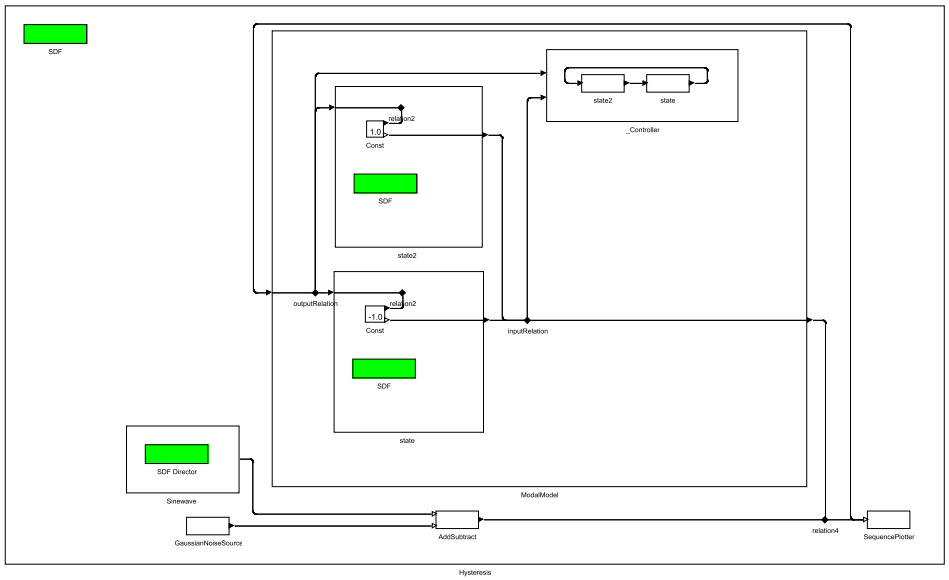


Figure 3.13. A nested graph with large vertices.

result is slightly odd edges and a slight unbalance. It may be even worse in the case of orthogonal edge routing, since every odd edge introduces unwanted edge bends.

Instead of an unsatisfying compromise, one usually wants to choose one criterion over a conflicting other criterion. It may also be a favourable way to let the user decide what is more important to him. Because of that, the flow structure of the original approach by Brandes and Köpf was changed to introduce more freedom of choice to the user. Instead of using the fourth balancing step as a fixed part of the node placement phase, it is now offered as a feature which may be turned off and on at will. With this, the user can decide whether he wants to favour balancing over edge bends and vice versa.

The effect of this choice can already be seen in the relatively small example in Figure 3.9. The solution with focus on edge straightness contains twenty-two edge bends, while the balanced solution has twenty-eight bends, or approximately thirty-six percent more.

To provide this choice, the fourth step of the algorithm was decoupled and a short decision algorithm was integrated to decide whether the balancing step should be applied. If the user chooses to reduce edge bends, the four aligned

layouts are used as possible results, since their block structure and the additional alignment inside the blocks favour straight edges. The decision between these four layout candidates must be made by another metric. In the given approach, the overall size of the layout is chosen as a metric, but it is also possible to use the overall number of edge bends as a decision criterion. The layout size was only chosen as a criterion to avoid too much white space, since the block alignment tends to have an especially strong effect on the height of a graph.

Another advantage that comes with this structure is that the node placement algorithm becomes more robust. With balancing activated, the decision algorithm has five potential layout candidates from which it can choose the final node placement result. In the case of an activated balancing, the balanced result should of course be favoured. But if, for any reason, the balanced solution breaks a constraint, the decision algorithm has still four layout candidates left from which to pick a valid solution. This makes the node placement phase more robust, since at least four placement proposals have to be wrong for the final result to be erroneous. This was originally necessary to avoid errors which the original approach tended to make, due to the mistake mentioned and corrected above. In the current version, the check could be left out, but since the check runs in linear time, it is a welcome source of further stability.

This concludes the description of the approach to the node placement problem chosen in this thesis. The next section will now deal with certain details of the implementation, regarding specialities of KLay Layered or code twists going beyond the level of detail of the already presented pseudo code.

3.3 Implementation

After the long and detailed description of the node placement approach and the ideas behind it, the following section will deal with technical aspects. Although they will seldom deliver a concrete contribution to the approach itself, they are still essential for a working implementation of the presented ideas.

The first part of the section will cope with the specialities and logistics of including the given approach into the KLay Layered algorithm. After that, a data structure for layout candidates is presented and motivated. Finally, the section concludes by presenting ways of dealing with special graph structures and a general overview of a full execution of the complete node placement algorithm.

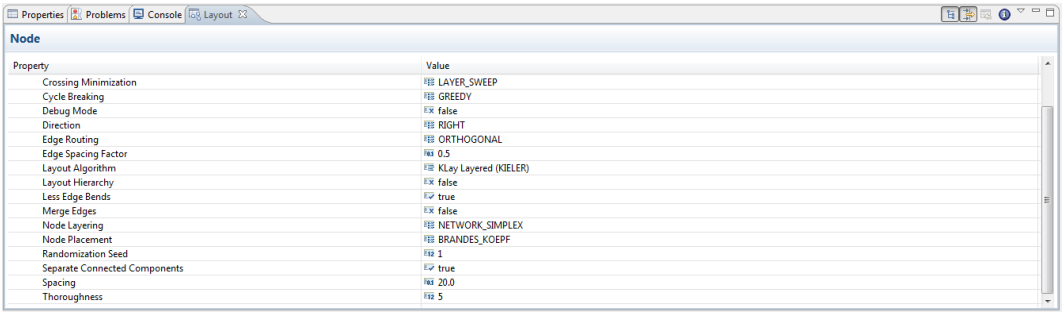


Figure 3.14. Some layout options for KLay Layered, especially the node placement selection and some other relevant options.

3.3.1 Implementation in the context of KLay Layered

Being a part of the KLay Layered layout algorithm, the newly created node placer has to be integrated into its source code structure. The choice of the right package in the Java source tree is fortunately obvious, due to the clear structure of KLay Layered. Every phase of the algorithm, as listed in subsection 2.4.1, has its own sub package. The new node placer is thus included in the package for phase four, `de.cau.cs.kieler.klay.layered.p4nodes`.

The implementing class itself is called `BKNodePlacer`, in reference to the authors of the basic approach, Ulrik Brandes and Boris Köpf. It extends `AbstractAlgorithm`, the abstract base class of algorithms in the KIELER project offering a reference to the currently used progress monitor. The progress monitor can be used to give the user visible feedback on the steps already done and steps to follow for the current task. Furthermore, the `ILayoutPhase` interface is implemented which is required by every phase of the KLay Layered algorithm. It declares methods for the layout itself and the required intermediate processors.

This node placer can be chosen by the user via a so called *layout option* in the KIELER tool. The node placer has to be selected from a list of other implementations for phase four, as shown in Figure 3.14. Additionally, the results of the node placement can also be influenced by these options. For example, a balanced or unbalanced layout can be chosen, or the minimum separation constraint δ (called *spacing* in KIELER) can be changed.

On the code level, the implementation has to differ slightly from the pseudo code given in Section 3.2. Normally, these differences are only small; for example, the position of a port in the graph data structure, the so called `LGraph`, is relative to

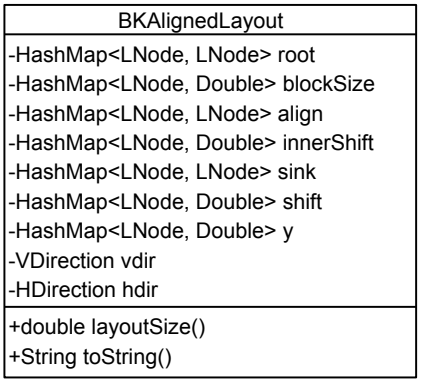


Figure 3.15. Class diagram of the `BKAlignedLayout` class, storing information about a layout candidate.

its vertex and consists of two components, the position and the anchor. They have to be added to get the final position of the port. Other minor differences concern loops: sometimes a for-each loop is chosen over a classical for loop. This is, because for the different iteration directions, some lists have to be traversed reversely. It is more tidy to reverse a list before a for-each loop than duplicating the whole loop with the only difference being the iteration parameters.

More important than these minor differences is the distinction between source and target of an edge in the `LGraph`. Especially with the switching iteration directions **LEFT** and **RIGHT**, the algorithm has to be aware of the direction and correctly choose between source and target of an edge.

3.3.2 Data structures

As seen in Section 3.2, up to five layout candidates are produced when executing the presented node placement approach. During the execution itself, and also when checking and perhaps balancing the layout, the numerous intermediate and final results for each of these four to five layouts have to be stored. There exist several ways for doing this, such as adding a dimension to each of the maps for addressing the different layout candidates. One could also think of creating individual maps for each candidate, since the number of candidate is fixed. Nevertheless, both solutions lack flexibility or are even error-prone, e. g., when five candidates are assumed in the code, but the fifth was not calculated, resulting in a null pointer or index out of bounds exception.

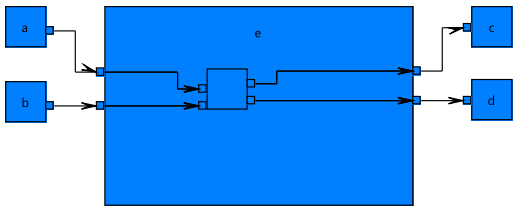


Figure 3.16. Different hierarchy levels in a graph, hierarchy crossings are done with ports.

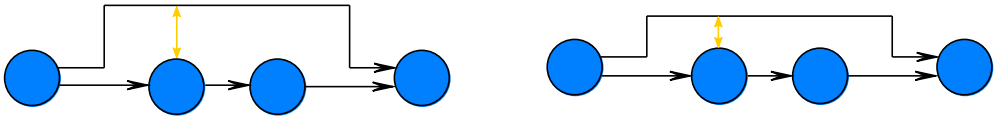
To have a safer and more dynamic way for storing results the `BKAlignedLayout` class, as given in Figure 3.15, was created. It stores all the intermediate and final results needed for a layout candidate and offers convenience methods for getting the size of a layout or the name of a candidate, derived from the choice of directions. With this, the number of candidates can be adjusted as needed by adding the `BKAlignedLayout` representations of all candidates to a list. The final candidate can then be picked from this list, or it can be created by using the list's content to create a balanced layout as described above.

3.3.3 Support for complex structures

KLay Layered is able to deal with a large variety of different graph types. Many different special structures in graphs have to be supported, in the least-invasive way possible. This means that the algorithms themselves should only have to be changed as little as possible, which can be done by mapping the complex structures onto simpler structures that can already be processed by the algorithms.

An example already given in the context of Section 3.2 are hierarchical vertices. In graphical modelling, hierarchical vertices are used to display the encapsulation of another model within a vertex of the current model. This allows the user to follow the events in an encapsulated model without switching windows, but requires the layout algorithm to support hierarchical vertices. At first, a trivial but effective solution comes to the mind. The layout is simply executed recursively, treating every new level of hierarchy as an individual graph and regarding the hierarchical vertices as normal, albeit large, vertices on their own hierarchy level.

The problems of this approach start once the connection of vertices on a higher hierarchy level span into other hierarchy levels. One possibility to solve the problem of inter-level-transitions are ports which connect the inner model to the outside model on the higher hierarchy level. An example for that is also given in Figure 3.16.



(a) A graph without edge compaction.

(b) The same graph, with an edge spacing factor of 0.5.

Figure 3.17. The effects of edge compaction using an edge spacing factor.

According to Christoph Daniel Schulze, these cases can be solved by introducing dummy vertices for such hierarchical ports [Sch11]. These dummy vertices can then be dealt with just like every other vertex during node placement.

To allow the insertion of dummy vertices for hierarchical ports, the intermediate processor `HierarchicalPortPositionProcessor` has to be marked as required by the node placement algorithm.

When ports are not present for inter-level-transitions, another approach has to be found. A feasible solution to this problem in KLay Layered was developed by Insa Fuhrmann [Fuh12]. To allow this method to work with the newly created node placer, the node placer must make sure that certain side protection dummies are included in the same block, to keep them straight. This is achieved by prioritizing the alignment of these compound side vertices.

This priority is also changeable for every edge in the graph to allow customization of the layout. Every edge can be assigned a priority which expresses the user's desire to have this edge drawn straightly. The algorithm will then favour the edge with the highest priority in the alignment, if the boundaries of the algorithm allow it.

Less a question of a special structure and more a problem of aesthetics and wasted space is the routing of long edges that are close to another edge or a block. Normally, the minimum separation constraint is used to leave enough space around the vertices of a layout. In the case of an edge close to other graph elements, the application of the minimum separation constraint may result in giving the edge way more space than it would need for a tidy and readable layout. To prevent this waste of space, an *edge spacing factor* was introduced. It allows to place two blocks closer to each other, by reducing the minimum separation constraint by the given factor. The test whether two blocks may be placed closer to each other than the minimum separation constraint takes place within the `place_block(`v`)` function

and depends on whether one of the two blocks only consist of dummy vertices or also includes regular vertices. Figure 3.17 shows an example for the usage of the edge spacing factor.

3.3.4 Handling of type 2 conflicts

As mentioned before, the handling of type 2 conflicts, that is, the crossing of two inner segments as described in Section 3.2, has to be thought about since the original approach by Brandes and Köpf assumes that there are no type 2 conflicts present. To keep the algorithm simple, rare type 2 conflicts are treated in a greedy fashion in the current implementation. The first conflict member in iteration direction is aligned, resulting in the latter conflict member to not be drawn straightly. This is also the most general solution, allowing the algorithm to stay untouched, e. g., when the crossing minimization phase is optimized to prevent type 2 conflicts in the future.

3.3.5 Steps to a node placement choice

To conclude the section about implementation details, Figure 3.18 shows the overall process of the newly implemented node placer in the practical application.

In the beginning, the algorithms described in Section 3.2 are executed, with vertical alignment, shifting and horizontal compaction applied to each of the four iteration direction combinations separately. After that, the additional balancing on the foundation of the four results is executed, if desired by the user. Then, the four or five results are checked for their validity, which is currently done by checking whether all placed vertices fulfil the order and minimum separation constraints. Every valid layout candidate is then used as an option in the final choice. If balancing is activated and the balanced layout candidate is valid, it is chosen without further hesitation. Otherwise, a metric is applied for choosing the final candidate. In the current implementation, the metric is based on the vertical size of the candidates, choosing the most compact of the candidates.

Once the choice is made, the coordinates from the final candidate are applied to the graph data structure. In this step, the overall height of the graph and the width of the layers is also determined. After this is finished, the graph data structure is passed to the final fifth phase, the edge routing.

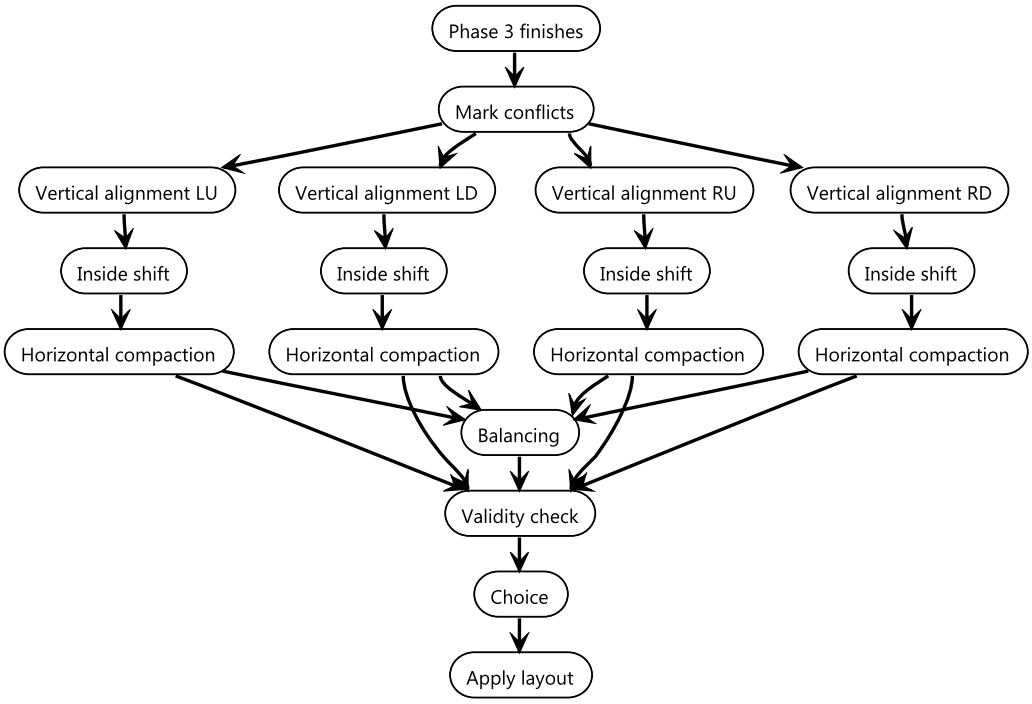


Figure 3.18. The process of phase four with the new node placer.

This last summary of the whole node placement process concludes the implementation section. The final section of this chapter will evaluate results from the new node placer and will compare them to results of other node placers.

3.4 Evaluation

Evaluating the aesthetics and looks of a layout is a non trivial task. One does not simply take a set of diagrams and decide on their quality on one's own, since the requirements a person has to a good layout may be strongly subjective. To solve this problem scientifically, different approaches exist. An user oriented approach is the evaluation of layout quality by running a study with a statistically significant amount of test subjects, e. g., a group of students or users of graphical modelling applications.

A simpler but effective and especially objective approach to evaluation is a choice of objective metrics affected by the layout algorithm under investigation. These are then calculated for a large set of diagrams and related to features of these diagrams, e. g., the number of vertices or edges. This approach has two main advantages that made it more attractive for this thesis: First, a larger amount of test diagrams can be evaluated in an acceptable time frame, and second, this type of evaluation can generally be employed simpler and with no overhead due to people management. Furthermore, the results cleanly represent the layout algorithm's performance in the measured fields, and does not simply show whether the users preferred this or that diagram.

To begin this kind of evaluation, the metrics to be investigated have to be chosen. Thus, an overview of possible metrics will be presented, followed by a reasoned choice of the metrics to be evaluated later on. Keep in mind that an orthogonal edge routing will be assumed since this is common for the field of application of KLay Layered, data flow diagrams.

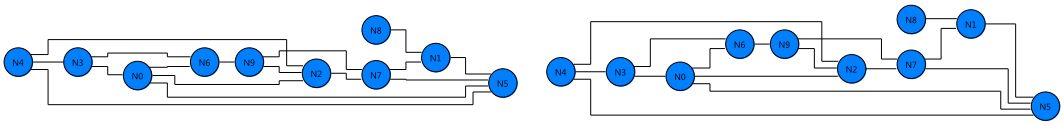
3.4.1 Evaluation metrics

In a drawn graph, many characteristics can be evaluated. Some of them should not appear in a layout result at all, such as vertex overlaps or overlaps of edges and vertices, because such overlaps make a diagram very difficult to read. Other metrics do not necessarily have an impact on whether a diagram is virtually unreadable or not, but can be used to make qualified statements about the performance of a layout algorithm in a certain field. The following list contains the aesthetics criteria found by Sugiyama et al. [STT81] and by Di Battista et al. [DETT99]:

- **Flow direction**: Describes the directions of the graph's edges. It is desired to have the majority of edges pointing to the same direction in order to emphasize the data flow aspect of a diagram.

- **Edge crossings**: With a certain amount of edges, edge crossings become unavoidable. Nevertheless, it is desirable to have a minimal number of crossings, since they influence a diagram's readability in a negative way.

- **Edge bends**: Evaluates the number of bends of an edge. This corresponds to the straightness criterion of Sugiyama et al., since non-straight edges introduce at least two bend points in orthogonal layout.



(a) A graph laid out with the linear segments node placer.

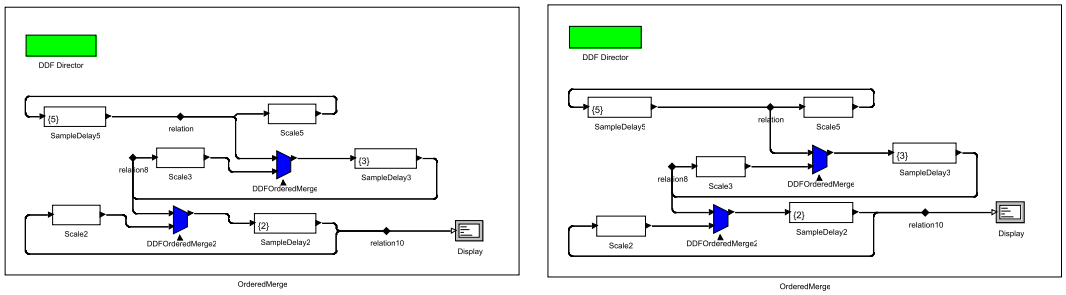
(b) The same graph laid out with the Brandes Köpf node placer.

Figure 3.19. A graph with different algorithms used for node placement.

- **Edge length**: The longer an edge is, the harder it is to follow while reading the diagram. Hence, the average of the length and the maximum edge length can be used as a metric.
- **Drawing area**: This represents the overall size of a graph, by multiplying the expansion of a graph in vertical and horizontal direction. In general, a smaller drawing is easier to read and to view on a screen or printing.
- **Aspect ratio**: The aspect ratio shows the relation between horizontal and vertical expansion, with the goal of yielding a ratio common to paper or screen formats.
- **Symmetry**: Symmetry in the graph should be emphasized in the layout, a very difficult criterion to satisfy. Symmetric drawings of a graph make it more easy to recognize structures.
- **Balancing**: Balancing as proposed by Sugiyama et al. refers to the relation between the highest and lowest vertices connected to a given vertex, as already presented in Section 3.1. This is connected to symmetry, since a balanced layout has the chance to introduce local symmetric structures.

For the evaluation of the newly implemented node placer, a subset of these metrics has to be chosen. This is because not all criteria are equally important to the readability of data flow diagrams and, more importantly, the node placement phase does not influence all of them. Edge crossings for example are determined by the vertex order computed by the crossing minimization phase right before node placement. The node placement does not influence the number of crossings, so they need not be included in the evaluation of a node placer.

Besides selecting the specific metrics to be evaluated it is also important to think about which algorithm to compare the new node placer with. However, that choice is easily made since the direct rival of the new node placer is found within the own



(a) A Ptolemy II diagram laid out with the linear segments node placer.

(b) The same Ptolemy II diagram laid out with the Brandes Köpf node placer.

Figure 3.20. A Ptolemy II diagram, displayed with the KAOM editor, with different algorithms used for node placement.

ranks of KLay Layered in form of the *linear segments node placer*. Figure 3.19 shows example results from both node placers in a graph editor, while Figure 3.20 does the same for a Ptolemy II[2] diagram displayed in KIELER's KAOM editor. It can be clearly seen that the results of the new node placer have more straight edges, but need more space, especially in vertical direction.

With these examples given, the following four criteria for an evaluation are chosen, and a first estimate of a possible result is given, based on experiences with and expectations of the node placers:

- **Edge bends**: As this is one of the core purposes of the new node placer, it is expected that it will yield significantly less edge bends than the linear segments node placer.

- **Edge length**: The usage of blocks in the new node placer leads to a slightly increased vertical distance between vertices. On the other hand, saving edge bends results in shorter edges. The new node placer is expected to perform about as good as the linear segments node placer.

- **Drawing area**: As mentioned before, the new node placer needs more vertical space. The linear segments node placer will use less space than the new node placer.

[2] `http://ptolemy.eecs.berkeley.edu/ptolemyII/`

- **Aspect ratio**: In the work of Schulze [Sch11], the linear segments node placer had an average aspect ratio result of 2 to 2.5. Since the new node placer needs more vertical space, it is expected to have a lower aspect ratio.

As another common metric in algorithm engineering, the computational performance of an algorithm can be investigated. This will briefly be done, to check whether the usage of the new node placer has any impact on the runtime of KLay Layered. Since most of the sub-algorithms of the Brandes Köpf node placer run in linear time, it is not expected to have any significant improvements or debasements when using it instead of the linear segments node placer.

3.4.2 Evaluation models

With the candidates and the metrics found, a method of measurement and a set of graphs and diagrams are the only things missing for an evaluation. Fortunately, implementations of the given metrics in the KIELER framework were already done in the work of Schulze mentioned above and in a work on graph analysis by Martin Rieß [Rie10]. Graphs for the evaluation will be taken from three different sets: a variety of randomly created, portless graphs, a variety of randomly created port-based graphs, and a collection of Ptolemy II KAOM models taken from the repository of demo models shipping with Ptolemy II. All sets consist of small and medium-sized graphs to cover a wide field of layout problems. The set of Ptolemy II models also contains some larger graphs. In the case of the randomized graph, vertex counts from ten to fifty will be used, with the number of outgoing edges on each vertex fixed to three, such that the vertex count implicitly determines the edge count, efficiently ruling out a source of unwanted influence. Note that an effective average of three outgoing edges is a rather high value. Table 3.3 shows a short overview over the characteristics of the used graphs. Table 3.4 shows the relevant layout options with respect to the different node placers. Most of the options are shared and equal, to ensure a fair comparison. The algorithms in the other phases are, of course, the same for both node placers, using the network simplex layerer in phase two and an orthogonal edge routing in phase five.

As one can easily imagine, the random graphs do not resemble real world data flow diagrams, so the basic idea of evaluating the node placers' performance there is to get a general comparison between both. The results from the evaluation with the models from the Ptolemy II domain promise to get results closer to a real world application.

Table 3.3. Properties of the evaluation model sets, denoting the minimum and maximum count of each value, followed by an average given in brackets.

	Portless	Port-based	Ptolemy
Diagrams	170	170	300
Vertices	10–50 (29.8)	10–50 (29.8)	2–425 (30.95)
Compound vertices			1–35 (3.65)
Compound vertex children			0–58 (5.81)
Vertex degree	3–14 (6)	3–12 (9.6)	0–29 (1.99)
Edges	30–150 (89.29)	30–150 (89.29)	1–648 (37.83)
Self-loops			0–6 (0.45)

Table 3.4. The layout options chosen for the two node placers.

	Linear segments	Brandes Köpf
Border spacing	20	20
Edge spacing	0.5	0.5
Spacing	20	20
Balanced		false

3.4.3 Evaluation results

With this, everything is ready to finally dive into evaluation. The results will now be presented textually and graphically for each model set and node placer. A table will show the average overall results for each metric, each node placer, and each model set, while graphical charts will show the relationship between vertex count and the different metrics for each diagram type. The rightmost column shows the improvement of the new node placer in comparison to the linear segments node placer in percent. A negative value indicates that the linear segments node placer achieved a better result. In terms of aspect ratio, the distance to a presumably desired aspect ratio of 1 is taken to measure the improvement.

The first results are from the evaluation of the set of portless models and are given in Table 3.5. The results roughly conform to the expectations from subsection 3.4.1.

In terms of edge bends, the new node placer performs about 11% better as the linear segments node placer. While this is already a significant result, the improvement was expected to be a little higher. The rather small improvement

Table 3.5. The results of the evaluation of the portless model set, with averaged results given.

	Linear segments	Brandes Köpf	Improvement
Edge bends	255	229	11.4%
Average edge length	616.65	655.64	−6%
Diagram area	1,380,648	1,727,429	−20%
Aspect ratio	2.6	1.87	84%

Table 3.6. The results of the evaluation of the port-based model set, with averaged results given.

	Linear segments	Brandes Köpf	Improvement
Edge bends	375	293	28%
Average edge length	1,514.04	1401.31	8%
Diagram area	5,492,264	4,719,196	16.4%
Aspect ratio	1.12	0.82	−33%

values might result from the synthetic random graphs, that are without ports and thus pretty simple.

The average edge length results are exactly as expected: very similar with a slight advantage for the linear segments node placer, that creates slightly shorter edges on average. The same holds for the diagram area in which the new node placer uses significantly more space, just as predicted.

The aspect ratio of diagrams created by the new node placer is very good, if an aspect ratio of 1 is regarded as desired. When considering common page and screen sizes, the Brandes Köpf node placer scores even better, since the average result is very close to common ratios like 4:3 (1.3), 16:9 (1.7) or the A4 paper ratio (1.4).

The second result set given in Table 3.6 represents the results from the evaluation of the port-based model set. In this result, some surprises occur with respect to the expectations and the results from the previous model set.

The improvement in the edge bend metric in this second result set is significantly higher, with the Brandes Köpf node placer achieving an improvement of 28%. This fits the improvement expectations better than the previous result and seems to confirm the guess, that the new node placer performs better in more complex diagrams.

Table 3.7. The results of the evaluation of the Ptolemy II model set, with averaged results given.

	Linear segments	Brandes Köpf	Improvement
Edge bends	46.74	34.49	35.5%
Average edge length	245.54	262.62	−7%
Diagram area	2,346,718	2,826,962	−16.9%
Aspect ratio	2.43	2.13	26.5%

The first surprise occurs in the average edge length metric. While the new node placer was expected to score slightly worse, than the linear segments node placer, the results show a performance which is the other way round, with the Brandes Köpf node placer performing slightly better.

An even bigger surprise unveils itself in the diagram area metrics. Here, the Brandes Köpf node placer uses 16.4% *less* space than the linear segments node placer, hinting that the new node placer is more space-effective in complex diagrams.

A trade-off for this seems to be the aspect ratio. While the results were pleasing with the set of portless models, the aspect ratio in port-based random diagrams is significantly worse than the result of the linear segments node placer. The investigation of some graphs showed that a strong presence of northern or southern ports led to the linear segments node placer creating some kind of stair effect, causing the diagrams to be larger in horizontal and vertical direction and closer to a quadratic drawing. The results of the Brandes Köpf node placer are more compact, but with a tendency to grow faster in vertical direction.

The final result set, given in Table 3.7, represents the results from the evaluation of the model set containing Ptolemy II diagrams. The results from this model set fit the expectations and contain no surprises. Since these diagrams are from a real world application, it is pleasing to see the expectations matched here.

The improvement in terms of edge bends is the highest from all model sets, with an average reduction of edge bends by 35.5%. It seems that the methods implemented to reduce edge bends work best in a complex real world environment, where vertex sizes are variable and hierarchy levels are present.

The average edge length and the diagram area evaluate as expected, with the new node placer performing slightly worse in the average edge length metric and significantly worse in terms of overall space usage.

As in the set of portless models, the aspect ratio of the results created by the Brandes Köpf node placer is better, but not as good as in the portless evaluation. This most likely results from the compound vertices containing parts of the model, because they use additional diagram space by being drawn around the graphs and model parts. This did not occur in the set of portless models.

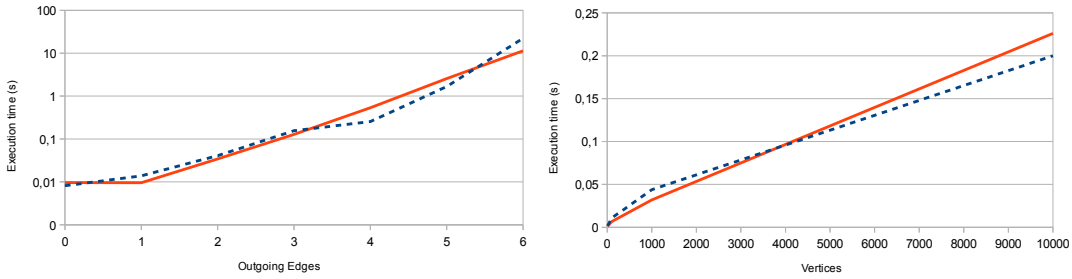
Although some positive surprises occurred in the evaluation of the port-based model set, the general impression is that the expectations posed in subsection 3.4.1 were met in both a synthetic testing environment and in real world diagrams. This gives the user of KLay Layered the choice between different focusses in his layout tasks. When the user desires a small drawing with shorter edges, the linear segments node placer is a good choice. If straight edges and an aspect ratio close to common screens and paper sizes is desired, the new Brandes Köpf node placer may be the node placer of choice.

Another aspect to be investigated was the computational performance of the two node placers. This was done by using a tool developed by Spönemann and Schulze [Spö09, Sch11], which uses randomly created graphs similar to those used in the sets of portless and port-based graphs. The tool uses a K-best runtime analysis procedure, evaluating the runtime for every graph five times and taking the best result, to rule out cache and clock effects.

To investigate the effect of different graph structures, two sets of graphs with different properties are evaluated. Both sets are port-based and can contain self-loops. The first set consist of graphs with a vertex count fixed to 100 and a fixed number of outgoing edges per vertex which lies between 0 and 6. The second set has a variable count of outgoing edges per vertex which lies between 0 and 2, and vertex counts that vary between 10 and 10000. Keep in mind that the graphs are not the same for the two node placers, but share the same properties. Of course, both runs use the same properties and algorithms, except for the node placement phase. The reference processor is an Intel Core i5-2520M @ 2.50 GHz running 64-bit Microsoft Windows.

The results of the performance evaluation can be seen in Figure 3.21. The plot of the evaluation of an increasing number of outgoing edges shows that both node placers perform equally well up to an above average edge count of 4. After that, the linear segments node placer performs slightly better for an edge count of 5, but starts to take significantly more time in the following. This result suggests that the new node placer performs better if the edge count is insanely high.

The second chart shows the opposite behaviour in terms of increasing vertex count in a graph. Up to 4000 vertices, the new node placer performs slightly better,



(a) The performance with a fixed vertex number and increasing edge count. Note that the y axis is scaled logarithmically.

(b) The performance with an increasing vertex count.

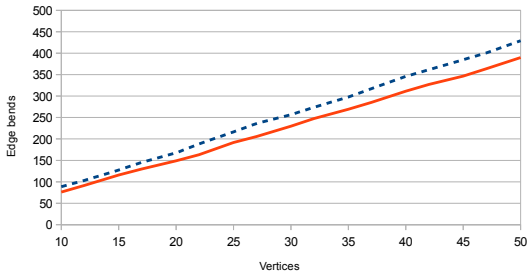
Figure 3.21. The performance comparison of the two node placers. The dashed line represents the results of the linear segments node placer, the solid line the results of the Brandes Köpf node placer.

but the linear segments node placer starts to outperform it after that. All in all it can be stated that the expectation of both node placers performing equally well for graphs of a reasonable size was met.

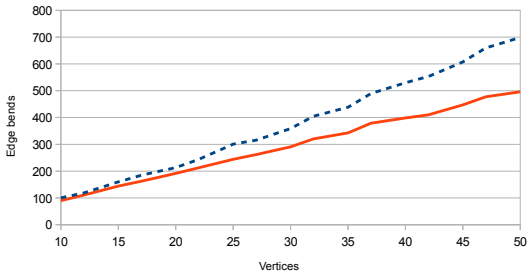
To conclude the evaluation section and thus the node placement chapter, two final figures will be investigated and compared between both node placers: the relation between the diagram size, represented by the number of vertices, and the performance in the edge length and edge bend metrics.

The results for the edge bend metric are given in Figure 3.22. The charts for the portless and port-based model sets show smooth curves, which may be a due to the synthetic random creation of the graphs. The charts show the expected results, with the number of edge bends rising linearly for both node placers (with the linear segments node placer denoted by the dashed line), but with the bend count of results of the new node placer rising more slowly, resulting in better scores with more vertices contained in a graph.

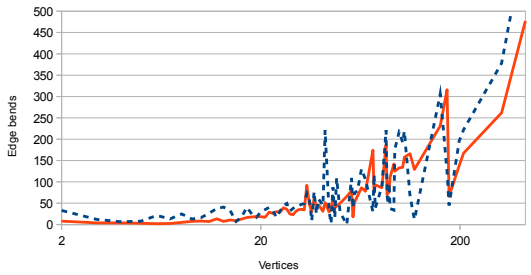
The plot of the results of the Ptolemy II contains more spikes and local maxima due to the nature of the diagrams. The diagrams show very different models and were created by humans, resulting in a more eccentric curve. Note that there are some diagrams where the linear segments node placer produces less edge bends than the new node placer. Nevertheless, the general expectations are met here as well, with the Brandes Köpf node placer producing less edge bends in many diagrams.



(a) The edge bend metric in the set of portless models.



(b) The edge bend metric in the set of port-based models.



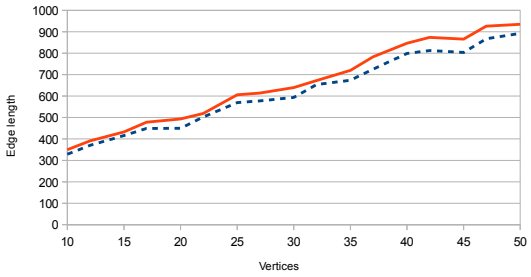
(c) The edge bend metric in the set of Ptolemy II models. Note that the x axis is scaled logarithmically, to improve the readability of the results for small graphs.

Figure 3.22. The edge bend metric in the different model sets. The dashed line represents the results of the linear segments node placer, the solid line the results of the Brandes Köpf node placer.

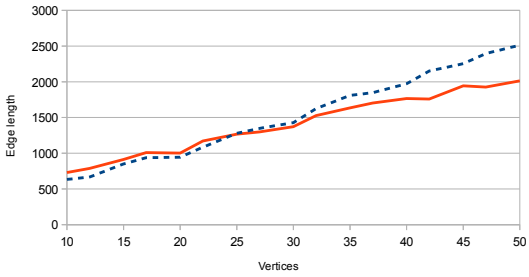
Similar observations can be made in the charts given in Figure 3.23, that show the performance of the node placers in terms of edge length with respect to the number of vertices. Again, the charts for the synthetic portless and port-based graphs show a clean curve, while the chart for the Ptolemy II model set shows a more eccentric curve.

As was already obvious in the averaged results, the linear segments node placer generally performs better in terms of edge length in the set of portless models. Both curves are approximately linear, with the curve for the new node placer rising quicker by a very small amount.

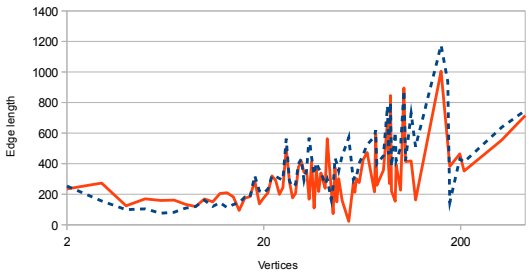
The chart for the port-based result shows where the Brandes Köpf node placer achieved its scores for the surprising result in the edge length metrics: It performs significantly better in larger graphs, with a vertex count of about twenty-five.



(a) The edge length metric in the set of port-less models.



(b) The edge length metric in the set of port-based models.



(c) The edge length metric in the set of Ptolemy II models. Note that the x-axis is scaled logarithmically, to improve the readability of the results for small graphs.

Figure 3.23. The edge length metric in the different model sets. Again, the dashed line represents the results of the linear segments node placer, the solid line the results of the Brandes Köpf node placer.

Generally the edge length curve of the Brandes Köpf node placer starts slightly higher than the curve of its competitor, but rises significantly slower, resulting in a better performance in larger graphs.

The chart of the results of the Ptolemy II model set for the edge length metric resemble the results of the edge bend metrics, but with the linear segments node placer scoring better here. Again, especially in diagrams that consist of about one hundred vertices, several diagrams exist in which the new node placer scores better than expected. This resembles the observations in the port-based graphs, in which the new node placer started to score better in larger diagrams.

All things considered, it can be concluded that the new node placer fulfils its purpose. The number of edge bends can be reduced significantly in many cases, resulting in cleaner diagrams, as could be seen in several examples in this chapter.

Furthermore, the trade-off to be made in terms of the increased diagram size, is not necessarily bad in all cases: as seen for portless and real world diagrams, the aspect ratio was improved by the new node placer, resulting in diagrams that are more suitable for display or printing.

Chapter 4

Label Placement

The second main chapter of this thesis deals with the problem of label placement in automatic layout and has a similar structure as the node placement chapter. The first section will define the problem of label placement, starting with very general placement tasks and refining them to tasks connected to graph drawing and modelling. The second section presents approaches and ideas to solve the placement problems given in the first section. The third chapter will describe the implementation of some of the presented approaches in the context of KIELER and KLay Layered. The last section will evaluate the label placements produced by the implementation.

4.1 Problem Statement

Labels are used to enhance a drawing, a graphic, or a model, where the information can not be expressed by the drawing alone. They have the task to clarify relations, semantics, and ideas. In many cases, the reasonable placement of the annotations is essential to the fulfilment of this task. For example, if labels are used to describe data which is propagated via a connection in a data flow diagram, the label has to be placed close to the connection, without being placed close to other connections or objects, so that the meaning of the label and its association with diagram objects is clear. An example for an annotated data flow diagram is given in Figure 4.1.

The problem of label placement itself is much older than computer science, since information exchange via annotated drawings can be found early in the history of mankind. A prominent example is cartography, represented in Figure 4.2 by an excerpt of the famous *Carta Marina*, a map of northern Europe, created around A.D. 1539 by Olaus Magnus. The author uses labels placed close to graphical objects to express their meaning. For example, the label **KIL** next to the drawing of a tower in the middle of the map expresses that the tower represents the city Kil, commonly known as the state capital of Schleswig-Holstein by its modern name, Kiel. As can be seen with the label **HOLSATHIA**, proximity alone may not suffice to clarify the

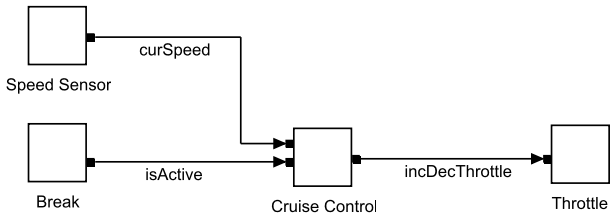


Figure 4.1. A simple data flow diagram, with the propagated values and component names annotated through labels.

Figure 4.2. An excerpt of the *Carta Marina*, showing parts of today's northern Germany and a few Danish isles.

meaning since it is placed close to several villages and lakes. By drawing the label in a larger font, the author expresses that **HOLSATHIA** is the name of the whole duchy containing the mentioned villages.

As could be seen with this example, the basic requirements of cartography and label placement in the graphical modelling domain are very similar. It can be expressed by two basic structures, the labels themselves and the graphical features to which the labels belong. The term feature covers all the objects that will be of relevance in both domains, cities, villages, and rivers or rather vertices, edges, and ports.

To develop this into a clearly defined problem one can solve, requirements of a good label placement have to be found. From the examples above, two essential requirements for any reasonable label placement can be derived:

- A label must not overlap another label. Label overlaps would decrease the readability of both labels dramatically and thus has to be avoided whenever possible.
- Every label has to be unambiguously associated with a graphical feature. While the first criterion is obvious and easy to check up on, this criterion is rather vague and may be difficult to evaluate. It can be influenced for example by proximity or distinguishing aspects like font, size, or color.

One can easily think of additional criteria, but the two given above are essential for the quality of any label placement result. In the field of cartography, Eduard Imhof and Pinhas Yoeli studied this subject thoroughly, and identified the given two criteria as essential as well [Yoe72, Imh75]. Since label placement in graphs is closely related to cartography, this can also be assumed for graph label placement.

This leads to a general definition of the basic label placement problem:

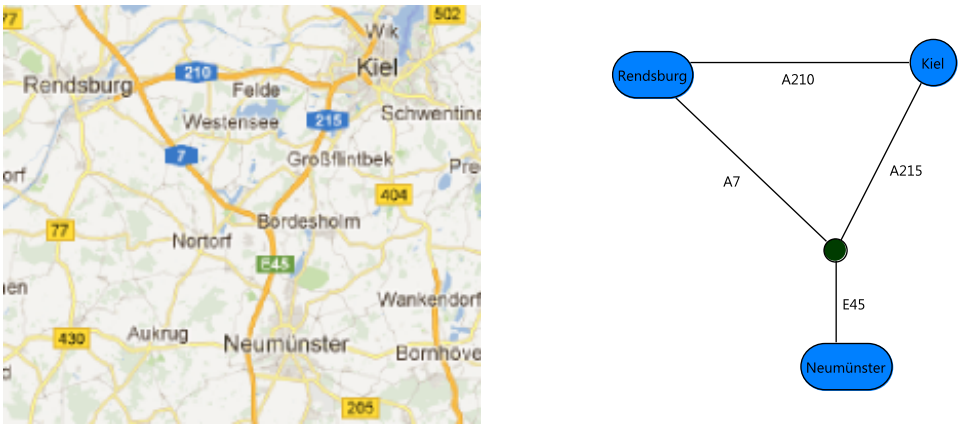
Definition 4.1. Given a finite set F of *graphical features* f, and a set Λ of possible label positions, a *label placement* has to be found which is represented by a total function $\lambda : F \to \Lambda$, meaning that λ is defined for every element of F.

In this definition, the criteria of good label placement are not taken into account yet. This is because often there is no clear distinction between a criterion's being satisfied or not. This leads to a function which tries to capture the extent to which a label placement satisfies the criteria: the *cost function*.

Definition 4.2. The function $COST_\lambda : \Lambda \to \mathbb{N}$ determines the cost of the placement of a label at a given position by taking the quality criteria into account.

The cost function dynamically changes with the already chosen candidate positons because overlaps may be introduced or avoided. It offers different possibilities for the evaluation of a placement. Breaking one of the given rules can be punished with higher costs, according to the importance of the criteria to the user. An overlap will always get a high penalty, but ambiguousness may be punished differently, e. g., if different fonts are used as an additional measure beside proximity.

With this way of evaluating a possible solution to the label placement problem, it can now be stated as an optimization problem:



(a) Map showing a part of eastern Schleswig-Holstein.[1]

(b) Graph representing a simplified version of the given map.

Figure 4.3. Equivalent structures in cartography and graph drawing.

Definition 4.3. For given set of features F and label positions Λ, a placement λ as defined in Definition 4.1 has to be found which minimizes the following function:

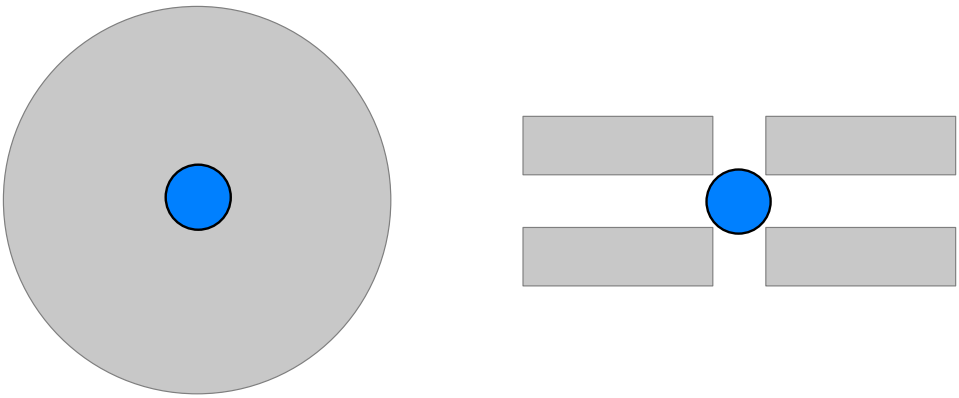
$$\sum_{f \in F} COST_\lambda(\lambda(f))$$

With this, a general definition of a labeling problem is found. This problem definition, without further extensions, is primarily applied in the field of cartography. Nevertheless, there also exist approaches in the field of graph drawing that use this general definition [KT03].

As one can easily imagine, the general problem definition can be extended for better specifying the label placement task and narrowing down the set of possible positions, resulting in a more manageable problem. Again, the closeness to cartography can help to identify basic structures that are shared by maps and graphs. As shown in Figure 4.3, one can express a map as a graph, by allowing a certain level of simplification. With this, the relationship between cartography labels and graph labels gets obvious, e. g., by mapping cities to vertices and roads to edges.

Vertex or node label placement and edge label placement are the two parts of the general label placement of a graph. Furthermore, the graph model was extended by ports in Section 2.1 that may also be labeled. There is no obvious

[1] `http://maps.google.com`



(a) An area for placing labels associated to a point.

(b) Candidate positions for placing labels associated to a point.

Figure 4.4. Possible reductions of the label position set in point feature label placement.

direct equivalent to ports in cartography. Nevertheless, ports are covered by the general definition of the label placement problem given above, because they can be regarded as graphical features placed very close to a vertex and an edge.

In the following, the labeling of these three basic structures will be investigated by giving a problem statement for each of them individually.

4.1.1 Node Label Placement

In cartography, a very common case is the labeling of single points on the map, when cities are simplified by being represented by a single point instead of their actual extend. Because of that, one sub problem of the general feature label placement problem is the point feature label placement problem [CMS95]. In there, a continuous region or a discrete number of positions around the point to be labeled are chosen as candidate positions for each point to restrict the candidate position space. An example for candidate areas and positions for a point feature label placement is given in Figure 4.4.

The introduction of candidate areas and positions extends the definition of the feature label placement problem from Definition 4.1 as follows:

Definition 4.4. For a given finite set $P \subseteq F$ of point features, a subset $\Lambda_p \subseteq \Lambda$ is defined for every $p \in P$. A label placement is then represented by the functions $\lambda_p : P \to \Lambda_p$.

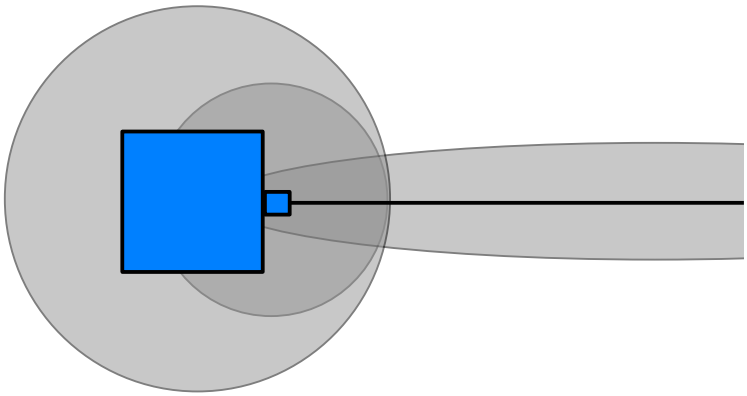


Figure 4.5. The placement areas of a port, a vertex, and an edge.

As seen before, vertices can be regarded as the counterparts of cities. Hence, the idea of point feature label placement can be applied to node label placement as well. A difference to the common idea of point feature label placement is that vertices may differ in size and can be very large. Additionally, a vertex may also be a rectangle with an odd aspect ratio. However, this does not prevent the basic idea of point feature label placement from being applied; it only changes the possible placement area or candidate positions.

Of course, the optimization problem as defined in Definition 4.3 can be applied to node label placement as well, by choosing a cost function suitable for the requirements posed to the label placement, most likely basing on the two essential rules of label placement given above. It is also possible to create a ranking of the candidate positions: in cartography, for example, the top right position is usually considered most desirable [Imh75]. When thinking about an approach later on, it might be helpful to keep in mind that point feature label placement, and with that node label placement, is NP-hard [FW91].

Another labeling problem which is closely related to point feature label placement is port label placement.

4.1.2 Port Label Placement

As cities, ports are usually drawn as point-like objects as well, e. g., small squares or triangles. Thus, one could easily infer that port label placement could also be solved using point feature label placement. The challenge, however, comes from the necessary closeness of ports to other features like vertices, edges, and other

ports. The areas where a vertex, its port, and a connected edge influence each other in terms of label placement is shown in Figure 4.5.

As one can clearly see, even this small example already poses a challenge to a pleasing placement of labels for all these features. When introducing more ports and edges on the given vertex, label overlaps will be unavoidable if the position and size of the features is fixed.

With ports and vertices, two of the three basic structures of a port-based graph covered, the third, edge label placement, which was already hinted at, will be investigated.

4.1.3 Edge Label Placement

At first sight, edges are the first graph structure which cannot simply be abstracted as a point and can instead be compared to roads and rivers in the cartography context. The major difference in most cases is the length and path of an edge in comparison to, e. g., a road. Especially in maps of a larger scale, long and curvy roads are the common case. In graph drawing, however, it is preferable to have short and straight edges, as was already discussed in Chapter 3. Of course, this results in a smaller placement area for edge labels. This can already be seen in Figure 4.3, even though the graph is drawn rather generously with regard to space usage.

Furthermore, roads usually only have one label. This is different in graph drawing; many modelling environments use a distinction of edge labels into three types:

- **Head Label**: On a directed edge, this label is placed near the target vertex.
- **Center Label**: This label is associated with the edge itself and is placed at its center.
- **Tail Label**: On a directed edge, this label is placed near the source vertex.

When the edge is undirected, the distinction between head and tail labels disappears. In the following, they will thus be called *end labels*.

With this, every edge has three different placement areas, one for each of the given label types. An example of an edge with its placement areas is given in Figure 4.6. Unfortunately, these areas have to be kept clear even if there is no label of the respective type present, to avoid any kind of ambiguity, such as the one seen in Figure 4.7.

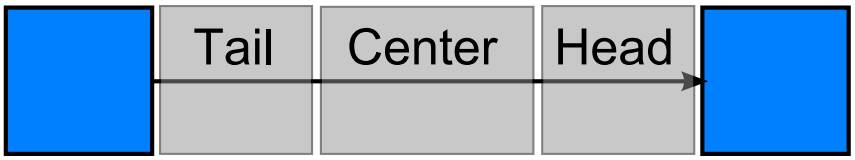


Figure 4.6. The placement areas of the different edge label types.

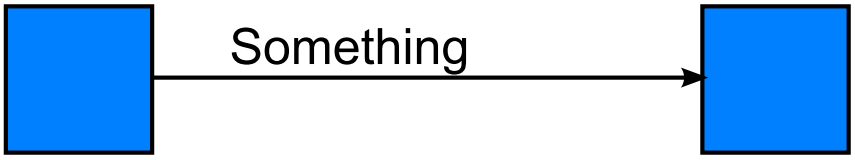


Figure 4.7. An edge label placement where a distinction between tail and center label is not possible.

Again, it is possible to find an edge label placement by using a point feature label placement approach. A point which lies on the edge, preferably in the center of the presented placement areas, can be chosen for each of the label types. But with this, the labels of each point on the edge will be placed individually.

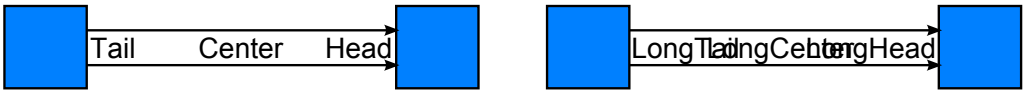
Especially in the case of edges, additional constraints may be applied to the placement, in an attempt to increase readability. A major decision when placing more than one label on the same edge is the choice of a side of the edge to place the label to. Switching sides on the same edge may lead to obfuscation and confusion. To get a grip on this, the Side Aware Edge Label Placement (SAELP) problem will be defined and discussed in the following.

4.1.4 Introducing Side Aware Edge Label Placement

When labels carry information or even semantics, e. g., in automaton graphs, it is essential to be able to unambiguously associate the right label with the right edge. When having several edges connected to one vertex, the decision on which side of the edge the label should be placed plays a key role. Figure 4.8 shows examples of label placements that lead to ambiguity, resulting in a nearly unusable diagram.

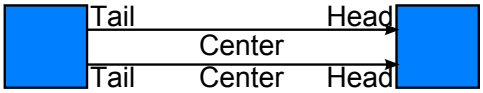
This shows that carefully choosing the placement side of an edge can be very important for a readable diagram. Furthermore, it is very favourable to have all labels associated with an edge placed on the same side of the edge.

With this in mind, the rules for a good label placement can be extended to the following:



(a) Labels which are placed between two edges, making it unclear which label belongs to which edge.

(b) The long labels result in unreadable annotations in this example.



(c) The labels of the upper edge are not on the same side, a possible source of confusion.

Figure 4.8. Ambiguous edge label placements due to poor choice of placement side.

- A label may not overlap another label.
- The association between labels and edges must be unambiguous.
- All labels associated with an edge have to be on the same side of the edge. This may be above and below, or left and right, depending on the orientation of the edge. Conflicts may arise when more than one edge is connected to a vertex.

An optimization problem for Side Aware Edge Label Placement (SAELP) can now be defined by choosing a cost function depending on the rules given above. It will then be used to find a label placement which minimizes the term given in Definition 4.3.

Before thinking about possible approaches to this problem, the complexity of the SAELP will be analyzed.

4.1.5 On the complexity of SAELP

As seen before, the general problem of label placement has been investigated most thoroughly in the context of cartography. The general point feature label placement has been found to be an NP-complete problem [FW91]. Although point feature label placement is applicable to many cases, placement problems for simpler cases in the graph context have also been proven to be NP-complete. Three groups investigated node label placement, discovering that despite several simplifications the problem is NP-complete [FW91, MS91, KH88].

Regarding the edge label placement problem, Kakoulis and Tollis made two discoveries while investigating its complexity [KT01]. Their first discovery was that there is no easy transformation from edge label placement to the node or point feature label placement problem. The second discovery then was that edge label placement is also NP-complete. To prove that, Kakoulis and Tollis defined several simplifications of the problem and proved each of them to be NP-complete. The general approach was to reduce a first simplification to the 3-SAT problem, as posed by Craig Tovey [Tov84].

In this context, the presented SAELP problem can also be regarded as a simplification of the general edge label placement. It will be proved here, by reduction on one of the simplified problems by Kakoulis and Tollis, that the SAELP problem is NP-complete as well.

The first step to take before starting a proof is to pose the SAELP problem as a decision problem rather than an optimization problem, since NP-completeness can only be shown in terms of decision problems. For doing so, the three rules of SAELP (no overlaps, unambiguousness and side awareness) are used in a more strict cost function:

Definition 4.5. Given a label assignment λ and a graphical feature $f \in F$, the cost function is defined as follows:

$$COST_\lambda(\lambda(f)) = \begin{cases} 0 & \text{if placement satisfies rules} \\ 1 & \text{else} \end{cases}$$

With this, the SAELP decision problem is represented by the following definition:

Definition 4.6. Find a label assignment $\lambda : F \to \Lambda$ which satisfies the given rules by having this term resulting in 0:

$$\sum_{f \in F} COST_\lambda(\lambda(f))$$

Additionally to this basic decisive criterion, several constraints to the problem are posed, to narrow down the possible placement positions and adjust the problem to the requirements of the label placement goals of this thesis.

- Every edge has six discrete, non-overlapping positions at which the labels can be placed. Thus, every edge label type has two candidate positions, as can be seen in Figure 4.9.
- Orthogonal edge routing will be used. This means, labels can be placed alongside their edges without interfering with, e. g., odd other edges.

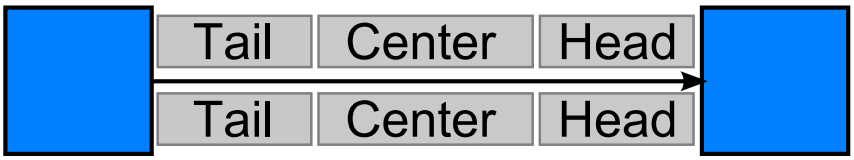


Figure 4.9. Six discrete positions for the three labels of an edge.

▻ Labels are of the same size and require one third of the length of an edge, resulting in enough space for all three label types on an edge.

The approaches to the SAELP problem presented later on fit these constraints even though they might seem overly restrictive at first.

Theorem 4.7. *The SAELP problem, with the given constraints, is NP-complete.*

Proof. The first step will show that SAELP $\in$ NP, and the second step will then show that SAELP is NP-hard.

(i) To show that SAELP is in NP, it has to be shown that a possible solution can be checked for its correctness in polynomial time.

To do so, an arbitrary label assignment is chosen for each edge label to be placed. After that, every label has to be checked for overlaps with other labels or ambiguity with respect to other labels or other graphical features. Here, ambiguity can be checked by requiring the corresponding label of a feature to be the closest label, and by requiring the feature to be the feature closest to the label. Then, every label has to be compared with other labels of the edge, to check whether the labels are on the same side.

Obviously, the first and second requirement can be checked in polynomial time, the third requirement in linear time.

(ii) To show that SAELP is NP-hard, is has to be shown that SAELP can be reduced to any problem that is in NP, and that any problem in NP can be reduced to SAELP, each in polynomial time. This can be done by showing these reductions for any particular NP-hard problem since this reduction relationship is transitive.

As mentioned before, Kakoulis and Tollis showed that the general edge label placement problem is NP-hard. In their proof, they introduced several simplifications of the general problem, who themselves proved to be NP-hard

as well. The problem relevant to this proof is the Discrete Admissible Edge Label Placement (DAELP) problem which assumes discrete, non-overlapping positions for a single label on an edge. This positions happen to be chosen as "above the edge" or "below the edge".

The NP-hardness of SAELP can now be shown by reducing SAELP to DAELP. This can be done by merging the three labels of a SAELP into a single one. These tasks can clearly be performed in polynomial time.

It has to be shown now that transformed solutions to the problems are still valid:

Theorem 4.8. *Given an instance of the SAELP problem, the transformation given above constructs an instance of the DAELP problem with zero cost if and only if the SAELP has zero cost.*

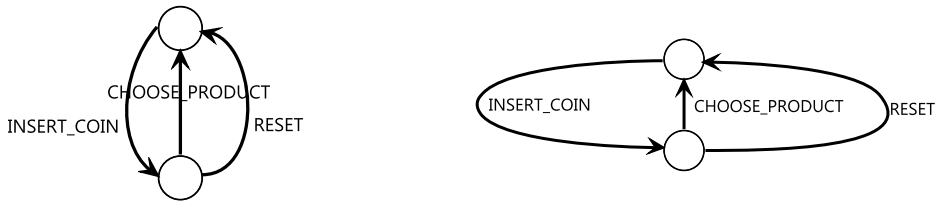
Proof. "⇒": Assume an instance of the SAELP problem with zero cost. It follows that the labels of the graph were assigned without violating the side awareness requirement and that the two basic rules of non-overlapping labels and unambiguous placement were adhered to. With this, the three label positions for each edge given by SAELP can merged to yield a zero cost DAELP.

"⇐": Assume an instance of the DAELP problem with zero cost. This means that the decision on which side of the edge the label has to be placed for a zero cost assignment has been made. Since the requirements of SAELP constrained edge labels to have a length of one third of the edge and DAELP reserves the whole side of an edge clear, the single DAELP label can now be split into the three SAELP labels. Since the DAELP was already free of overlaps and unambiguous, and since the created SAELP positions are on the same side, a zero cost SAELP is yielded. □

This shows that SAELP is in NP and that it can be reduced to an NP-hard problem; thus, SAELP is NP-complete.

□

This concludes the statement of label placement problems covered by this thesis. The following section will now present ideas and approaches to label placement.



(a) Moderately long labels make it hard to distinguish which label belongs to which edge.

(b) The same state machine, but the layout left enough space for the labels.

Figure 4.10. Readability can be improved by explicitly reserving space for labels.

4.2 Approach

Unlike node placement, there will not be a single approach or algorithm to all the label placement problems given above (node label placement, port label placement, and edge label placement). While this would be possible by transforming all of these into point feature label placement problems, specialized algorithms will yield better results for the different label placement problems. The general and essential idea of candidate positions however is applied as well. Additionally, the approaches presented here will also use the advantages of a fully accessible and changeable layout process, to improve the quality of difficult label placements by explicitly reserving space for labels.

4.2.1 Spicing up layout with label placement

As seen before, the core problem of label placement is finding enough space to place all labels while fulfilling at least the two basic rules of an overlap-free and unambiguous placement. In some cases, for example when a single label is exceptionally long, it may even be impossible to find a placement which satisfies the placement rules without modifying the layout of the graph. Especially when labels carry semantic meaning, e. g., transition labels in a state chart or automaton, the labels tend to become rather large. An unfortunate placement may render the diagram unreadable and thus unusable, as the example of Figure 4.10a shows. There, the placement algorithm did not have much choice, since it was not given any influence on the positions of vertices or on the routing of edges.

So, when implementing a label placement algorithm in a post-processing approach after the general layout of a graph is determined, the possibilities are limited. However, when integrating label placement into an existing layout algorithm such as KLay Layered, a larger variety of solutions is possible. Remember that the structure of KLay Layered supports intermediate processors between the main phases, giving the opportunity to alter the graph or the layout result in a way which allows to create exactly the space that is needed for the labels. With this, the example of Figure 4.10a may result in an altered, more readable layout as shown in Figure 4.10b.

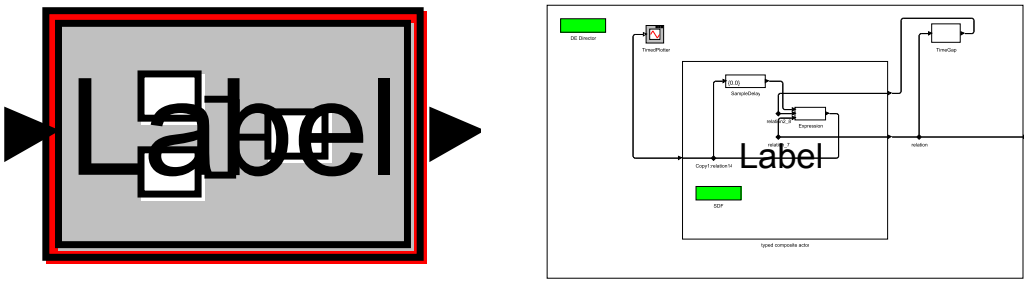
This approach of altering the layout in different ways to grant each label the space it needs will be applied to the label placement problems presented in Section 4.1. For some problems, multiple approaches will be presented and discussed. Depending on the concrete use case, one approach might be better than others, or might not be applicable at all.

4.2.2 Node Labels and Node Margins

The problem of node label placement is the graph label placement problem which has the closest relation to a pure and general point feature label placement. Candidate positions are chosen around the vertex or point and the position with the highest score is picked for the final placement, taking overlaps with other labels into account.

In the graphs laid out by KLay Layered, vertices are not regarded as points, but also have a size. This has no further influence on the concept of candidate positions, but will play a role when thinking about placing the label inside of the vertex. Whether a label can be placed inside of a vertex depends on two conditions. The first condition is that a vertex must not contain anything that could overlap with its label. As seen in Figure 4.11, this can be the case when a vertex uses an image to display its meaning, or when a vertex in a nested graph contains another graph.

Nevertheless, in many cases it is possible and desirable to place labels inside of a vertex, e. g., in the context of state machines. To do this, the second constraint has to be kept in mind: is the layout algorithm allowed to change the size of a vertex, and if so, under which conditions? If the vertex size is fixed, only labels that are smaller than the vertex itself can be placed inside of it. Other requirements to the size may be that the aspect ratio of the vertex has to stay the same, that a certain size must not be exceeded, or that port positions are constrained. The latter will also be important when thinking about port and edge labels.



(a) Label placed inside of a vertex, overwriting its graphic content.

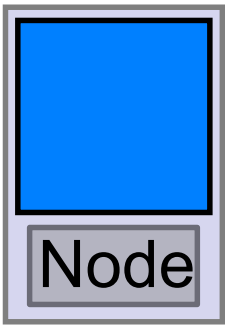
(b) Label placed inside of a compound vertex, overwriting the nested graph.

Figure 4.11. Inside-node label placement is not applicable to all types of vertices.

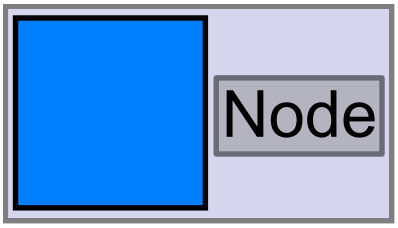
To avoid checking and dealing with these constraints on a placement inside of a vertex, one can choose to place labels outside of a vertex. In this case, an approach similar to the point feature label placement approach has to be chosen, resulting in choosing candidate positions around the vertex and decide on one for the label placement.

One important difference to the normal point feature placement is that the placement algorithm is not forced to work with fixed vertex positions, and thus without any influence on the amount of space reserved for a label. It rather is able, due to the structure of KLay Layered, to decide on a candidate position early on in the layout process and reserve the space for the label during the layout process. To do that, an intermediate processor adds a certain *margin* to the outline of a vertex before node placement takes place, to reserve the space needed to place the label at the chosen candidate position. An example for candidate positions and resulting margins can be seen in Figure 4.12.

With this, a label placement satisfying the rules of good placement can only be prevented by further elements that are part of the vertex, namely vertex content or connected ports and edges. In the worst case, all four sides of the vertex are occupied by edges and the interior is used by, for example, a nested graph. In this case, it depends on the constraints whether a satisfying label placement is yet possible. When the vertex size is allowed to be altered, the vertex can be drawn large enough to place a label well away from a blocking edge on either side, as can be seen in Figure 4.13. If the constrains allow no change in size, a less favourable candidate position has to be picked, due to a lack of alternatives.



(a) Candidate position and margins if the label is placed below the vertex.



(b) Candidate position and margins if the label is placed on the right side of the vertex.

Figure 4.12. Examples for node label placements and the resulting margins.

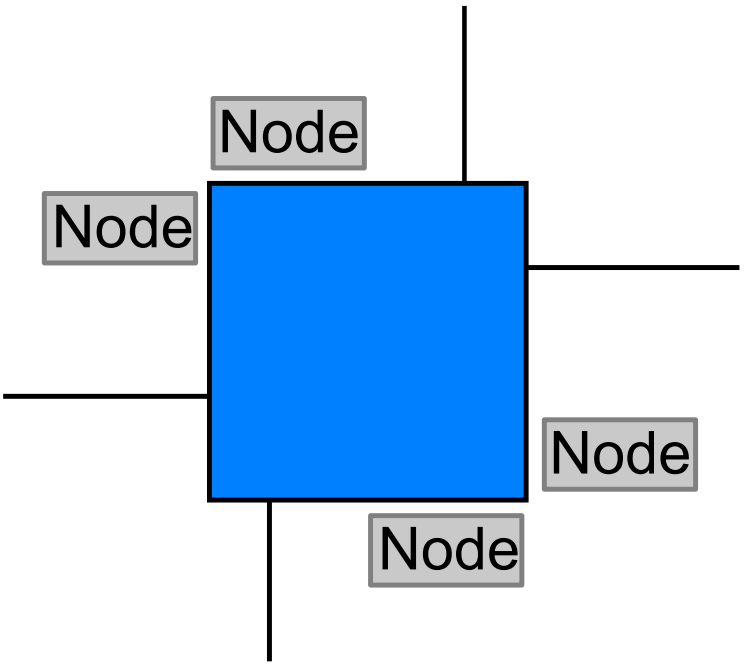


Figure 4.13. Possible placements when all sides of the vertex are blocked by edges.

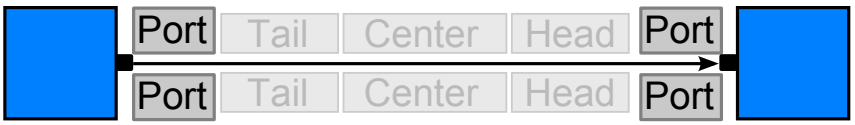


Figure 4.14. Stretching the edge to introduce enough space for the port label.

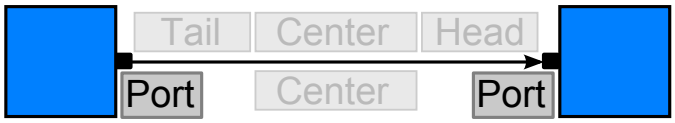


Figure 4.15. Reserving edge label candidate positions for port labels.

4.2.3 Port Labels

When placing port labels, the difficulty comes from the unfortunate location of ports. Since ports are connection points for edges on a vertex, they necessarily lie close to their vertex and at least one edge. As seen before, the placement area for port labels lies well inside the placement areas for edges and labels, leaving almost no placement area belonging to the port label alone.

Because of that, a successful port label placement algorithm has to negotiate with the placement algorithms of the other label types to make sure that it does not introduce overlaps or ambiguousness. Depending on the situation in the respective graph, several approaches to port label placement are possible.

Again, an approach that works in most situations is to simply create the space required to place the port label. This can be done by increasing the margin of its vertex by the amount needed for the port label, effectively creating enough space for port and edge labels, as can be seen in Figure 4.14. Nevertheless, this approach should be regarded as a fall-back solution to be used only if none of the following approaches can be applied, because several disadvantages exist: if only one port label or edge label is present, it may not be clear which if the two kinds of labels it is. Additionally, this approach will end up creating very long edges if each port has a label and all three types of edge labels are present.

A similar approach is to place the port label close to the port on one side of the edge, and placing the edge labels a little away on the other side of the edge, as shown in Figure 4.15. While this solution saves a little more space, other disadvantages still exist. It may introduce ambiguousness when only one label is present and one is not sure whether it belongs to the port or to the edge. When vertices have more than one port on a given side, there might also not be enough

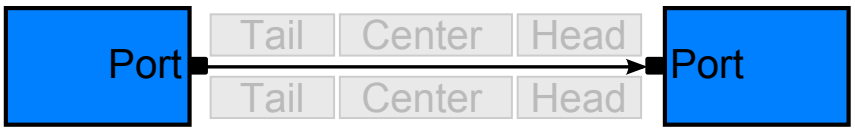


Figure 4.16. Placing port labels inside of vertices.

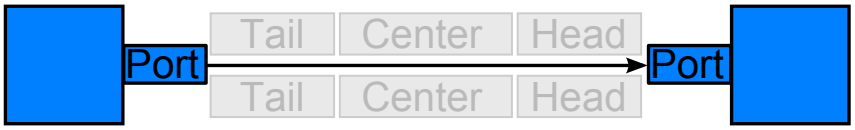


Figure 4.17. Port label placement inside of the port itself.

space for this approach, if the vertex cannot be resized to accommodate the labels.

A different approach is similar to one already seen in the context of node labels, and is limited by similar constraints. In this approach, the space inside of a vertex is used to place the port labels. With that, the placement space available for edge labels is not influenced and a major source of ambiguity is ruled out. An example for this is shown in Figure 4.16. Although this approach is preferable, in many cases it can not be applied. Apart from the reasons already mentioned in the similar node label placement approach (content, size restrictions), the size problems are more severe in this case. If the vertex is not to be resized, it is very unlikely that enough space is available for all port and node labels.

A final approach requires the ports themselves to be resizeable and filled with a color on which the labels in their normal font and color can be read. Additionally, ports would be limited to rectangular shapes. Then, the ports can be resized such that every port can contain its own label, as shown in Figure 4.17. This approach ensures the placement to be unambiguous and rules out any influence to the other placement types. Unfortunately, the resized ports start to look odd when a certain size is reached. Therefore, this approach should only be used with small port labels.

For the application of a port label placement approach, only one or two of the ideas presented here should be chosen, to prevent confusion among users. For the general case, the placement inside the vertex should be the method of choice because it results in the most clear placements when vertices are resizeable. The edge stretching approach can be used as a fall-back solution if the constraints of the graph prevent an inside placement. It is also possible to leave the choice between these two approaches to the user.

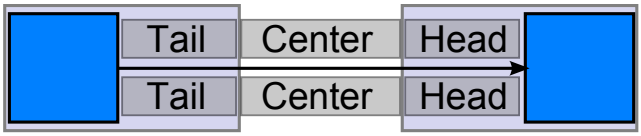


Figure 4.18. End label space is kept clear via vertex margins.

4.2.4 Edge Labels

The final label type left to think about is edge labels. Here, three different edge label types that were presented in subsection 4.1.3 have to be considered. Furthermore, a feasible approach to the SAELP problem is in the scope of this thesis as well. Since it is NP-complete, several heuristics will be presented instead of giving an optimal solution with respect to a given cost function.

Again, the focus of the approaches will lie on creating placement space instead of being content with the space left after the graph was laid out. Because of that, the approach for the end labels of an edge will rely on a concept already presented. To ensure that there is enough space for each end label of an edge, the margins of the respective vertices are increased by the size of the labels, resulting in a free area around the vertex and edge into which the label can be comfortably placed. An example for such a margin is given in Figure 4.18. A problem could still occur if there is more than one edge connected to a vertex, because end labels might overlap in an unwise placement. A solution to this problem will be included in the SAELP approaches.

As for center labels, this approach is not applicable. This is because an edge might be routed around other vertices and edges, resulting in a simple horizontal margin being useless. Of course, it is impossible to increase the vertex margins in horizontal and vertical dimension until the label has enough space: with this, large chunks of white space would be introduced around every vertex and edge with a center label.

Instead, a method already used quite extensively in layered layout is applied: dummy vertices. Eiglsperger et al. proposed a similar approach to label placement for the orthogonal layout of UML class diagrams [EKS03].

The basic idea of this approach is to split every edge which has an associated center label by inserting a newly created dummy vertex that represents the label. The size of the dummy depends on the size of the label, to make sure that enough space is reserved for the label. Figure 4.19 shows the introduction of dummy vertices for a center label. Note that the dimensions of the dummy vertex depend

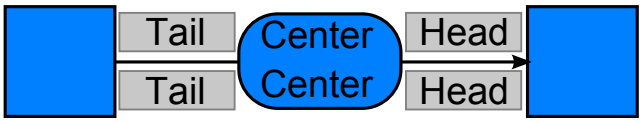
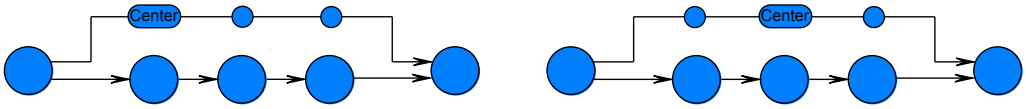


Figure 4.19. A dummy vertex is introduced to reserve space for a center label with enough space to be able to choose a placement side later on.



(a) Label dummy vertex position after layering.

(b) Label dummy vertex position after swap.

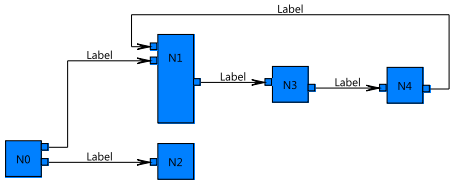
Figure 4.20. Swapping label dummy vertices to centermost position after layering.

on whether a side for the label was already found. If so, the dummy can occupy exactly that space below or above the edge. If not, the dummy has to extend to both sides to reserve enough space for both possible placements.

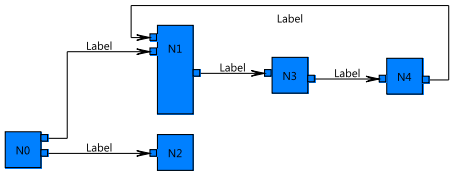
A problem with this approach in a layered layout algorithm is the choice of at which point in the algorithm the dummy vertices should be inserted. Clearly it should take place before the layering, otherwise a layerless dummy or a complicated adjustment of the given layering is the result. Unfortunately, when having long edges, this usually results in having the label dummy vertex at the beginning or the end of a long edge, which is not desirable for a center label. This can be solved by letting the layering algorithm do its work and trying to swap the label dummy with the centermost long edge dummy afterwards, as can be seen in Figure 4.20. Unfortunately, it can still happen that the center label is placed a little away from the center of the edge if the dummy count of an edge is even. Then, the swap takes place with one of the two center dummies.

With this, end and center labels are guaranteed to have enough space for a placement. The last choice to be made now is the placement side, addressed by the SAELP problem presented in subsection 4.1.4. One can think of several heuristics that make a more or less educated decision on the placement side. Examples for the three approaches are given in Figure 4.21.

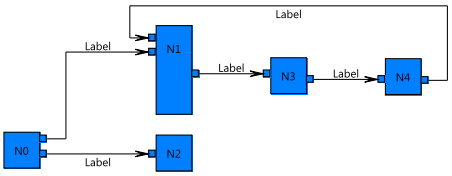
A first and simple heuristic is found by deciding on a static side choice for all labels. Because of the edges being routed orthogonally, the edge segment on which the label will be placed can always be drawn horizontally and long enough, such that a label has enough space to fit close to the edge. In the case of northern or



(a) Labels are always placed above edges.



(b) Labels are placed above edges with respect to direction.



(c) Labels are placed depending on edge patterns in a greedily manner.

Figure 4.21. Examples for the three SAELP approaches.

southern ports, the labels have to be placed above or below each other, or the vertex has to be resized to fit the labels. Furthermore, when every label is placed on the same side, e. g., above or below the edge, there will be no overlaps, again, because all of the placement spaces are horizontal, except for the case where there is not enough space between the edges and the vertex is not resizeable.

Another approach connects label placement with the semantics of a diagram by placing the labels on a certain side depending on an edge's direction. For example, the labels can be placed above an edge if the edge points from left to right and below it if the edge points from right to left. Contrary to the first approach, this approach does not exclude label overlaps that are not related to edge spacing issues. As shown in Figure 4.22, if edges with different directions meet on the same vertex side, overlaps can occur. To avoid that, ports of a vertex would have to be sorted by incoming and outgoing edges, which might or might not be allowed, depending on the diagram to be laid out.

The final approach presented in this thesis works in a greedy manner. Several possibilities of edges connecting to a vertex are distinguished that dictate a certain side choice, e. g., a vertex with two connected edges. The preferable placement would be above the edge for the upper edge and below the edge for the lower

Figure 4.22. Overlap in a direction-based label side choice.

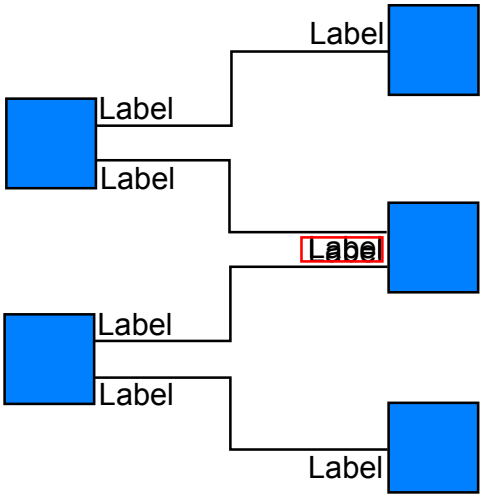


Figure 4.23. Overlaps in a greedy approach that does not take other placement choices into account.

edge. If this was applied to all vertices without further constraints, overlaps as shown in Figure 4.23 could occur. Because of that, the placement algorithm marks all connected vertices according to the choice of sides if they were not marked beforehand. Further vertices that are connected to a marked vertex must place the labels on the same side to avoid overlaps. If a new marker differs from an already present one, a conflict might occur, as can be seen in Figure 4.25. In this case, a compromise has to be made by placing the respective end labels on the side indicated by their associated vertex. This approach is greedy because the graph is traversed vertex by vertex and the side markers are set according to the earlier vertex's structure.

An example for that is given in Figure 4.24. The algorithm starts with N0, recognizing the pattern of two outgoing edges. The connected vertices N1 and N2 are checked for markers, but none are found since they have not been visited before. The labels of the two edges are then marked for UP or DOWN placement, N1 is marked with UP, and N2 is marked with DOWN. The algorithm continues with N1. Since there is no special pattern, and N3 is not marked, it defaults to UP, marking the labels of the edge and N3 with UP. Continuing with N2, the algorithm finds the marking UP on the connected vertex N3. Thus, the labels of the edge are marked with UP as well.

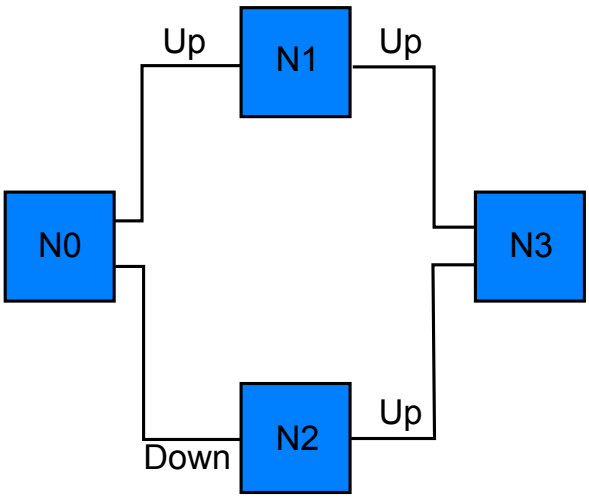


Figure 4.24. An example graph showing the general process of the greedy heuristic.

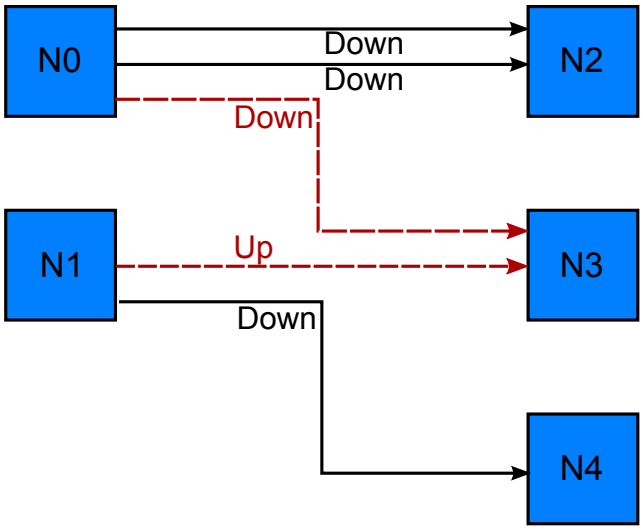


Figure 4.25. The vertex N3 was marked for a label placement below the edges. This conflicts with the desired placement above the edge by vertex N1.

A possible improvement of this approach may be an algorithm that considers several placement choices and selects the placement with the smallest number of conflicts.

This concludes the proposal of approaches to the different graph label placement problems. The following section will deal with implementations of selected approaches in the context of KLay Layered.

4.3 Implementation

As described in subsection 2.4.1, KLay Layered consists of phases and intermediate processors. As one can easily see, label placement is not part of the main phases. When thinking about integrating label placement with this algorithm, a choice has to be made between introducing a new phase or using intermediate processors for the label placement. This choice depends greatly on the type of label placement algorithm. If it is a post-processing, the introduction of a sixth phase in KLay Layered for the label placement would be acceptable. However, the approaches presented in this thesis instead need to influence the layout at several points. For that reason, the label placement approaches have to be implemented as intermediate processors that are carefully integrated into KLay Layered at the right places. Therefore, the following section will present the intermediate processors used to implement the given approaches, along with information about when in the algorithm they are executed and the necessary pre- and postconditions that apply to each processor.

4.3.1 Node size and margins

When labels are chosen to be placed inside of a vertex, the vertex has to be resized to fit the label. In the KIELER tool, the diagram editors perform this rescaling automatically when entering a new label. Nevertheless, it would be preferable for the layout algorithm to be able to perform the required resizing when necessary. Additionally, in the case of an outside placement with all four sides blocked by edges, the vertex might be resized to fit the label as well. Unfortunately, the current structure of the KLay Layered algorithm does not allow intermediate processors to resize vertices. A task for future work would be to change that. To be prepared for that case, the intermediate processor for this resizing will already be included with the `NodeLabelSizeAdjuster`, given in Table 4.1.

For the placement of labels outside of a vertex, the approach using margins to reserve space for the labels is applied. As a part of his diploma thesis, Schulze

Table 4.1. The `NodeLabelSizeAdjuster` intermediate processor.

Preconditions	None.
Postconditions	Vertices are resized, such that labels fit inside or beside the vertices. The labels store information about the reserved space, to avoid having to calculate this again later on.
Slot	Before phase 1 (cycle removal).
Dependencies	None.

Table 4.2. The `NodeMarginCalculator` intermediate processor [Sch11].

Preconditions	The graph is layered. Port positions are fixed.
Postconditions	Node margins are set to reserve space for labels. The labels store information about the reserved space, to avoid having to calculate this again later on.
Slot	Before phase 4 (node placement).
Dependencies	`PortPositionProcessor`.

created a `NodeMarginCalculator`, as shown in Table 4.2, which can be used for this [Sch11].

After these two intermediate processors, a final processor will be needed for the actual placement of vertex labels once that is made part of the layout algorithm. The `NodeLabelPlacer`, given in Table 4.3, depends on information about the space reserved by the previous intermediate processors.

4.3.2 Port label space

As noted before, port labels share their possible placement space with vertex and edge labels. The two approaches for port label placement that are implemented here are to place port labels inside of their vertices and to place them outside, moving the possible end labels further towards the middle of the edge.

The intermediate processors are responsible for these two placement areas were already presented with the `NodeLabelSizeAdjuster` and the `NodeMarginCalculator`. Thus, the necessary measures for reserving port label space are included in these two processors, rather than creating new ones.

The processors have to be modified to add the space needed for placing the port labels, be it inside or outside of a vertex, to the already adjusted vertex size or node margins. Furthermore, the actual placement of the port labels has to be

Table 4.3. The `NodeLabelPlacer` intermediate processor.

Preconditions	Vertex sizes or margins reserve space for labels. Labels are annotated to hint at the chosen placement.
Postconditions	Labels are placed inside of the reserved space.
Slot	After phase 5 (edge routing).
Dependencies	None.

included in the processors for the actual placement of node or edge labels after phase 5. In the case of an inside placement, the label can be placed beside the port inside of the vertex. On northern or southern ports, enough horizontal space has to be reserved. In the case of an outside placement, the port label has to be placed along with the other edge labels, to avoid having port labels and edge labels on different sides of an edge

4.3.3 Simplifying label placement

With the placement structure for node and port labels in place, the last missing label type are edge labels. According to the approach presented before, the placement is simplified by introducing a discrete set of candidate positions. For every edge label, the placement decides between two positions, above or below the edge, denoted as UP and DOWN. This translates to left and right in the rare case of vertical edges. To keep the side decision algorithms as clean as possible, intermediate processors will be introduced for end and center labels that calculate the concrete position for a label depending on the given side choice.

4.3.4 SAELP heuristics

The placement algorithms need information about which side of the edge the labels should be placed on. The side is decided by a heuristic that approximates a solution to the SAELP optimization problem. Three possible heuristics were given in subsection 4.2.4. The implementation of these heuristics is included in the `LabelSideSelector` intermediate processor that is described by Table 4.4.

The processor implements the strategy pattern from the famous design patterns by Gamma et al. [GHJV95]. Depending on the choice of the user, offered to him as a layout option as presented in subsection 2.4.1, a heuristic is picked from the set of heuristics implemented in the processor. The choice is between ALWAYS_UP, ALWAYS_DOWN, DIRECTION_UP, DIRECTION_DOWN, and SMART.

Table 4.4. The `LabelSideSelector` intermediate processor.

Preconditions	The graph is properly layered.
Postconditions	A choice of placement side is made for every edge. The choice is documented in every concerned label.
Slot	Before phase 3 (crossing minimization).
Dependencies	None.

The implementation of the first four strategies is straightforward. For the completely static side selection, every label is simply annotated with UP or DOWN respectively. In the direction dependant strategy, the direction of the edges is regarded before making a choice. For example, if the user chose DIRECTION_UP, labels on edges from left to right, or in-layer edges, are marked with UP, whilst labels on edges from right to left are marked DOWN.

If the choice was SMART, the greedy approach tries to find patterns of desirable side choices, or defaults to UP if no pattern matches, and marks the edges respectively. If the target vertex of an outgoing edge is already marked with a placement side, that side is taken for the outgoing edges connected to current vertex as well. In the current implementation, the only pattern is to place labels on a vertex with two outgoing edges above and below the upper and lower edge.

4.3.5 End label placement

For end label placement, space has again to be reserved by an earlier processor, namely the `NodeMarginCalculator`. It is modified to take the space into account that is necessary for placing the end labels. The side of the edge the labels should be placed on was decided by the `LabelSideSelector`. With this space reserved, the `EndLabelProcessor` can translate the chosen side and the reserved space into a final placement for the end label. It has to make sure that the label placement does not interfere with a possible port label that could have been placed outside of the vertex, alongside the edge. The `EndLabelProcessor` can be seen in Table 4.5.

In case of end labels on vertices with more than one vertical edge connected to the same vertex side, the already placed labels' heights have to be used as an offset to prevent label overlaps. One could also prevent overlaps between labels and edges by modifying the `NodeLabelSizeAdjuster` such that every vertex is resized until all edge labels can be placed between the edges without touching another edge. However, that would result in very large vertices even with small labels since

Table 4.5. The `EndLabelProcessor` intermediate processor.

Preconditions	Vertex sizes and margins reserve space for labels. Labels are annotated with the chosen placement. Space needed for port labels is known.
Postconditions	Labels are placed inside of the reserved space.
Slot	After phase 5 (edge routing).
Dependencies	None.

Table 4.6. The `LabelDummyInserter` intermediate processor.

Preconditions	The vertices have not been assigned to layers.
Postconditions	Center labels are represented by dummy vertices.
Slot	Before phase 2 (layer assignment).
Dependencies	None.

the size of all labels and some additional spacing will add up to the final vertex size.

4.3.6 Center label processing

This leaves center labels as the last missing label type. As seen in the approach to center label placement given in subsection 4.2.4, the space for center labels is reserved via dummy vertices. Thus, the task of a first intermediate processor is to introduce dummy vertices for every center label. The `LabelDummyInserter`, given in Table 4.6, splits every edge annotated with a center label into two edges with a dummy vertex of the size of the label in between them. This takes place even before the layering, to avoid complex modifications of an already calculated layering.

After the graph is layered, the `LongEdgeSplitter`, which was already a part of KLay Layered, transforms the layered graph into a properly layered graph (edges do not span more than one layer, see Chapter 2 for more details) by introducing dummy vertices to split long edges. Now, the center label dummies have to be swapped with a centermost long edge dummy, to have the label as close to the middle of the edge as possible in the final placement. This task is performed by the `LabelDummySwitcher` given in Table 4.7.

An additional benefit of the `LabelDummySwitcher` is that the center label dummy vertex can now be resized according to the choice of side given by the `LabelSide-Selector` since the height of the label dummy only influences the layout after

Table 4.7. The `LabelDummySwitcher` intermediate processor.

Preconditions	The graph is properly layered. Center label dummy vertices have been inserted.
Postconditions	Center label dummies are the centermost dummies in a long edge.
Slot	Before phase 3 (crossing reduction).
Dependencies	`LongEdgeSplitter`. `LabelSideSelector`.

Table 4.8. The `LabelDummyRemover` intermediate processor.

Preconditions	Center label dummy vertices have been inserted.
Postconditions	Center labels are placed at the position of the dummy vertices. The edges split by a center dummy are joined again.
Slot	After phase 5 (edge routing).
Dependencies	None.

phase 4. This simplifies the task of the final processor for center labels, the `LabelDummyRemover`, as given in Table 4.8. This intermediate processor removes the label dummy vertices and places the actual label accordingly. The label is placed at the exact position of the label dummy that was decided on in the node placement and edge routing phase.

This concludes the implementation section, since all selected label placement mechanisms were included in KLay Layered via intermediate processors. The following section will investigate results of the new placement methods and reason about possible advantages, disadvantages, or problems that remain to be investigated.

4.4 Evaluation

Compared to the evaluation methods of node placement presented in Section 3.4, the methods of evaluation for the label placement results will be quite different. This is because there is no actual candidate for a fair and meaningful comparison via aesthetics criteria: no approach with similar conditions, a label placement completely included in the layout algorithm, is known to the author. Comparison with a post-processing approach, e. g., an approach by Kakoulis and Tollis [KT03], is not suitable, since the challenges of both placement methods are utterly different. When using aesthetics criteria based on the basic label placement rules, no overlaps, ambiguity and a close proximity to the labeled elements, no fair comparison is

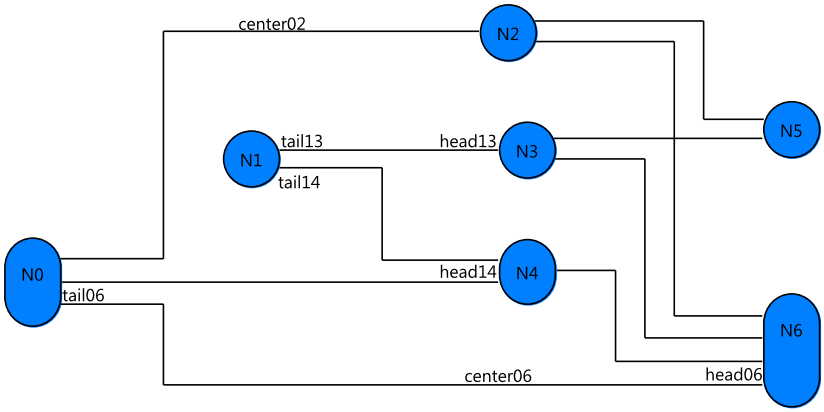


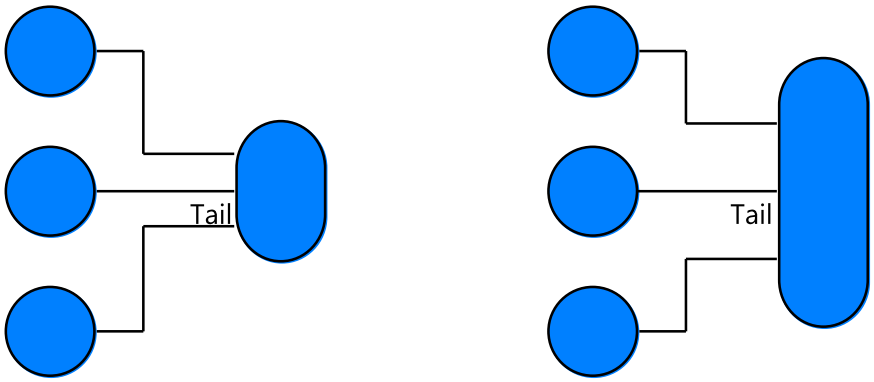
Figure 4.26. Edge label placement in a portless graph.

possible. The approach by Kakoulis and Tollis struggles with the given drawing to place the labels as well as possible while the approach presented in this thesis simply alters the graph drawing such that enough space is available for each label. Furthermore, even if a comparison of the two algorithms were meaningful there are practical problems: since the algorithm by Kakoulis and Tollis is patented it cannot simply be implemented in the context of KLay Layered.

Therefore, the evaluation in this chapter will rely on examples of the new placement method, to point out advantages or disadvantages of this method or and to discover problems that might still exist. Note that the resizing of vertices to fit the respective labels in between edges or inside of the vertex was done manually for these examples, since KLay Layered does not allow the resizing of vertices by the layout algorithm at the time of writing. As seen in the previous section, the algorithms for this resizing are already present and ready to be included as soon as KLay Layered allows the resizing.

4.4.1 Example results

The first example, given in Figure 4.26, shows the result of edge label placement in a portless graph with the greedy heuristic with side awareness activated. One can see that the side awareness is complied for all edges and that, of course, no overlaps exist. A possible problem can be seen with the vertices N0 and N6. The labels of a connected edge forced a resizing of the vertices N0 and N6, making them twice



(a) Ambiguity introduced by too few labels and too little space between the three edges.

(b) A possible solution is keeping edges well away from each other.

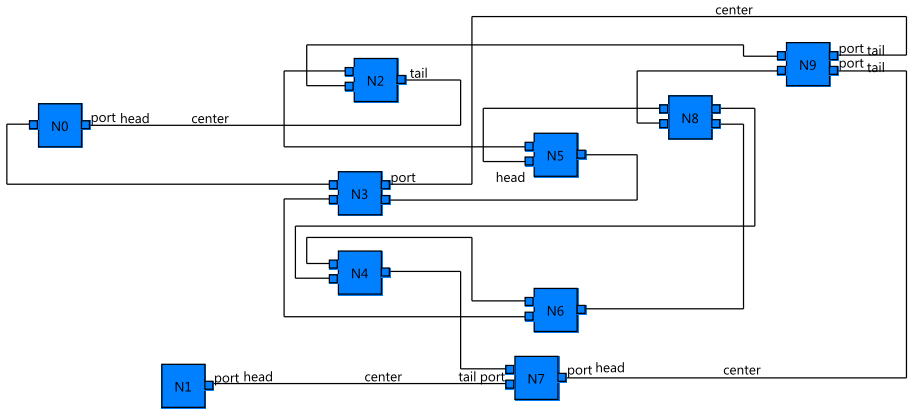
Figure 4.27. Problems of ambiguity on vertices with more than two edges on the same side.

the size of some other vertices in the graph. This effect will get even stronger with more edges connected on the same side, but is an unavoidable trade-off if space for the labels is created.

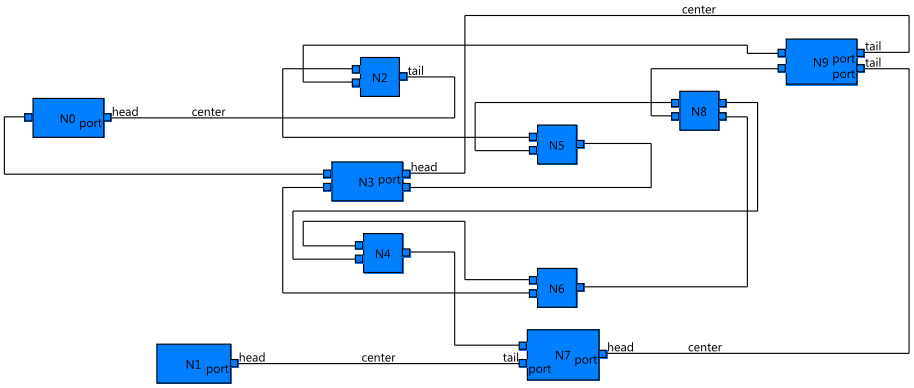
A case of ambiguity would occur with the label "head14" if the side awareness was not ensured. Without side awareness, a reader could not decide to which edge the label belongs. With side awareness the reader can check the other labels of the edge and compare them to the side of the questionable label.

Nevertheless, a general problem can occur if not at least two labels of an edge are present. As can also be seen in Figure 4.27a, ambiguity is introduced if only a few edges are labeled. In case of labels of outer edges this could be solved by using a different heuristic which forces this label to stay on the side of the edge that does not introduce obfuscation. This is not possible for inner edges. There, two possible solutions exist: the usage of a static choice of sides makes the user know that the labels of an edge are e. g., always above it. The other solution requires more space: the vertex is resized such that there is enough space between the edges for a clean placement, as seen in Figure 4.27b.

Larger examples of a port-based graph with port label placement inside and outside of a vertex are shown in Figure 4.28. The overall placement of labels in the examples is acceptable, however, the possible disadvantage of reserving space



(a) Port labels are placed outside of vertices.



(b) Port labels are placed inside of vertices.

Figure 4.28. The same, port-based graph with the two different port label placements.

Table 4.9. Properties of the evaluation model sets, denoting the minimum and maximum count of each value, followed by an average given in brackets.

	Portless	Port-based
Diagrams	100	100
Vertices	5–100 (52.5)	5–100 (52.5)
Vertex degree	3–14 (3.04)	3–12 (2.97)
Edges	30–150 (79.8)	30–150 (77.76)

for the labels can be seen here as well: the vertices that contain the rather short port label "port" are resized to twice the length of vertices without port labels. This would be even worse with longer port labels.

4.4.2 Evaluation of space usage

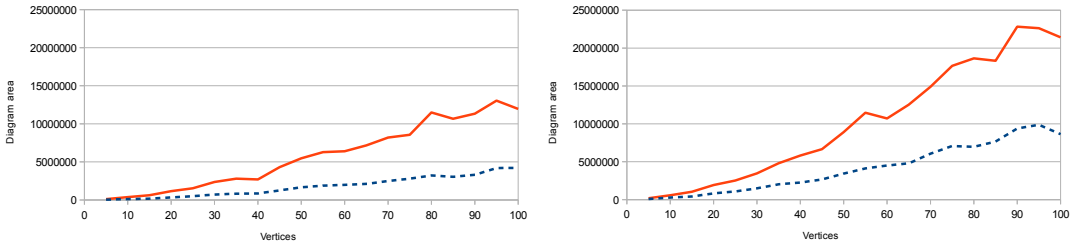
As seen in the examples given above, the major trade-off of reserving space for label placement is, not surprisingly, a notable increase in the overall size of the graph. Because of that, a final evaluation will be done to quantify the effect of additional space usage when using the presented label placement approaches. In this evaluation, the focus will lie on two of the metrics introduced in subsection 3.4.1: the drawing area of the graph and the aspect ratio. Again, the analysis framework implemented by Martin Rieß can be used to analyse the results [Rie10].

For performing this analysis, a set of test graphs will be created, similar to the evaluation as performed in Section 3.4. As there exist only few labels in the Ptolemy II Model set, which are vertex labels for the most part, it is not suitable for this evaluation. However, sets of random graphs can be used again for this evaluation. As before, a set of portless and a set of port-based diagrams will be used. Edges will be fully labeled, with labels "Head", "Center", and "Tail" at the respective positions. Both sets will consist of graphs with a vertex count between five and one hundred, with one or two outgoing edges per vertex, chosen randomly. The port-based graphs may also have northern or southern ports. The characteristics of the two graph sets can be seen in Table 4.9. The relevant layout options can be seen in Table 4.10. The evaluation takes place by laying out both sets of graphs, once with label placement activated and once without label placement.

In terms of drawing area, it is expected to have a significant rise in the used area, due to the introduced dummy vertices for the center labels and the larger node margins for the end labels.

Table 4.10. The layout options chosen for the two evaluation layouts.

Option	Value
Border spacing	20
Edge spacing	0.5
Spacing	20
Label Side	smart / none
Node Placement	Brandes Köpf
Edge Routing	Orthogonal



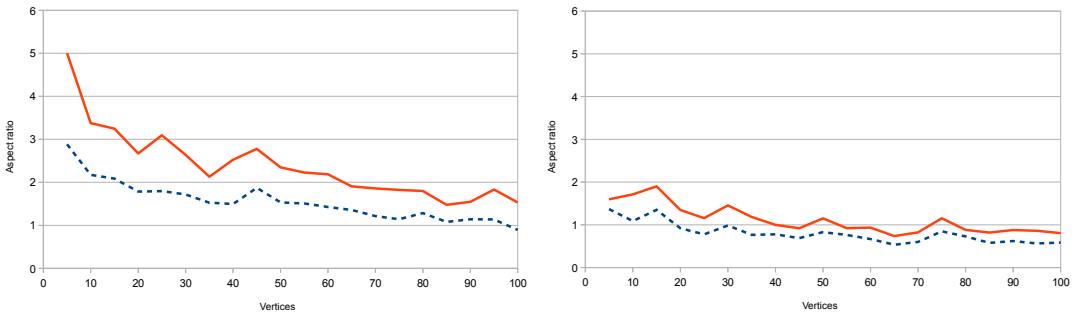
(a) Diagram area usage in the set of portless models.

(b) Diagram area usage in the set of port-based models.

Figure 4.29. The diagram area metric in the different model sets. The dashed line represents the results without label placement, the solid line the results with label placement. The unit of the y axis values is square pixels.

The aspect ratio is also expected to rise because the text of labels, and with that the reserved space, always grows from left to right. With that, the graph size will experience a significant growth in the horizontal direction. However, since the Brandes Köpf node placer, presented in Chapter 3, will be used for evaluation, the larger horizontal spread from the labels might balance the exceptionally large vertical spread created by the node placer in port-based diagrams.

The results from the evaluation of the overall diagram area are given in Figure 4.29. The expectation of a significant increase of the diagram area is met in both model sets, with the diagram size of models from the port-based set being larger in general. Regardless, the results show that reserving space for labels takes a great amount of space. In diagrams with a vertex count of twenty, the space usage is doubled, while the diagrams with one hundred vertices take are three times as large.



(a) Aspect ratio in the set of portless models.

(b) Aspect ratio in the set of port-based models.

Figure 4.30. Aspect ratio with and without label placement in the different model sets. The dashed line represents the results without label placement, the solid line the results with label placement.

The results of the aspect ratio metric evaluation are shown in Figure 4.30. Again, the results coincide with the expectations given above. It is flamboyant that the aspect ratio is especially high when label placement is applied to small portless graphs. This is simply explained, though, because the horizontal spread increases strongly with every labeled edge, while the vertical spread only rises when the layers grow, which happens in larger graphs or in port-based graphs that introduce more dummy vertices because of northern, southern, or inverted ports. In the long run, a layout without labels has a smaller aspect ratio than a layout with label placement.

A different behaviour can be found for port-based graphs. There, the vertical growth introduced by the Brandes Köpf node placer causes the aspect ratio to get very small in graphs with twenty vertices or more. As one considers either an aspect ratio of 1 (for square drawing) or an aspect ratio between 1.3 and 1.7 (for common paper or screen formats) desirable, the additional horizontal growth introduced by label placement actually balances the aspect ratio, resulting in an overall more pleasing diagram format.

This concludes the evaluation of the new label placement and thus the second main chapter of this thesis. The evaluation showed that the new label placement approaches can create feasible placement results, but still lack optimization, especially with respect to space usage.

Chapter 5

Conclusion

Now, after two topics in the domain of automatic layout or, more specifically, in a layered layout approach were thoroughly investigated, it is the right moment to take a step back and look at the work that was presented in this thesis. In this conclusion, it will be investigated from several points of view. For a start, the content of this thesis will be summed up, and the goals posed in Chapter 1 will be investigated in terms of whether they were achieved or not. After that, a look at the loose ends left by this thesis to be picked up by promising young scientists will be taken, presenting tasks and ideas for future work.

5.1 Summary

In this thesis, the existing layered layout algorithm KLay Layered was extended by several new features, to improve the layout results especially with respect to edge bends and to extend the algorithm by giving it the ability to deal with diagram annotations and labels.

In the first part, a new algorithm for the node placement phase of KLay Layered, the determination of vertical coordinates for the vertices, was presented, implemented, and improved. An approach by Ulrik Brandes and Boris Köpf was presented and explained, by extending the algorithms and explanations from the paper by more detailed explanations and algorithms that were omitted in the paper, probably because of space limitations.

The presented approach was extended by several new features of which some are essential for applying this approach in a real world environment. The algorithm is now able to deal with vertices of any size, a very important feature in real world diagram editors. Additionally, the arrangement of vertices that have the same y coordinate due to being inside of the same block was modified, such that every edge inside a block is drawn straightly. With this, the algorithm creates feasible results for port-based graphs as well. Furthermore, several mechanisms for compacting the node placement were introduced, like giving edges less space

than vertices or moving blocks as close to each other as possible. A best effort mechanism for drawing certain edges straightly was introduced as well.

The approach was improved by allowing it to also choose a node placement from the layout candidates calculated during the algorithm as a result, and not only the final, balanced result.

Complex structures that could be laid out by KLay Layered using the old node placement algorithm were also integrated into this new approach, to keep the abilities of the layout algorithm. For example, support for northern and southern ports, in-layer edges, and hierarchical vertices was included.

The evaluation showed that the main goal, the reduction of edge bends, was achieved by the new node placer. However, a trade-off had to be made in terms of diagram area, because the new straight edges require more drawing space, especially in the vertical direction.

While investigating label placement in graphs or diagrams, a new variant of the label placement problem was discovered with the Side Aware Edge Label Placement proven to be NP-complete.

The general approach to the presented label placement problems in this thesis was different to the post-processing approaches often applied to label placement. Here, the label placement was fully integrated into the layout algorithm by employing several mechanisms that explicitly reserve space for the respective labels.

Because these mechanisms have to be applied throughout the whole KLay Layered algorithm, the intermediate processor architecture of KLay Layered was leveraged to cleanly integrate the required algorithms.

The example results of this new integrated label placement were promising, showing a readable label placement for the different kinds of labels. However, the evaluation of the space usage of this approach showed that the space requirements rose significantly when additional space was saved for the labels. This shows that the work on integrated label placement is not finished with this thesis.

5.2 Future Work

It is a commonly accepted fact that scientific work is never truly finished. This is also the case for the topics of this thesis, so the following section will present ideas for future work on the two topics.

5.2.1 Node Placement

The evaluation of the new node placer showed that the space usage of the result is significantly higher than when using linear segments node placer. While this is unavoidable to a certain degree since straight edges require more space than edges bent around vertices, it is quite possible that methods exist that could be applied to get a more compact result. For example, the spacing around northern or southern ports is quite generous at the moment, to make sure that edge decorators such as arrowheads have enough space. This could be optimized, e. g., by checking for the presence of decorators.

The possibility of drawing a given set of edges straightly is already present as a best effort mechanism. It could be better to have a mechanism which enforces the straight drawing of selected edges. To achieve that, the algorithm would have to be restructured to be able to form blocks for such edges outside of the normal block creation routine. Nevertheless, each of these possible approaches will have its limits since, for example, two outgoing edges of a vertex that are connected to different target vertices in the next layer will always result in one of the edges being bent. Otherwise, the target vertices would overlap.

5.2.2 Label Placement

As with node placement, the general disadvantage of the new label placement approach discovered in the evaluation are the space requirements. As the approach of integrated label placement in the layered layout approach is new, there is no work on optimizing the used space. Especially the new dummy vertices for center labels require a lot of horizontal space. This might be reduced by several measures.

An approach to reduce the space usage of long center labels might be label management. There, one area of research is to investigate possibilities of introducing line breaks into labels to use otherwise wasted vertical space instead of horizontal space. Another, more general solution would be to reduce the spacing of label dummy vertices, similar to the edge spacing factor presented in the node placement chapter.

As for the SAELP problem, different heuristics besides the already presented ones may be investigated. For example, the transformation into an already existing optimization problem might be investigated, as the problem might translate well into linear equations or a graph coloring problem.

Last but not least, a restructuring of KLay Layered has to take place, to allow the algorithms to change the size of vertices. Once this is done, the new mechanism has to be integrated into the already present intermediate processors that resize vertices to fit the respective labels.

Bibliography

[BJL01] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. A Fast Layout Algorithm for k-Level Graphs. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 86–89. Springer Berlin / Heidelberg, 2001.

[BK02] Ulrik Brandes and Boris Köpf. Fast and Simple Horizontal Coordinate Assignment. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 33–36. Springer Berlin / Heidelberg, 2002.

[CGJ+07] Markus Chimani, Carsten Gutwenger, Michael Jünger, Karsten Klein, Petra Mutzel, and Michael Schulz. The Open Graph Drawing Framework. Poster at the 15th International Symposium on Graph Drawing (GD07), 2007.

[Cla10] Ole Claußen. Implementing an Algorithm for Orthogonal Graph Layout. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.

[CMS95] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Trans. Graph.*, 14(3):203–232, July 1995.

[CMT01] R. Castelló, R. Mili, and I. Tollis. An Algorithmic Framework for Visualizing Statecharts. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 43–44. Springer Berlin / Heidelberg, 2001.

[CRR10] John Carstens, Claas Anders Rathje, and Ulf Rüegg. Splines and their Use within an Edge Routing Process. Seminar proceedings, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2010.

[DETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[Döh10] Philipp Döhring. Algorithmen zur Layerzuweisung. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. *LNCS*, 2265:594–597, 2002.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming Heterogeneity—The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[EKS03] Markus Eiglsperger, Michael Kaufmann, and Martin Siebenhaller. A topology-shape-metrics approach for the automatic layout of UML class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03, pages 189–ff, New York, NY, USA, 2003. ACM.

[ES90] Peter Eades and Kozo Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–437, 1990.

[Fuh11] Hauke Fuhrmann. *On the Pragmatics of Graphical Modeling*. Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.

[Fuh12] Insa Fuhrmann. Layout of Compound Graphs. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2012.

[FvH10] Hauke Fuhrmann and Reinhard von Hanxleden. Taming Graphical Modeling. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 196–210. Springer Berlin / Heidelberg, 2010.

[FW91] Michael Formann and Frank Wagner. A packing problem with applications to lettering of maps. In *Proceedings of the seventh annual symposium on Computational geometry*, SCG '91, pages 281–288, New York, NY, USA, 1991. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, New York, 1979.

[GJ83] Michael R. Garey and David S. Johnson. Crossing Number is NP-Complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[GKN02] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, February 2002.

[GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *Software Engineering*, 19(3):214–230, 1993.

[Imh75] Eduard Imhof. Positioning names on maps. *American Cartographer*, (2):128–144, 1975.

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[KH88] T. Kato and H.Imai. The NP-completeness of the character placement problem of 2 or 3 degrees of freedom. *Record of Joint Conference of Electrical and Electronic Engineers*, pages 11–18, 1988.

[KT01] Konstantinos G. Kakoulis and Ioannis G. Tollis. On the complexity of the Edge Label Placement problem. *Computational Geometry*, 18(1):1 – 17, 2001.

[KT03] Konstantinos G. Kakoulis and Ioannis G. Tollis. A Unified Approach to Automatic Label Placement. *International Journal of Computational Geometry & Applications*, 13(01):23–59, 2003.

[Kut10] Christian Kutschmar. Planarisierung von Hypergraphen. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.

[LM87] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[MS91] Joe Marks and Stuart Shieber. The Computational Complexity of Cartographic Label Placement. Technical report, Harvard University, 1991.

[Pur02] Helen C. Purchase. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages & Computing*, 13(5):501 – 516, 2002.

[Rie10] Martin Rieß. A Graph Editor for Algorithm Engineering. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.

[San96] G. Sander. A fast heuristic for hierarchical Manhattan layout. In *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 447–458. Springer Berlin / Heidelberg, 1996.

[San04] Georg Sander. Layout of Directed Hypergraphs with Orthogonal Hyperedges. In *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, volume 2912 of *LNCS*, pages 381–386. Springer, 2004.

[Sch11] Christoph Daniel Schulze. Optimizing Automatic Layout for Data Flow Diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2011.

[Spö09] Miro Spönemann. On the Automatic Layout of Data Flow Diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.

[Tov84] Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85 – 89, 1984.

[WMP+05] Pak Chung Wong, P. Mackey, K. Perrine, J. Eagan, H. Foote, and J. Thomas. Dynamic visualization of graphs with extended labels. In *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pages 73 – 80, oct. 2005.

[Yoe72] Pinhas Yoeli. The Logic of Automated Map Lettering. *Cartographic Journal, The*, 9(2), 1972.