Improved Vertical Segment Routing for Sugiyama Layouts

Thies Weber

Bachelor Thesis
March 2019

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

Advised by
Dipl.-inf. Christoph Daniel Schulze

Selbstständigkeitserklärung

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

Kiel,

Abstract

Automatic layout algorithms represent a vital enhancement for taking care to maintain good styles for constantly growing graph structures. For instance, the ELK Layered algorithm achieves this by arranging all nodes in subsequent layers. This way it provides in the end a notion of direction which the majority of edges will stick to. For routing the edges it may use the orthogonal edge routing, a style drawing edges only with horizontal and vertical segments. The orthogonal edge routing assumes each edge to consist of either one horizontal segment or otherwise of exactly three—one vertical and two horizontal ones. Such a vertical segment bridges the complete gap in height, which must be overcome for an edge to reach its target, at once. This thesis will prove that in specific cases of equal port positions this style of routing may lead edges to partially overlap and thus to be ambiguous. It will provide a solution consisting of two additional segments in order to enhance one of the two overlapping edges' flexibility for bridging the height-gap. The edge routing determines a layout's width whereas more segments mean more required space for edges. Hence, an important step of this solution involves a careful consideration of alternative placements for the additional segments to not stretch the final layout more than necessary. This holds especially for the new horizontal segment. Also it tries to not produce additional crossings in the process. For this matter the thesis provides a number of theoretical discussions about additional crossings and layout settings leading to them. It turns out that additional crossings can only show up if there is a hyperedge involved in the overlap in the first place. But still the new algorithm is able to handle those without causing additional crossings in most cases.

Contents

List of Figures

Acronyms

EES edge-to-edge spacing

ELK Eclipse Layout Kernel

ELKT ELK Text Format

FS Feedback Set

FVS Feedback Vertex Set

JSON JavaScript Object Notation

KIELER Kiel Integrated Environment for Layout Eclipse RichClient

RS routing spaces

RTSYS Real-Time and Embedded Systems Group

SCCharts Sequentially Constructed Statecharts

Chapter 1

Introduction

In theory a graph is a conceptional mathematical structure bringing things in relation. It consists of two sets of vital structures: the *vertices* (or *nodes*) represent elements of the relation and the *edges* establish relations between vertices. But the human brain usually can understand things better when seeing it, instead of descriptive mathematical structures. This is why there is a necessity for graphical representations. Those drawn graphs are quite helpful to understand and explain complex relations within their theoretical and practical setting. But for this matter it is vital for the visualization to be comprehensible, which is the more difficult the bigger the graph is. Consider for example the drawing process of a mind map, which is a common procedure of developing a graph directly during its visualization. By doing so the graph may become more and more complex and thus unstructured, since there is usually a limited area to draw on. Plus there could be new relations added later conflicted by the initially chosen structure. An exemplary scenario is shown in Figure 1.1. There is a mind map describing a number of animals divided by land-living and marine-living ones. After beginning to draw, creating the first relations, and noting the first animals the drawing person noticed that a turtle

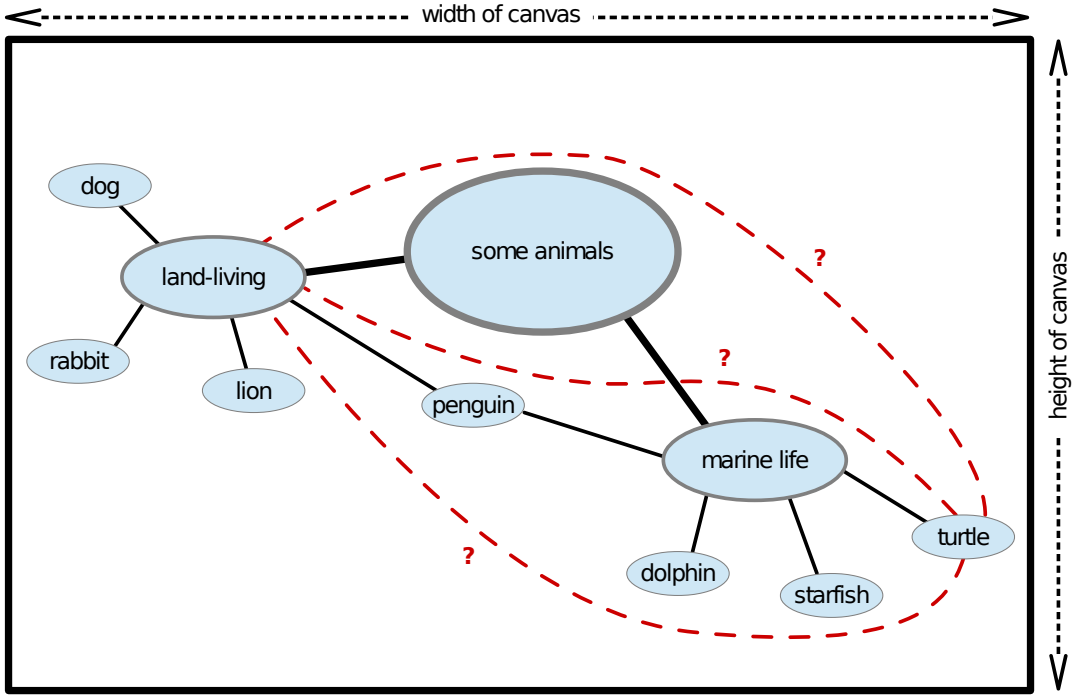
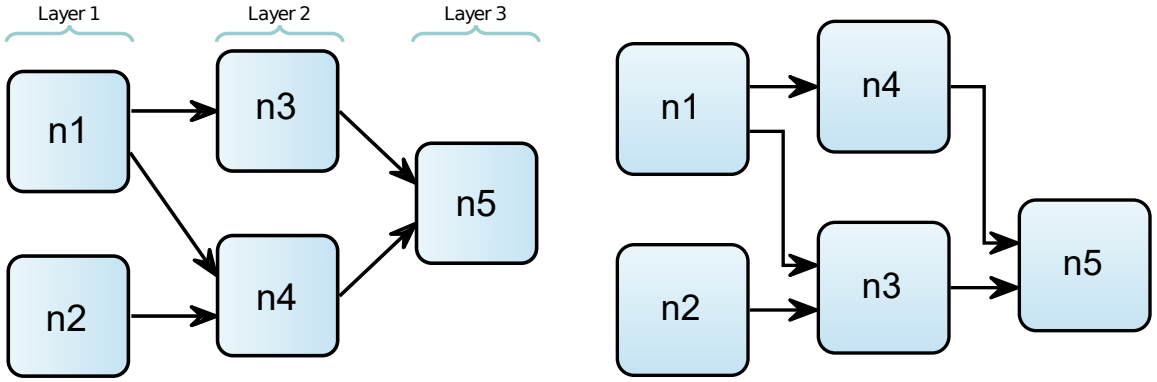


Figure 1.1. An example of a mind map becoming unstructured during the process of creation. Potential new edges are highlighted as dashed lines.



(a) Layed out with *polyline* edge routing. The layers are explicitely highlighted.

(b) Layed out with *orthogonal* edge routing.

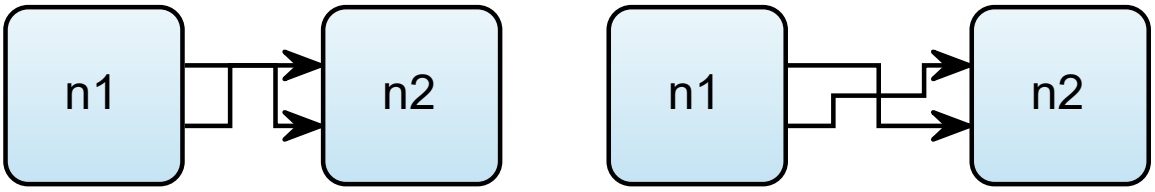
Figure 1.2. A graph layed out with the *layered* algorithm, but with two different edge routings.

could live on land as well as in the oceans. In order to correct the structure it is necessary to add a new relation between *turtle* and the *land-living* node. But since the layout is already set the new edge now crosses other edges or nodes or go a long way around the other nodes, which is not longer comprehensible or at least bad style. The best solution would be to rearrange the node *turtle* and place it below the node *penguin*, but this would be additional work and would be quite complicated if the graph is for example drawn on paper. To avoid this problem one could create this mind map directly on a computer, so it is possible to simply rearrange the structure. But there is still the problem of how to arrange it and of course the necessity of doing this regularly in order to have the expanding graph be comprehensive at all times. This is where *automatic layout* may come in handy.

Automatic layout relies on an algorithm to place nodes and edges, which takes quite often as much time as the development of the graph itself. This is why Kiel University's Real-Time and Embedded Systems Group (RTSYS) developed the framework Eclipse Layout Kernel (ELK)[1] as an infrastructure to compute the layout of graph structures. But what should such a layout look like? Since a graph as a conceptional structure is so diversified in its representation, there could be numerous additional requirements for a good layout relying on the actual relation a graph is standing for. For example the nodes of a family tree should be drawn in the chronological order of generations or birth dates, while it is necessary for a subway infrastructure's map to somewhat resemble geographical positions of the stations. This is why there cannot be one ultimate automatic graph layout algorithm and thus why ELK provides a number of layout algorithms as well as the opportunity of implementing and integrating a very own one. This thesis will discuss a problem introduced in Section 1.1, dealing with one of those provided layout algorithms: the *layered* algorithm.

The layered algorithm is based on the *Sugiyama approach* [STT81], which is also called the *layered approach*. It relies on a partitioning of its nodes in *layers* which results in a layout with a notion of direction often used to show specific data flows between components, which I will refer hereinafter as *routing direction*. A layer is a subset of the graph's nodes and arranges them on the same level. An example is given by Figure 1.2a. There are three layers—the first one contains the nodes n1 and n2, the second n3 and n4 and the last only n5. In order to make descriptions easier I assume the routing direction to always be from left to right. This means the layers are ordered horizontally and each layer arranges its nodes in a vertical order. Of course ELK's actual implementation has to work for all four

[1] https://www.eclipse.org/elk/



(a) Problem: an overlapping occurs. **(b)** Solution: Additional segments and space.

Figure 1.3. The problem of overlapping horizontal segments and a proposed solution.

directions, though.

The layered approach runs through five phases every one of them dealing with a specific problem. The first four phases are amongst other things about assigning the graph's nodes to aforementioned layers and determining their placement. The last phase is called *edge routing* and lays out edges between those layers. This phase is what this work will be about. Since the algorithm already ran through the other phases once the edges get routed, the following assumptions hold: First, every node has a layer and its node ordering is fixed and second, the edge routing only has to care about the space between adjacent layers. A more precise description of the layered approach and its phases is given in Section 2.2.

1.1 Orthogonal Edge Routing

The layered algorithm runs through five phases and thus through five algorithms solving the specific problems. It is important to note that every one of these algorithms is replaceable by another one which solves the problem as well, but maybe in a different way and with different results. For example in Figure 1.2a the so-called *polyline* edge routing was used. However, the specific problem I am going to discuss will be about *orthogonal* edge routing shown in Figure 1.2b. The orthogonal character of this approach leads every edge to contain only vertical and horizontal *segments*. Further it lets every regular edge, which cannot be drawn as straight horizontal edge, have exactly three segments in total. It turns out that this feature can become problematic in some cases. This problem is shown in Figure 1.3a. There are two horizontal segments overlapping each other. I already mentioned that every graph may have special requirements for its layout to be good, but one characteristic is equal for almost every graph: it must not have overlapping edges. This is because they would be impossible to distinguish and thus to comprehend. Avoiding this means adding more segments as in Figure 1.3b. During this thesis I will explain the orthogonal edge router's functionality and search for reasons why this problem may occur. Since the orthogonal edge routing assumes every edge to have not more than one vertical segment, finding a solution may cause additional problems in the process.

1.2 Related Work

During my work I will cover a number of topics such as orthogonal edge routing and graph visualization in general. As for the former, Georg Sander [San04] introduced an approach of routing orthogonal hyperedges, which is an edge with multiple sources or targets, within a layered graph layout. It arranges vertical segments with a specific internal graph structure, the *segment crossing graph*. This is an idea the orthogonal edge router benefits from by using the approach for its own structure, which will be explained in more detail during Section 2.3. Furthermore Sander provides an intuitive

style of layout for drawing hyperedges. He proposes to draw a hyperedge for the longest possible path as single edge and to perform the actual forking only right before reaching its sources or targets. This idea inspired my handling with hyperedges when adding new segments which is described in Section 3.3.2.

Visualization of graphs is a very complex topic due to the lack of generalization. Graphs are conceptional structures which can be utilized for almost everything. Each use case comes with its own requirements for a good layout. However, there are actually a number of common criteria applicable for almost every graph. Characteristics like symmetry or the avoidance of overlaps and crossings are generally considered as indicator for good layouts. Battista et al. [BET+98] propose a number of notions a good layout should adhere to, as well as approaches and algorithms for automatic layout algorithms used as examples to explain the layered approach in Section 2.2. While overlaps are trivial cases of making a graph's components incomprehensible, which I will show on examples during my work, there is still the question how severe crossings and symmetry influence a layout's clarity. This is a matter Purchase et al. [PFJ95] dealt with by performing empirical studies to validate the hypotheses of increasing a graph's understandability due to reduction of crossings and providing local symmetry. While they indeed provide evidence for graphs benefiting from crossing minimization, the advantages of symmetry as a general way for enhancing clarity remain unproven.

1.3 Outline

After introducing the problem this work will begin with giving the basic information about the layered approach and the orthogonal edge routing in Chapter 2. The reasons why the proposed problem's solution may not be as easy to realize as it seems will be discussed in Chapter 3. Further I will discuss in this chapter why the problem shows up in the first place and prepare the actual process of implementation by analyzing a number of problems I might encounter. The actual process of implementation will be proposed during Chapter 4, its evaluation in Chapter 5. Finally in Chapter 6 I will discuss a number of things, which could be realized or further improved in the future.

Chapter 2

Basic Knowledge

This chapter is meant to provide the theoretical basic knowledge needed to understand this thesis. For this matter it starts with introducing a number of terms and definitions in order to understand graphs as theoretical but also as visual structures. Additionally I present technologies I will use during this thesis. Afterwards I explain the Sugiyama approach I already hinted at in Chapter 1. Finally the orthogonal edge router is explained, the algorithm this thesis relies on.

2.1 Description of Terms and Technologies

Before discussing the theoretical basics and the algorithm to work with, I will first outline the terminology. This contains the establishment of terms and introducing a number of technologies of actual relevance for this thesis.

2.1.1 General Graph Structure

A graph is a conceptional structure representing a relation and consists of vertices and edges. Amongst other things the following chapter will be about extending its structure, but initially I define the most important terms needed to work properly with it.

In the core it is defined as pair (V, E) with V being the set of vertices and $E \subseteq \mathcal{P}(V) \times \mathcal{P}(V)$ the set of edges. In the literature each graph is either directed or undirected, whereas this thesis assumes it to be directed at any time. Hence for an edge e and two vertices s, t hold that $e = (s, t) \neq (t, s)$. So the edge has an explicitly defined *source* s and *target* t. Source and target could either be a node or a list of nodes. An edge with multiple sources or targets is called *hyperedge*.

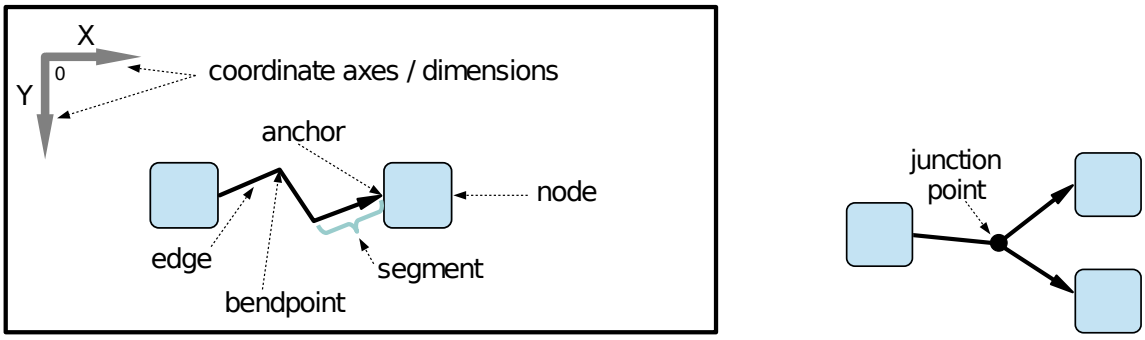
Further I define two functions $in\colon V \to \mathcal{P}(E)$, $v \mapsto \{(x, v) \mid x \in V\}$ with $in(v)$ representing all its incoming edges as well as $out\colon V \to \mathcal{P}(E)$, $v \mapsto \{(v, x) \mid x \in V\}$ with $out(v)$ representing all its outgoing ones. A vertex v with $out(v) = \varnothing$ is called a *sink*.

Furthermore I define the term *path* as sequence of vertices $(v_1, \ldots, v_i)$ for which holds that $\{v_1, \ldots, v_i\} \subseteq V$, $\{(v_1, v_2), \ldots, (v_{i-1}, v_i)\} \subseteq E$, and $i \in \mathbb{N}_{>2}$ being its length. A path $(v_1, \ldots, v_i, v_1)$ is called a *cycle*.

2.1.2 Layout

As a *layout* this work will refer to a graph's computation of relative placements in order to visualize it. For this matter there are given dimensions *width* and *height*. Within those dimensions each vertex gets arranged by a specific Cartesian coordinate (x, y) with $0 \leqslant x \leqslant d_w$ and $0 \leqslant y \leqslant d_h$ and d_w, d_h being the dimensions. Further it has a size, too, also consisting of width and height.

The most important components are hinted at in Figure 2.1. In general an edge is defined as pair of nodes (s, t) and is routed as straight line between those with an arrow head pointing at its target. Since nodes in a layout are no longer theoretical points but have actual shapes, it is important to



(a) A rendered layout and its components.

(b) A rendered hyperedge.

Figure 2.1. General descriptions of a layout's components.

define where the edge is linked to the node by *anchor points*. So an edge starts at its source's anchor point and ends at its target's one, between both there may be *bendpoints*. Such a bendpoint bounds the edge partially to a coordinate (x, y) and thus divides it in *segments*. This makes it a very important tool for routing edges, since now an edge could be defined as sequence $(s, b_1, \ldots, b_n, t)$ with s, t being anchors and b_i being a bendpoint with $i \in \{1, \ldots, n\}$ and $n \in \mathbb{N}$. Between those each segment is drawn as straight route from one point to another. In fact an edge may be a curve as well, but most of this work's theoretical discussions regard to angular ones.

As for hyperedges, besides the fact that they have multiple sources or targets those may be indicated by so-called *junction points*. A junction point is used as a semantic link between two segments and to beware the structure for being mistaken as crossing or overlap.

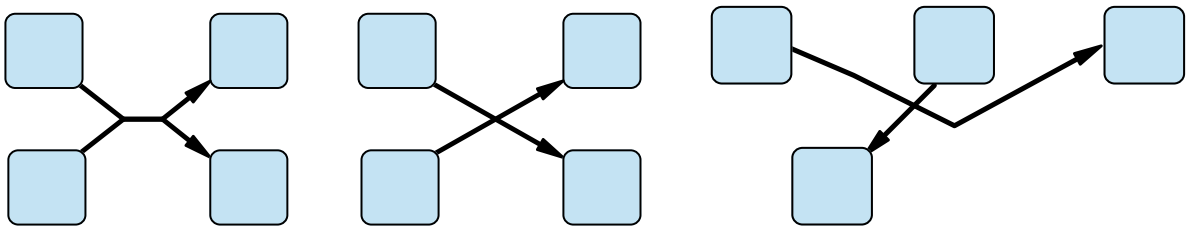
Layouts rely on a specific notion of *good* and *bad*. What a good layout may look like might be a matter of opinion. As described in the introduction it might be as well a matter of which kind of graph is visualized in the first place. There are a few general characteristics marking a good layout, though. In order to be comprehensible each layout should stick to those. The following three characteristics are ordered in their relevance of severity.

1. *Avoid overlaps*
   Avoiding overlaps is a very serious matter, since they make a layout in general incomprehensible. Those could happen to every component of the graph, for instance between two edges' segments like shown in Figure 2.2a. A node could be overlapped as well, either by an edge or another node.

2. *Use as few as possible bendpoints* [BET+98, Chapter 2]
   Every edge should be as short as possible to be comprehensible [PFJ95]. Hence this notion should only be broken, if it could avoid another. In the following work this will be only done if it could avoid an overlap.

3. *Avoid crossings*
   If possible crossings should be avoided, since two edges crossing each other are hard to comprehend as well [PFJ95]. There is an example of a crossing shown in Figure 2.2b. However, in most layouts the chances of crossing edges are accepted in order to adhere to the aforementioned second approach as it is done in Figure 2.2c.

Note that those are only examples, but the following work will stick to this notion and its given relevance.



(a) Two edges overlapping each other partially.

(b) Two edges crossing each other.

(c) Adding a bendpoint to an edge to prevent an overlapping of the node. The crossing is not avoided. Both could be avoided with a rearrangement of node locations.

Figure 2.2. Examples of bad layouts.

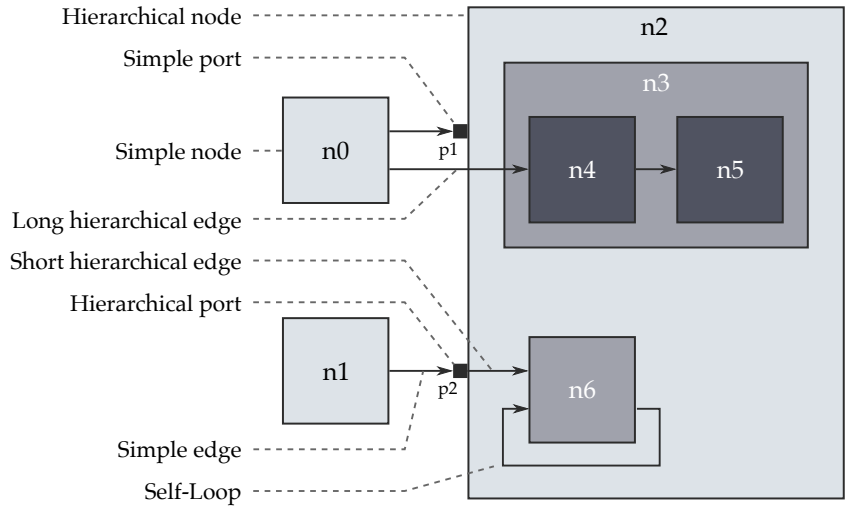


Figure 2.3. Hierarchical structure of an `ElkNode`. Source: [Sch]

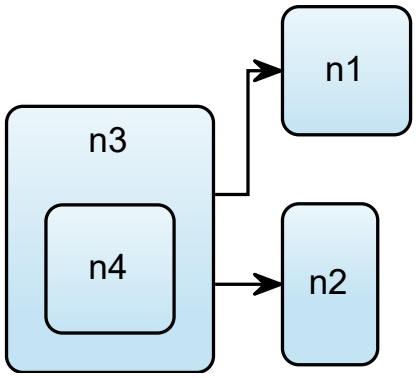
2.1.3 Eclipse Layout Kernel

In Chapter 1 the Eclipse Layout Kernel (ELK) was introduced as framework providing an infrastructure for automatic graph layouts. For this matter it provides a number of automatic layout algorithms, such as the layered algorithm, as well as the opportunity of implementing a new one. For this matter there is the `ElkNode` which is the graph structure used for the layout's actual computation. Every graph supposed to be laid out by ELK must be translated to an `ElkNode`. Afterwards it can simply be translated back by extracting the computed coordinates and sizes.

`ElkNode` relies on *hierarchy* and is thus meant as nested graph structure. Each instance could be either a simple node or otherwise hierarchical and hold its own graph. An example is shown in Figure 2.3 hinting at the components. Hence an `ElkNode` is a tree, with the root node being the main graph and each included node might include own graphs as well. A layout process will run recursively and hence starts always with the most nested graphs. A very mighty tool provided by the `ElkNode` is the management of additional options and properties. Those could be applied via the `IPropertyHolder`. This way one may change for instance the intended algorithm used for the layout or the routing phase if using the layered algorithm. Further note that the *ports* mentioned in Figure 2.3 stand for the anchors linking the edges to nodes. For now I leave the definition of an edge as given in Section 2.1.1 and will

```
1   // apply additional layout options
2   algorithm: layered
3   edgeRouting: ORTHOGONAL
4
5   node n1                    // define nodes
6   node n2 {
7      layout [size: 30, 50]   // define size
8   }
9   node n3 {
10     node n4                 // nested graph
11  }
12
13  edge n3 -> n2              // define edges
14  edge n3 -> n1
```

(a) As textual representation in ELK Text Format (ELKT).



(b) As a rendered layout.

Figure 2.4. A graph with three nodes and two edges described by two different representations.

extend it for using ports later in Section 2.3.

Such an `ELkNode` could be delivered to ELK directly or for example as JavaScript Object Notation (JSON). Besides, there is the ELK Text Format (ELKT) letting one define a graph directly in a very simple textual structure. An example for such a definition and the resulting layout is given in Figure 2.4. Nodes can be simply defined with the keyword `node` followed by a single sequence of characters defining its name which may be displayed as label in the final visualization shown in Figure 2.4b. Edges can be defined with the keyword `edge` as it is done in lines 12 and 13 bringing the defined nodes in relation. Optional values and properties can be passed to the graph, which is the root `ElkNode`, as it is done in lines 1 and 2. As for all other nodes, they can be include additional options as well. In line 9 the node `n3` defines a nested graph consisting of the simple node `n4`, whereas for the node `n2` there is an explicitly node size applied. Note that in the following work I will not explicitly define any node sizes most of the time, when hinting at a textual graph structure an example relies on. If not stated otherwise the nodes use an equal default node size as `n1` does.

2.1.4 KIELER and SCCharts

An example for the usage of ELK and especially its layered algorithm is the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER)[1]. This is a tool for modelling and visualizing Sequentially Constructed Statecharts (SCCharts) and further a research project of the RTSYS group. SCCharts is a visual language for specifying safety-critical reactive system [HDM+14] and are an enhancement of *Statecharts* introduced by Harel [Har87]. It relies on concurrent *regions* corresponding to threads. Within those regions there are *states* and *transitions*, further there has to be an initial state to start at. Each transition has a source state and a target state, is triggered by an input *signal* and may emit an own output signal in the process. Such an output signal can be an input signal for another region's transition. Hence there is a data flow between regions. In Figure 2.5 an example of SCCharts is shown. There is no need to understand the purpose of every region, but one may see that some transitions are triggered of

[1] https://www.rtsys.informatik.uni-kiel.de/en/research/kieler

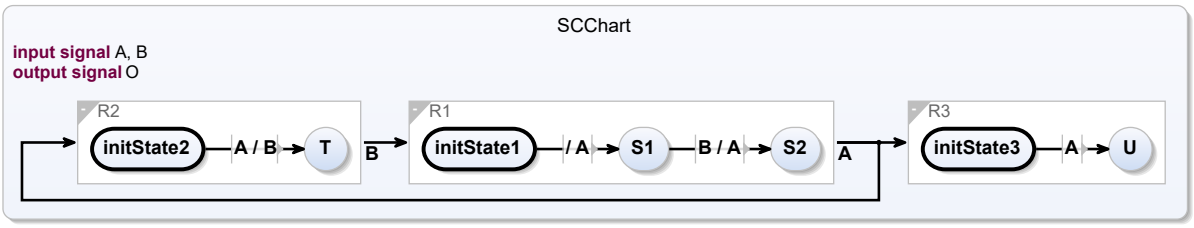


Figure 2.5. An example of Sequentially Constructed Statecharts (SCCharts) with data flow between regions R1, R2, and R3 routed by the orthogonal edge routing.

output from other ones. Those relations of one region's output being other regions' input is visualized by orthogonal edges. It means furthermore that the regions correspond to nodes.

2.2 The Sugiyama Approach

To really understand the introduced problem and the algorithm to solve it, it is necessary to realize it as part of many problems. This edge routing algorithm, which I will refer hereinafter as *edge router*, is the last of a number of phases. Hence it has no power over any node's vertical location already determined by the node placement. It only gets a layout of placed nodes and some more information about edges. Each of these phases is replaceable by another algorithm, which could result in a different layout, even if the edge routing remains the same. In the following I will introduce the *Sugiyama Approach* as context for the discussed algorithm, which will be introduced in detail afterwards.

The Sugiyama Approach was first introduced by Sugiyama et al. [STT81] and draws a directed graph by arranging the majority of its edges in the same direction. Its basic characteristic is the partitioning of the graph's node set into *layers*, which are the reason the approach is also called *layered approach*. Every node is assigned to exactly one layer. These layers are subsequent and rely on the routing direction, which as mentioned is here from left to right.

More precisely the approach is divided into five phases introduced in the following, each with a simple example:

2.2.1 Phase 1: Cycle Breaking

Assume an input graph $G = (V, E)$ where V is the set of vertices and E the set of edges.

Since the layer arrangement only supports acyclic graphs, the first phase is about breaking potential cycles of an input graph. This is important because as long as there are cycles it is impossible to draw every edge in the same direction. A common approach is to use the Feedback Set (FS), a subset of the directed graph's edges whose reversal makes the graph acyclic. Reversing an edge $e = (s, t) \in E$ with source $s \in V$ and target $t \in V$ means to swap source and target, so $e' = (t, s)$ is the reversed edge corresponding to $e \in$ FS. A very simple way of finding such a FS is to order all vertices along a horizontal line, so every edge is pointed to the right or left as shown in Figure 2.6. Now pick one direction (preferably the one with fewer edges) and add each edge headed towards it to the FS—now a reversal of all added edges results in an acyclic order for the graph. No vertex can now be part of a cycle, because there are no edges leading back to it [BET+98, Chapter 9]. Later the layered approach's final task will be to restore every edge's initial direction.

Since not every FS is actually a minimal one, a significant part of the cycle breaking is to minimize a possible solution. For example the proposed simple approach chooses a random order for the vertices.

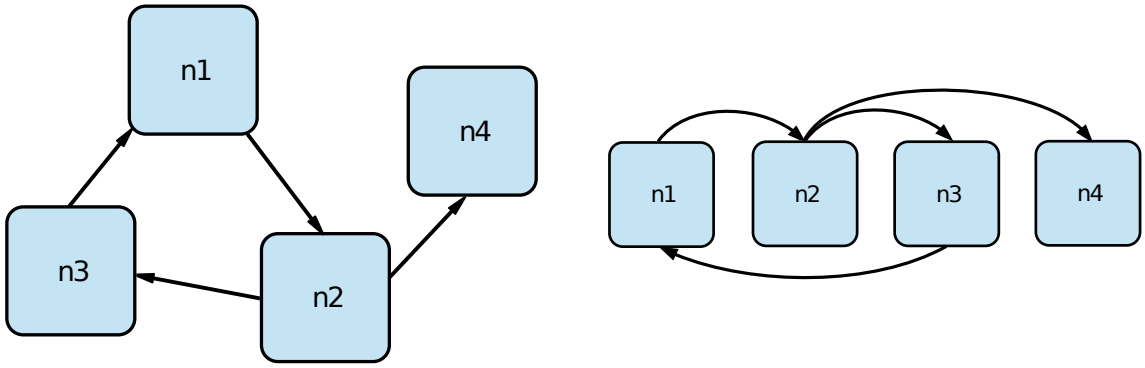


Figure 2.6. Comparison of a cyclic graph (left) and its linear representation (right).

So even if it is assumed to pick the direction with fewer edges to be reversed, an algorithm still has to modify half of all edges in the worst case. A more efficient and popular variant is the *greedy algorithm* presented by Eades et al. [ELS93], which relies on the proportion of outgoing to incoming edges for every vertex and may produce a better but still not optimal solution.

The final output of this phase is a new graph $G' = (V, E')$, where E' might differ from E if the phase actually reversed any edges.

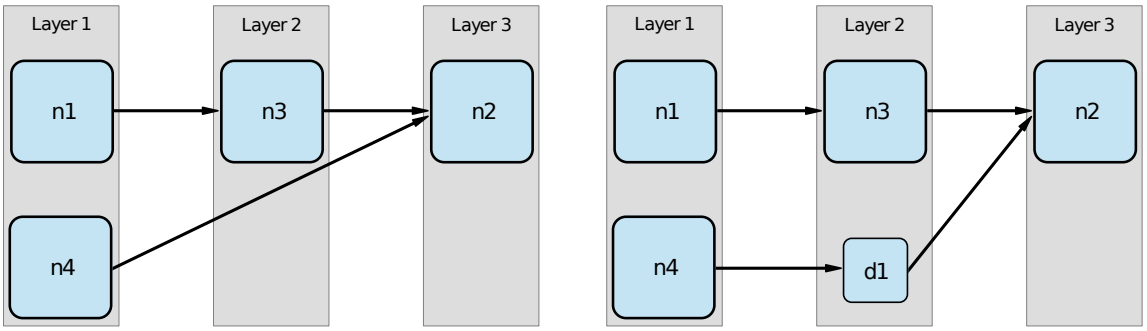
2.2.2 Phase 2: Layer Assignment

Assume an acyclic input graph $G = (V, E)$ where V is the set of vertices and E the set of edges.

The second phase introduces the layers, for instance as sequence $L = (L_1, \ldots, L_n)$ with $n \in \mathbb{N}_{\geqslant 1}$. Every vertex is assigned to a layer for example with a function $l\colon V \to L$. The most important characteristic for this partitioning is that every edge's source has to be in a layer with an index smaller than that of its target's layer, thus every edge points in the same direction. An example is shown in Figure 2.7a. So for every directed edge $e = (s, t) \in E$ with source $s \in V$ and target $t \in V$ and $j, k \in \{1, \ldots, n_L\}$ with $n_L \in \mathbb{N}_{\geqslant 1}$ the number of layers it holds that $l(s) = L_j \wedge l(t) = L_k \Rightarrow j < k$. A common approach for this is to compute the longest possible path in the graph and to define this many layers. Next assign all sinks to the highest layer, remove them from the graph and repeat the process for the next highest layers until all vertices are assigned. Through the removal of the sinks from the step before and the fact that the graph is acyclic, there are new sinks in every step and because there are as many layers as vertices in the longest possible path, there is at least one vertex in every layer.

However, the layered approach needs a *proper layering* which comes with an additional requirement. Every edge's source and target layer must be adjacent. So now for every directed edge $e = (s, t) \in E$ with source $s \in V$ and target $t \in V$ and $j, k \in \{1, \ldots, n_L\}$ with $n_L \in \mathbb{N}_{\geqslant 1}$ the number of layers holds that: $l(s) = L_j \wedge l(t) = L_k \Rightarrow k - j = 1$. This is the reason for the introduction of *dummy nodes*. Every long edge, which is an edge that does not satisfy the proper layering, fills their layer gaps with dummy nodes and is replaced by a number of *dummy edges* which respect the proper layering. For example the edge $e = (n4, n2)$ shown in Figure 2.7b with $n4 \in L_1$ and $n2 \in L_3$ is replaced by two new edges e_1, e_2 with dummy node $d_1 \in L_2$, such that $e_1 = (n4, d_1)$ and $e_2 = (d_1, n2)$.

The final output is a new graph $G' = (V', E', L, l)$, where V' is the unity of V and the set of dummy nodes, E' is E with no long edges but a possible number of dummy edges, L is a sequence of layers and l is the function assigning the vertices to those layers.



(a) The layering: All nodes are assigned to a layer. There is a long edge between layer 1 and layer 3.

(b) The proper layering: Introduced new dummy node d1.

Figure 2.7. An example layer assignment. Note that this is only meant as a conceptionally sketch. The nodes have no order until phase 3 has finished and yet no specific placement at all.

2.2.3 Phase 3: Crossing Minimization

Assume an acyclic input graph $G = (V, E, L, l)$ where V is the set of vertices, E the set of edges, L a sequence of layers, and $l\colon V \to L$ a function assigning every node to a layer.

The third phase is about reducing the number of crossings inside the current graph layout. Assume that each layer is a set of nodes with an own order defined by the function $w\colon V \to \mathbb{N}_{\geqslant 1}$, $w(x) \mapsto i$ with $i \in \{1 \dots n_L\}$ and n_L being the size of the layer $l(x)$.

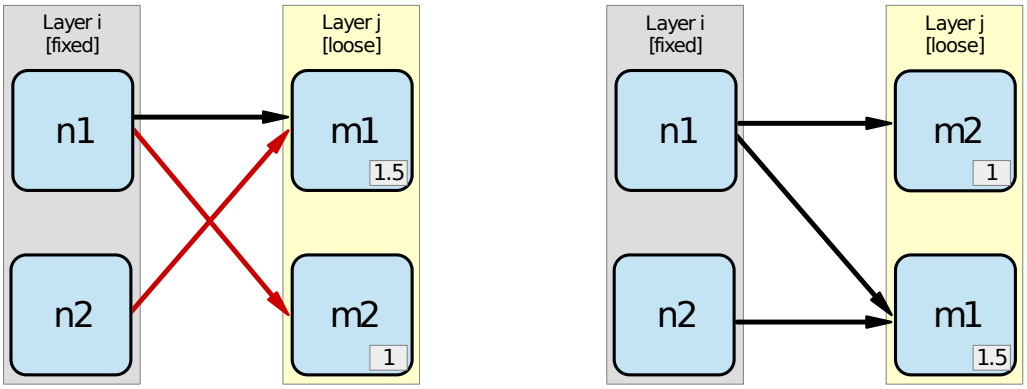
Thanks to the proper layering explained in Section 2.2.2, it is possible to adapt the number of crossings by just reordering the nodes. The *barycenter* heuristic also presented by Sugiyama et al. [STT81] for example is doing just this by analyzing the connections between pairs of adjacent layers. Declaring one layer as *fixed* and one as *loose*, the approach looks up the connected nodes from the layer to be sorted (loose) to the fixed layer. For each node in the loose layer it now picks all connected nodes in the fixed layer. For those connected nodes it sums up all its layer positions interpreted as *weights* and divides this sum by the amount of connected nodes. This results in an average weight w_a for the original node in the loose layer.

It becomes clearer if looking at an example. There is one given in Figure 2.8a where a crossing occurs. Two layers are taken in consideration: Layer i is fixed and Layer j is loose. Each position of a node in those layers is enumerated from top to bottom. Let us compute the average weight w_a for m1. For now for m1 holds that $w(m1) = 1$ since it has the first position in layer j. It is connected to two nodes in the fixed layer: n1 and n2 with the weights 1 and 2. Hence the average weight is $w_a(m_1) = \frac{w(n1)+w(n2)}{2} = \frac{1+2}{2} = 1.5$. As for m2, the average weight is $w_a(m_2) = \frac{w(n_1)}{1} = \frac{1}{1} = 1$.

Now one can simply sort the nodes by their average weights as done in Figure 2.8b. Here it would mean to create a new ordering function w' with an updated relation. Doing this for the whole loose layer is called a *layer sweep*. Normally one *turn* of this algorithm is to perform a layer sweep for every pair of layers from (L_1, L_2), (L_2, L_3) to (L_{n_L-1}, L_{n_L}) or from (L_{n_L}, L_{n_L-1}) to (L_2, L_1) with $n_L \in \mathbb{N}_{\geqslant 1}$ the number of layers.

It depends on the graph's complexity how many turns should be applied actually. Further the results may differ depending on the first layer's initial order. A common method is to run the algorithm various times with the same number of turns, each with a randomized order of the first layer, and then to pick the result with least crossings.

The final output is a new graph $G' = (V, E, L, l, w)$ with a new ordering function for the sequence of layers.



(a) A crossing occurs. Crossed edges are highlighted in red.

(b) Alter ordering of loose layer to eliminate the crossing.

Figure 2.8. Example layer sweep during a crossing minimization. The loose layer is highlighted in yellow and the nodes to be ordered hold their average weight.

2.2.4 Phase 4: Node Placement

The fourth phase determines the vertical placement for every node. There are two fundamental requirements: First, to draw every layer's vertices one below the other and second, to maintain the ordering between and inside the layers determined by the previous phases.

There are numerous ways to calculate the actual placement. For example one could care to place the nodes such that the edges could be drawn as straight as possible. A common simple approach is to let the dummy nodes shift the other nodes to the side, so the long edges become straight. A more complex approach is the *BK Placement* by Brandes and Köpf [BK01].

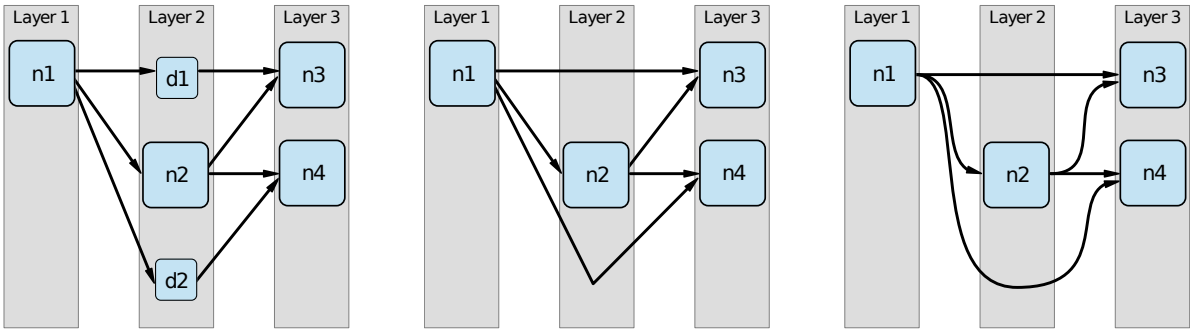
2.2.5 Phase 5: Edge Routing

The last task is to draw the edges, replace the dummy nodes and to restore their original direction. Depending on the preferred style of routing there are many ways to do this. A few examples are shown in Figure 2.9. The easiest way would be to simply draw straight lines from source to target and to replace the dummy nodes with bend points like in Figure 2.9b. In this case the direction could be reversed by simply placing the arrow-head of the drawn edge on the other side. In fact this phase may have to deal with hyperedges as well—that is, edges with multiple sources or targets. Another approach is the *orthogonal edge routing* explained in the following.

2.3 Orthogonal Edge Router

The *orthogonal edge router*, which this work is about, was already hinted at during Chapter 1. Now I dive deeper into its functionality.

The orthogonal edge router considers every adjacent pair of layers and routes their edges by using only vertical and horizontal line segments. Thus, most important for this phase is the space between the layers, the *routing space*, which is determined by this algorithm. Even after the node placement phase has finished, the actual positioning of nodes is not final. The vertical coordinate is fixed, but the horizontal one is not, because it depends on the computed routing space.



(a) Initial layout right after node placement.

(b) Routed with straight edges.

(c) Routed with curved edges.

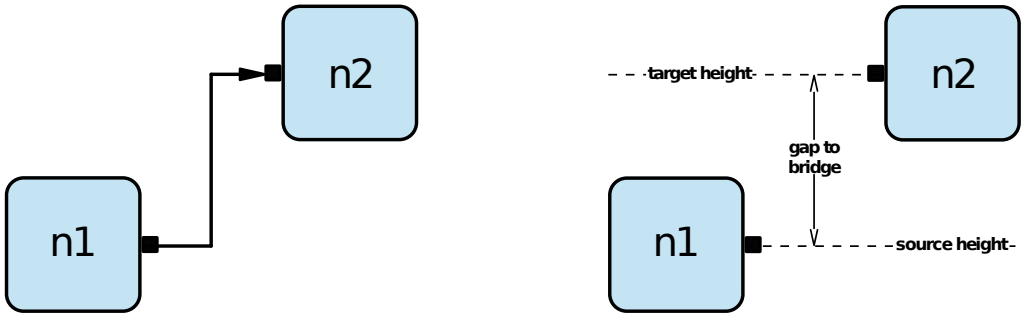
Figure 2.9. Two different routing strategies of a setting given by the node placement.

Further it works not directly on nodes but on their ports. I already mentioned those in the terms describing part of this chapter, now I introduce them properly. Such a port represents a potential link between a node and its edges and corresponds to the anchor described in Section 2.1.2. Henceforth an edge's source and target are no longer defined as nodes but as ports. The same holds for a hyperedge's sets of source and target. In the input those ports may be explicitly defined, otherwise an edge's source or target node is replaced by a dummy port. In detail, assume for example two nodes $n1$ with port p and $n2$ with ports $q1$ and $q2$. Now an edge $e1$ could be defined explicitly by the ports, for example $e1 = (n1.p, n2.q1)$, but a second edge $e2$ could be defined like $e2 = (n1, n2.q2)$ with a bare node as its source. Thus $e2$ would be replaced by $e2' = (n1.d, n2.q2)$ with $n1.d$ being a dummy port d added to the node $n1$. The general advantage of ports is to gain control over an edge's anchor points. For example, one could specify their positions or fix their internal ordering, because the location of every port is determined by the node placement phase, as long as not especially demanded otherwise. By the time edge routing runs, a port is applied to a node's side and the algorithm does distinguish the ports in sources and targets depending on this side. So as long as the routing direction is defined as right (or east) for a routing between two adjacent layers, a *source port* is defined as bounded to the right side of a node in the left layer, whereas a *target port* as bounded to the left side of a node in the right layer. An example is given in Figure 2.10a, where the explicitly defined ports are recognizable as little black squares. While n1 is part of the left layer and has the edge's source port on its right side, n2 is part of the right layer and has the edge's target port on its left side. Note that, since the cycle breaking may have reversed some edges of the graph, the notion of source and target does not always correspond to the actual input until the natural direction is restored in the end.

Let us move on to the actual process of routing edges between a pair of layers as it was when I started work on this thesis. For this matter I start with a general description before describing details.

2.3.1 In General

The most general characteristic is that the orthogonal edge routing only generates vertical and horizontal segments. Figure 2.10 shows an example of an edge routed with this approach as well as the actual setting leading to this routing. There are three segments in total: two horizontal ones and a vertical one. Within a routing space each edge has a source and a target specified by ports and may have to bridge any vertical gap between them. Those corrections in height, which are the cause for vertical segments, are placed in so-called *routing slots*. The main task of the orthogonal algorithm



(a) The actual routed edge. The segment with an arrowhead is the *target segment*, the other horizontal one is the *source segment*.

(b) The given setting to compute a routing.

Figure 2.10. Routing an edge from n1 to n2, both with an explicitly defined port, using the orthogonal edge routing.

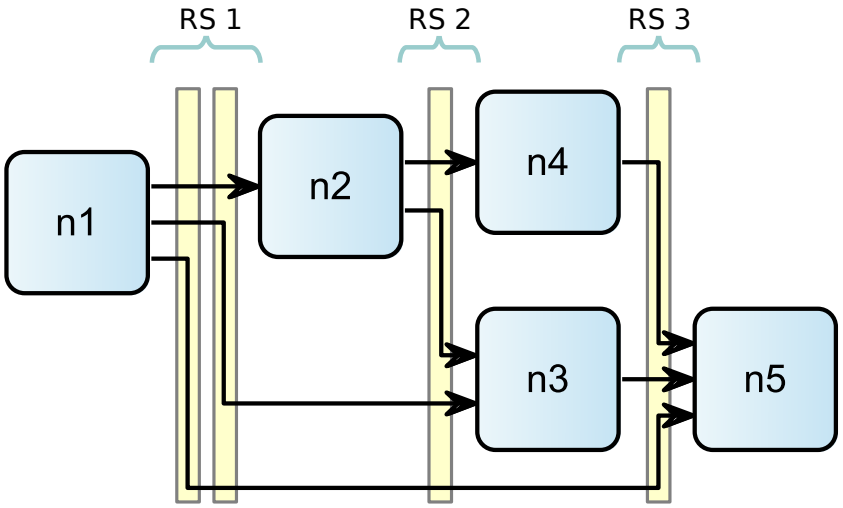
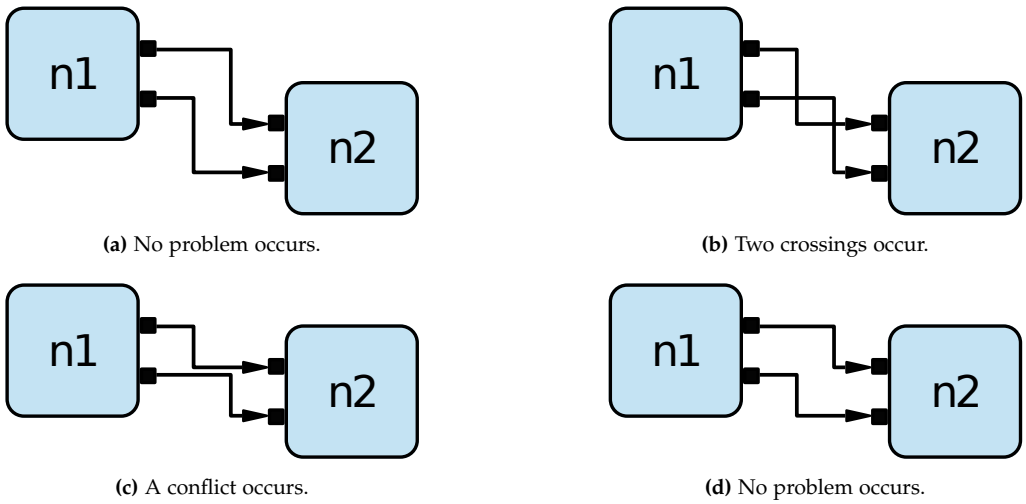


Figure 2.11. An orthogonal layout with highlighted routing slots and routing spaces (RS).

is to assign such a slot to the edges and to decide whether multiple edges can share one and thus have their vertical segments on the same horizontal coordinate. This happens if that does not result in overlapping segments. Otherwise the edge router adds another slot which causes one edge's vertical segment to be shifted rightwards and the space between the layers to grow. In order to produce a good layout the algorithm should avoid as many routing slots as possible, so the routing space does not grow excessively and stretch the final layout's width more than necessary. In Figure 2.11 for example there are two routing slots within the first routing space (RS 1), because the vertical segments cannot be drawn on the same horizontal coordinate without overlapping. The vertical segments of the rightmost routing space can share one slot without causing any problems. In conclusion there are two general and very important assumptions about the orthogonal edge router directing most of its strategy. First, every edge is drawn with at most one vertical segment. Second, the algorithm's actual objective is to determe the horizontal placement of those vertical segments, which will be explained in the following.



(a) No problem occurs.

(b) Two crossings occur.

(c) A conflict occurs.

(d) No problem occurs.

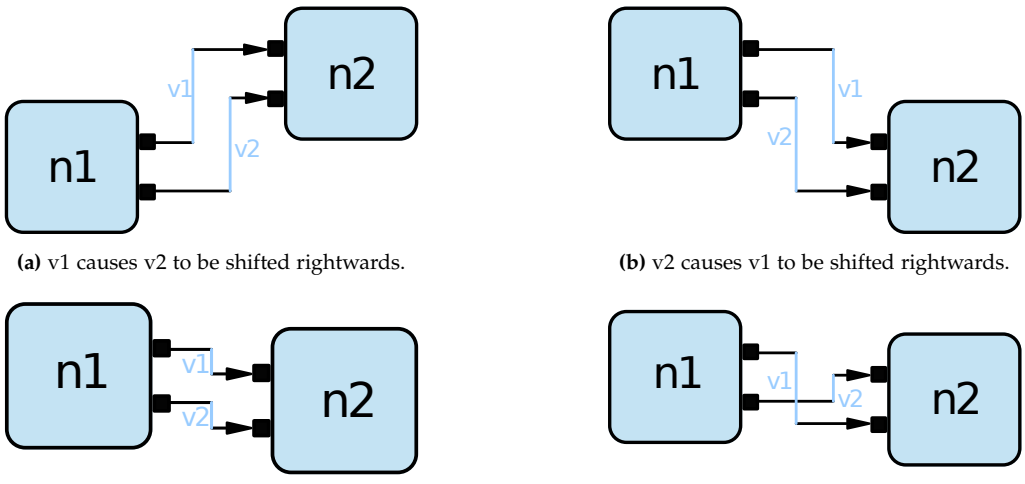
Figure 2.12. The relevance of ordering multiple vertical segments.

2.3.2 The Dependency Graph

By studying Figure 2.10b one may notice that, since the vertical coordinates of the ports are fixed, the source's and target's heights are absolutely final. Thus the only thing the algorithm could change about the horizontal segments would be their length. Of course those depend on the vertical segment's placement. For example placing the vertical segment more to the right would cause the horizontal *source segment*'s length to increase while the *target segment*'s length decreases and vice versa if placing it more to the left. But of course the sum of both is always the width of the routing space. After all this is the reason the algorithm cares only about placing the vertical segment, which more precisely relies on the routing slots. So the whole process of finding the best placement is a process of defining the number of routing slots and assigning the vertical segments to them—or in other words, it is a matter of ordering the vertical segments. Such an order is vital to avoid problems which could be caused by the horizontal segments. To demonstrate the necessity of a proper order and to introduce the two core problems, there are two graphs shown in Figure 2.12, each with two different orderings. For example the graph in Figure 2.12a needs two routing slots, because otherwise the vertical segments would overlap. Figure 2.12b shows the same graph with a different ordering causing two crossings. The graph in Figure 2.12c describes a more complicated problem, the *conflict*. In order to produce a good layout the algorithm always tries to avoid a conflict, which is the case for two horizontal segments coming too close. The notion of *too close* is an arbitrary value defined by the algorithm itself. Again we need a proper ordering like in Figure 2.12d and two routing slots overall, because otherwise the vertical segments would come too close. Now I describe the process of arranging the vertical segments, as well as the structure it relies on.

As aforementioned, a great amount of the edge router's work is a process of ordering vertical segments. There are routing slots to place the vertical segments inside. Some of those segments may share routing slots, but for this matter it is vital to find a good arrangement of assigning those to the slots, so they cause as less problems as possible. This arrangement is determined by the *dependency graph*, an internal graph structure relying on an idea of Georg Sander [San04]. It serves to outline relations of relative placements between vertical segments. What this means will become clear in the following.

The algorithm works on its own representation of a vertical segment, the *hypernode*. For cases



(a) v1 causes v2 to be shifted rightwards.

(b) v2 causes v1 to be shifted rightwards.

(c) Both segments share the slot.

(d) No version is better.

Figure 2.13. Four different outcomes during comparison of vertical segments v1 and v2.

in which the differences between both terms do not absolutely matter, I proceed to refer to them as vertical segments, though. The router creates a node in the dependency graph for every vertical segment within the routing space with the final goal of computing the *rank* for every vertical segment, which is in other words the index in the ordering of routing slots. But first it needs to find out the edges for the dependency graph by comparing all vertical segments pairwise and deciding for each pair whether one of them has to be *shifted* to the right to avoid problems. If a node is shifted to the right, this means that its rank must be higher than the rank of the shift causing node.

Assume such a pair of vertical segments v_1 and v_2. The algorithm compares the layout for two states: First, v_2 is to the right of v_1 and second, v_1 is to the right of v_2, which I will refer hereinafter as $v_1 < v_2$ and $v_2 < v_1$, respectively and for a *partial layout* $v_1 < v_2$ the *alternative* layout is $v_2 < v_1$. Further I call a layout *better* in case it produces fewer crossings or conflicts than another one (for now the algorithm assumes one conflict to be much worse because this could be an overlap as well). For such a better layout the algorithm may want one vertical segment to shift another one to the right and represents this as *dependency* and thus as edge in the dependency graph. During the comparison of v_1 and v_2 there are now different outcomes distinguished in four cases corresponding to Figure 2.13:

(a) v_1 shifts v_2 to the right, because the layout for $v_1 < v_2$ is better than the one for $v_2 < v_1$. The algorithm creates a dependency and thus an edge $d = (v_1, v_2)$ in the dependency graph.

(b) v_2 shifts v_1 to the right, because the layout for $v_2 < v_1$ is better than the one for $v_1 < v_2$. The algorithm creates a dependency and thus an edge $d = (v_2, v_1)$ in the dependency graph.

(c) There is no need to shift one of them anyway, because both layouts produce no crossings or conflicts. The algorithm creates no dependency and thus no edge in the dependency graph.

(d) It is not relevant which one is shifted, because both layouts produce the same number of crossings or conflicts. The algorithm creates two dependencies and thus two edges $d_1 = (v_1, v_2)$ and $d_2 = (v_2, v_1)$ in the dependency graph.

The vital difference between case **(c)** and case **(d)** is the position of the vertical segments: in case **(c)** for both layouts there are no crossings or conflicts, because the segments are on completely different

heights. Hence, they could share the same routing slot provided that no other dependency prevents this. In case **(d)** the shapes of both vertical segments intersect and thus they must not reside in the same routing slot in order to avoid an overlap. As for horizontal straight edges, the router creates no dependencies to those, since there are no vertical segments to be arranged. In fact it generates hypernodes for them, though, but since there are no edges in the dependency graph I spare those in most of my examples.

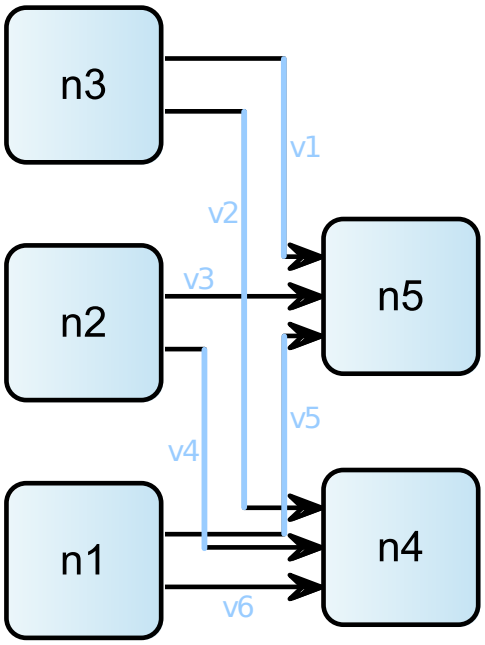
Furthermore, each dependency has a *weight* indicating how much better the layout is than its alternative. For this matter the comparison counts each occurrence of conflicts and crossings for both partial layouts and uses those values to compute the weight. For example if the layout $v_1 < v_2$ causes two crossings and no conflict, while $v_2 < v_1$ causes six crossings and no conflicts as well, than the result is a dependency (v_1, v_2) with a weight of $6 - 2 = 4$. Case **(d)** creates two dependencies each with zero weight, which is called a *two-cycle*.

2.3.3 Assigning the Routing Slots

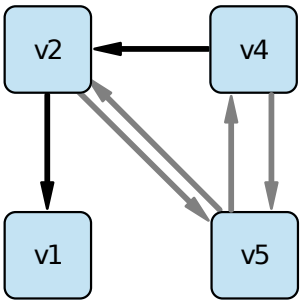
After the algorithm compared all vertical segments and created all dependencies, the internal dependency graph must be checked for cycles. This is vital, because a cycle in the dependency graph means a contradictory recommendation about the ordering. For example, three vertical segments v_1, v_2, v_3 in a dependency graph forming a cycle with three dependencies (v_1, v_2), (v_2, v_3) and (v_3, v_1) would result in a layout for which holds that $v_1 < v_2 < v_3 < v_1$. The reason is that this cycle means that v_3 should be right hand of v_1, while in contradiction v_1 should be right hand of v_3. Since the realization is impossible this cycle must be broken.

For this matter it tries to find a set of dependencies with the lowest sum of weights to reverse them in order to break all cycles. Now it is important to be aware of the meaning of reversing a dependency. Until now the router always chose the dependency causing the better layout. So reversing a dependency (v_1, v_2) means choosing the dependency (v_2, v_1) and thus setting for a layout where holds $v_2 < v_1$ instead $v_1 < v_2$. The simple fact that there is a dependency (v_1, v_2) means that the partial layout $v_1 < v_2$ is better and its dependency's weight indicates by how much. So finding such a set of minimal weights is important, because if the algorithm really has to reverse an edge, this means to increase the number of crossings or overlaps. Reversing the set with minimal sum of weights means choosing the least bad solution. Furthermore it breaks every two-cycle by deleting one of its edges. Note that, since finding such a set of minimal weights is an NP-complete problem[BET+98, Chapter 9], the algorithm's solutions may be not optimal. There is an example graph and its final dependency graph in Figure 2.14. The lighter colored edges of Figure 2.14b rely on two-cycles. Further now one could see the reason why the algorithm needs those two-cycles in the first place, instead of simply letting the comparison decide on one dependency. For instance in Figure 2.14b there is a cyclic path (v2, v5, v4, v2) which would still exists after deleting for example the dependencies (v4,v5) and (v5,v2) in order to break the two-cycles. Instead the algorithm deletes the dependencies (v5,v4) and (v5,v2) and thus breaks all cycles with a set of dependencies whose sum of weights is zero.

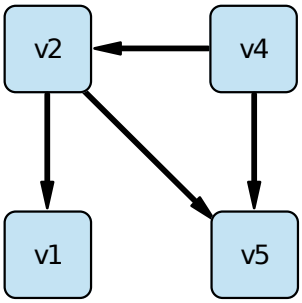
The next task is to assign the actual rank value to those vertical segments. Since the dependency graph is now free of cycles the algorithm computes the longest possible path in the graph and defines this many routing slots for all vertical segments to be assigned to. Like the cycle breaking in the step before this is reminiscent of a phase of the layered approach. Similar to the layer assignment phase the algorithm now assigns all vertical segments to routing slots depending on their incoming and outgoing dependencies. In Figure 2.15 this process is described for the setting shown in Figure 2.14. As described in Section 2.2.2 for the assignment to layers the algorithm now puts all sinks in the routing slot of the highest rank and deletes them from the graph. For the given example this means firstly to



(a) The rendered graph with highlighted vertical segments.



(b) The dependency graph before cycle breaking.



(c) The dependency graph after cycle breaking.

Figure 2.14. A graph with the corresponding dependency graph. Note that v3 and v6 are theoretical vertical segments not relevant for the dependencies and thus left out in visualized dependency graphs.

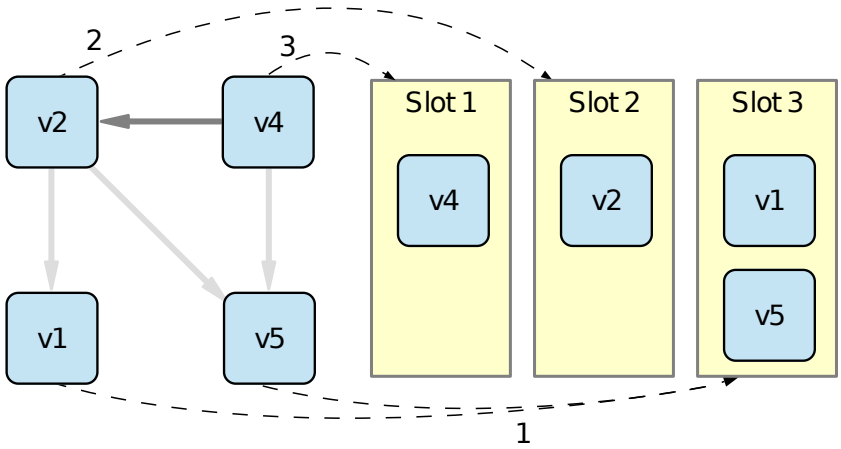


Figure 2.15. The assignment of routing slots which leads to Figure 2.14a and relies on the acyclic dependency graph Figure 2.14c. There are two potential longest paths: (v4, v2, v1) and (v4, v2, v5), both of size 3. Corresponding to the enumeration, the edges of lighter color hint at the chronological order of steps.

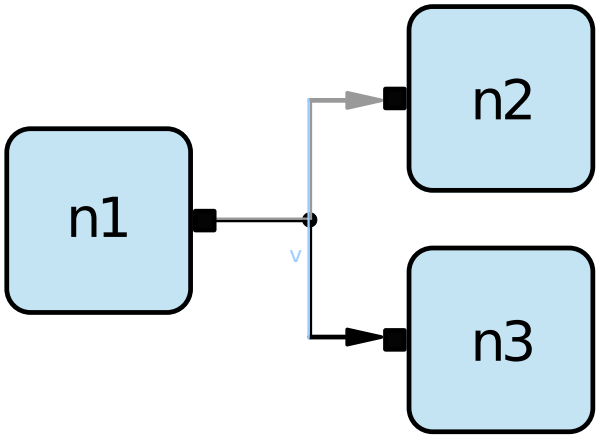


Figure 2.16. Construction of a hyperedge forming the same hypernode. The hyperedge is constructed by two regular edges sharing the same source port. Both vertical segments touch and construct a bigger vertical segment v.

assign v1 and v5 to routing slot 3 and deleting them from the graph. Now v2 is the only sink, which is assigned to routing slot 2 in the second step. After deleting v2 there is only v4 left, so it is assigned to slot 1 in the last step. The chronological order of deletions of nodes and edges in the dependency graph is hinted at by the edges' color becoming lighter. The routing space of the graph has three routing slots, because of the longest possible path's size of 3. Note that had the algorithm deleted the dependency $(v2, v5)$ to break the two-cycle, the longest possible path would be of size 4, causing the routing space to be wider than necessary. So this is another reason to be careful when deciding on how a two-cycle is supposed to be broken. In the end all vertical segments are assigned to a routing slot's rank. According to this rank value the algorithm places the vertical segments and routes the complete edge by simply placing two bend points, one at the end of the source segment and one at start of the target segment.

It is important to note that the actual routing process does not distinguish between hyperedges and regular edges. In fact the router does only know regular edges, but merges a set of them into one hypernode if they share a port. By using the same port two edges are forced to have overlapping source or target segments. Since the complete hyperedge uses only one hypernode with one rank, their vertical segments overlap or touch as well in order to bridge its own height-gap. Hence in the final layout it is understood as one vertical segment. In this case the difference between a hypernode and a vertical segments does actually matter. This relies on the fact that it is important to understand a hyperedge as set of regular edges forced in the same hypernode. Those form together one homogenous vertical segment in the final layout. Still each edge is routed independently. In Figure 2.16 there is an example of a hyperedge consisting of two regular edges.

Chapter 3

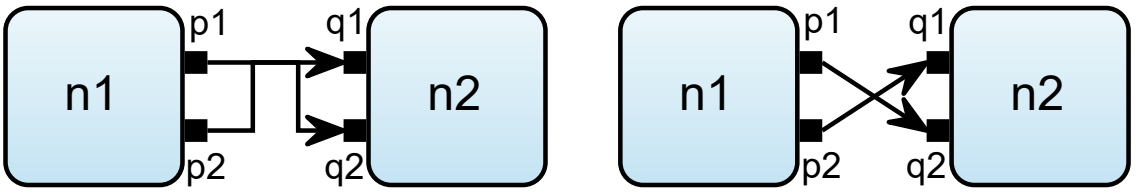
Theory

Before starting to code, it is vital to think through as many potential problems I might encounter as possible. As described in Section 2.3 the original algorithm assumes the edges to have at most one vertical segment. But the proposed solution makes it possible for an edge to have more. Breaking such a fundamental assumption will come with a number of problems during implementation. Besides, there may be some parts of the original algorithm which I have to adapt in order to integrate the new features. For those theoretical analytics there is an important fact which needs to be mentioned again. The current algorithm uses a threshold to detect conflicts which relies on a notion of two edges coming too close to each other. In the current state the algorithm does not distinguish between pure conflicts and overlaps, because both are cases of segments coming too close. When analyzing layouts I will indeed do this, though. For this matter it is now necessary to being absolutely clear about the difference of conflicts for the algorithm, conflicts for me and actual overlaps. This is the reason I will refer hereinafter to every conflict which is no overlap as a *pure conflict*, while a general conflict for the algorithm still could be an overlap.

This chapter starts with a detailed description of the general problem.

3.1 The Problem

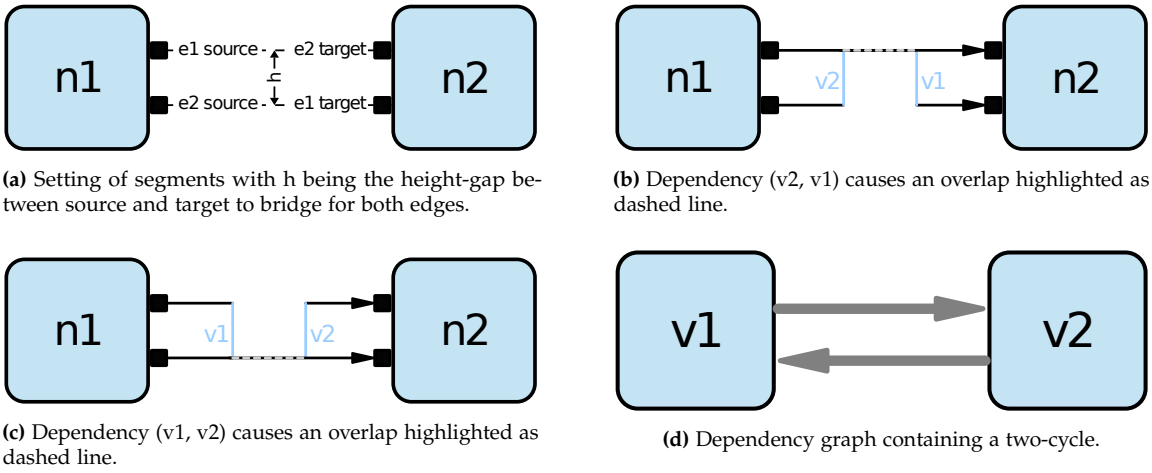
In Chapter 1 I already introduced the problem of overlapping horizontal segments. Figure 3.1a shows this problem again, this time with additional port information. Of course there are more examples of overlaps which I will show later in this chapter, but for the introductory explanation I use this case, because it is easy to comprehend. Furthermore this is the reason why I refer to it in this work by the graph's working title, the *Simple Case*. There are two nodes, each in its own layer and both placed at the same y-coordinate, just like their ports. As mentioned before, the vertical placement of the nodes and ports is fixed by the time the algorithm starts the routing process. Now there are two edges $e1 = (\mathrm{p1}, \mathrm{q2})$ and $e2 = (\mathrm{p2}, \mathrm{q1})$. The simple solution of routing both edges as straight lines would form an *X-shaped* symmetric crossing, which is shown by laying out the same graph with the polyline algorithm in Figure 3.1b. Using the orthogonal edge router, each regular edge is drawn with three segments. In Figure 2.10b (see page 14) I explained the composition of orthogonal segments



(a) With orthogonal routing. An overlap occurs.

(b) With polyline routing. An X-shaped crossing occurs.

Figure 3.1. The *Simple Case*.



(a) Setting of segments with h being the height-gap between source and target to bridge for both edges.

(b) Dependency (v2, v1) causes an overlap highlighted as dashed line.

(c) Dependency (v1, v2) causes an overlap highlighted as dashed line.

(d) Dependency graph containing a two-cycle.

Figure 3.2. Algorithm's steps for routing the Simple Case's edges with hypernodes v1, v2.

for an edge with two different heights for source and target, where the edge has two fix horizontal segments and a height-gap between them to bridge. Corresponding to this one can see the same kind of composition of segments for the Simple Case in Figure 3.2a. Now there are two height-gaps to bridge for each edge, but it turns out that the source segment of $e1$ is on the same height as the target segment of $e2$ and vice versa. Despite the fact that both bridge the same gap, each needs its own vertical segment. This is why there are two vertical segments in the end and thus two hypernodes in the dependency graph. Since those hypernodes have to bridge the same height-gap there must be two routing slots. When comparing both hypernodes in order to find the best arrangement now both alternatives suffer from overlapping, which is visualized in Figure 3.2b and Figure 3.2c. Therefore the algorithm adds a two-cycle to the dependency graph in Figure 3.2d, meaning that no alternative layout is better, but both vertical segments need an own routing slot to not overlap. The dependency graph consists only of this two-cycle. Hence the cycle breaking deletes one of its dependencies and decides on the other. After all the algorithm has no choice but to leave the layout with its overlap, because there is no other choice than swapping the vertical segments' placements. For that reason an alternative must be found. As already hinted at during the introduction, this alternative implies additional segments for one of the edges involved in the overlap. Those would give it more flexibility for bridging the given height-gap, such that it would be possible to do this with two or more vertical segments instead of one. What this may look like was sketched in the introduction as well and will be discussed in more detail later on.

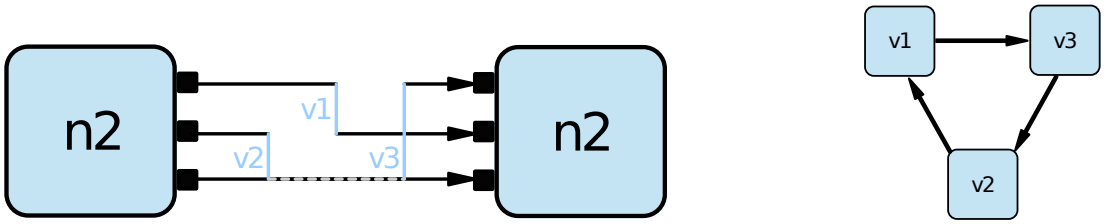
3.1.1 Critical Cycles

During the work on the general task I encountered an additional problem causing overlaps. Using the Simple Case I described the situation of a two-cycle with both dependencies producing overlapping horizontal segments. The cycle breaking needs to break this cycle and decide for one alternative. Which one to decide for is in this case irrelevant, since both cause an overlap. But there may be cases like the one shown in Figure 3.3 which I will refer to as the *Cycle Case*. There is a number of partial layouts with each having one partial layout creating an overlap and an alternative one which does not. Hence for each vertical segment there is a dependency avoiding an overlap. If those dependencies form a cycle in the dependency graph one of them must be reversed and the overlap, which it was intended

(a) With polyline routing. Two crossings occur.

(b) With orthogonal routing. A crossing and an overlap occur.

Figure 3.3. The *Cycle Case* caused by three dependencies each avoiding an overlap. They form a cycle, which is why the cycle breaking must reverse one of them.



(a) Orthogonal routing with hinted hypernodes and the overlap highlighted as dashed line.

(b) Dependency graph before cycle breaking. Contains a cycle.

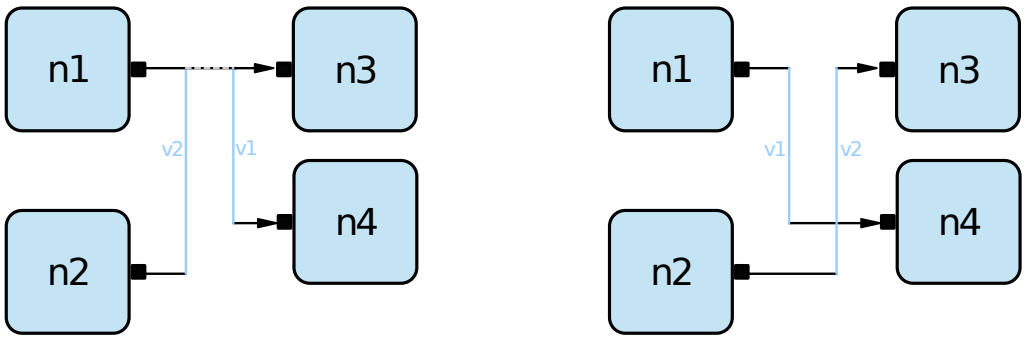
Figure 3.4. Ordering of hypernodes v1, v2, v3 for the Cycle Case.

to avoid, would be visible in the final layout. Referring to Figure 3.4a one can see that the comparison checks three pairs of hypernodes v1 and v2, v1 and v3 as well as v2 and v3. When comparing for instance v1 and v2 the partial layout v1 < v2 causes an overlap, while its alternative v2 < v1 does not. The same holds for the other two pairs. For each pair there is one alternative causing an overlap and one which does not. In the end this results in a cycle in the dependency graph shown in Figure 3.4b. In this case the cycle breaking reverses the dependency (v3,v2), which produces an overlap in the final layout. This problem could be handled together with the first problem, because additional segments could solve it as well.

The actual difference between both is the difficulty of detecting them, which depends on the number of hypernodes in the causing relation. The first problem was caused by only two hypernodes, while this one relies on at least three. As long as the problem is restricted to only two hypernodes, or in other words to a two-cycle, the algorithm could already detect it during the comparison of hypernodes. Any other cycle must be detected afterwards.

3.2 Relevance and Reproduction

The next task is to discuss the actual relevance of the problem this thesis is about. In other words, how often does this problem occur and how badly does it influence the layout? However, the very first thing to do is to reproduce the problem, which turns out to be harder than expected. As simple as the aforementioned X-shaped symmetric crossing may seem at first glance, its reproduction relies on the fact that there are ports that belong to different nodes, but share the same y-coordinate. At this point it is important to note that an overlap is always caused by two ports sharing the same vertical coordinate. Consider the example given in Figure 3.5. The ports of n1 and n3 are placed at the same y-coordinate and result an overlap for the layout shown in Figure 3.5a. As long as its alternative, shown in Figure 3.5b, has no overlap the algorithm chooses that and the overlap will not be visible in the final



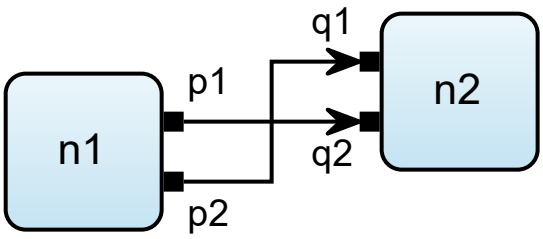
(a) Layout for which holds that v2 < v1.

(b) Layout for which holds that v1 < v2.

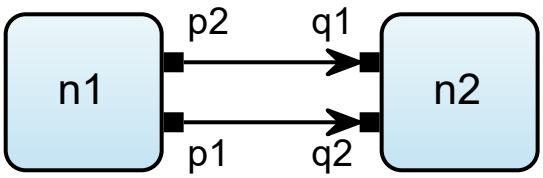
Figure 3.5. Creating dependency between two hypernodes suffering from overlap.

```
1   nodePlacement.strategy: SIMPLE
2   node n1 {
3      portConstraints: FIXED_ORDER
4      port p1 {portSide: EAST}
5      port p2 {portSide: EAST}
6   }
7   node n2 {
8      portConstraints: FIXED_ORDER
9      port q2
10     port q1
11  }
12
13  edge n1.p1 -> n2.q2
14  edge n1.p2 -> n2.q1
```

(a) Code for reproduction provided in ELK Text Format (ELKT). It contains additional restrictions.



(b) Layout without node placement restriction.



(c) Layout without fixed port order.

Figure 3.6. Things to consider when reproducing Simple Case with ports p1, p2, q1 and q2.

layout. To force the algorithm to decide upon the inferior layout, one must bring the ports of n2 and n4 to the same height as well. But in a real application example the nodes do not always have the same size nor the same number of ports, which may result in non-symmetric layouts and different port positions making overlaps unlikely to appear. As for the Simple Case, it needs a number of additional settings for reproduction which an average user may not even set in the first place. Nevertheless I start the discussion of relevance by describing how to reproduce the Simple Case, since it was the issue which initiated this thesis, and then continue with more common examples showing that the problem is indeed relevant.

3.2.1 The Simple Case and Hierarchies

Figure 3.6a shows the code in ELK Text Format (ELKT) leading to the layout of Simple Case. There are additional restrictions needed to trigger the overlap. First, in line 1 the node placement is changed

to a simpler one, because the default node placement strategy, which is an adaptation of the one by Brandes and Köpf [BK01], leads one of the edges to be rendered as a straight line by moving one of the nodes upwards (see Figure 3.6b) and therefore avoids an overlap. The second restriction in lines 3 and 8 overcomes a more general feature; by default the port positions are not fixed, which leads an X-shaped crossing to be routed as two horizontal straight lines (see Figure 3.6c) with no crossing at all by simply swapping the positions of one of the node's pair of ports. Additionally one node's ports must be in lines 4 and 5 forced right hand, because for ELK a fixed port order implies a fixed port side, which is left on default.

It may seem like one has to force the algorithm to fail. However, the high amount of variety provided by ELK's layered algorithm is the reason both properties may be activated unintentionally by other characteristics. For example the intervention of the default node placement is based on the simple intention of minimizing the amount of bendpoints by drawing an edge straight, if possible. This could easily be avoided by adding a third straight edge. Now the node placement does not change the nodes' positions, because it would cause the third, currently straight edge to lose its shape.

As for the port ordering, there are a number of other ways to fix it unintentionally, for instance with hierarchical layouts. I already explained in Section 2.1.3 that ELK models hierarchy as a node containing its very own graph structure. Such an included graph influences its *parent* node's size and is rendered inside. In Figure 3.7 there is an example of a hierarchical graph, its ELKT code as well as the tree structure belonging to the graph. Note that the edge (p3, q3) is the aforementioned straight-lined edge supposed to fix the node placement without changing the whole node placement strategy. The actual graph includes three sub graphs in total, each consisting of two nodes. There are also four ports p1, p2, q1, q2 connecting the nested graphs with the root graph's nodes. Computing the layout of a hierarchical graph runs recursively and starts with the innermost graphs, that is those with the deepest depth in the hierarchical tree. These are the graphs inside n1 and n2. After the algorithm routed both it cares for the root graph containing n1 and n2. Since the nested graphs' layouts are already finished and thus fixed, the ports are fixed as well, because changing their placement now would probably invalidate the routes if the edges connected to the ports from the inside.

3.2.2 A More Likely Case

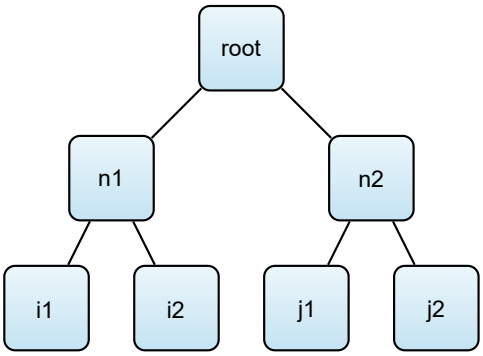
This example is more likely to occur, the *4Nodes Case*. It is shown in Figure 3.8b and its code in Figure 3.8a. Despite an equal node size there are no restrictions or special requirements as in the cases before. Furthermore, the vertical segments belong to nodes which are not at the same height. Each node's position is bounded to the same height in order to draw the outer edges straight, which are placed outward by the crossing minimization so the other edges do not cross them when bridging their height-gap. The edges are linked with dummy ports which were added by the layered algorithm and which are by default equally distributed over a node's side. Hence, as long as the nodes' sizes and number of ports remain equal their ports will be placed at the same heights. In Figure 3.8c I reproduced the case in SCCharts using the induced data flow mode. For this case to appear I used equal node sizes for the regions. Note that this is no general characteristic for a layout in SCCharts, since those are highly variable.

3.2.3 Conclusion and Relevance

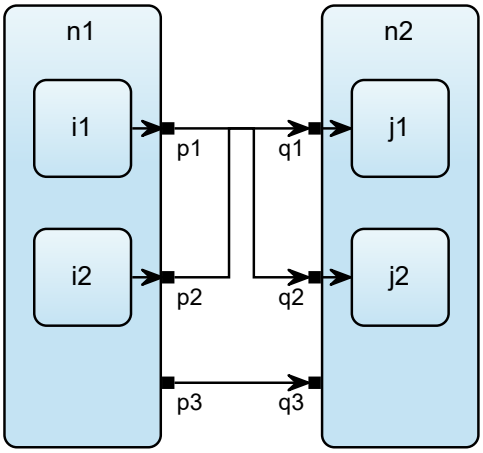
After having introduced the characteristics of reproducing the problem it is necessary to discuss the relevance of overlaps in general. One should note that every case just introduced relies on equal node heights making a positioning of one node's source port and another node's target port at the same

```
1   node n1 {
2      port p1
3      port p2
4      port p3
5
6      // nested graph
7      node i1
8      node i2
9      edge i1 -> p1
10     edge i2 -> p2
11  }
12  node n1 {
13     port q1
14     port q2
15     port q3
16
17     // nested graph
18     node j1
19     node j2
20     edge j1 -> q1
21     edge j2 -> q2
22  }
23
24  edge n1.p1 -> n2.q2
25  edge n1.p2 -> n2.q1
26  edge n1.p3 -> n2.q3
```

(a) Code for reproduction provided in ELKT.



(b) Corresponding hierarchy as tree. All parents root, n1, n2 include one graph.



(c) The layout with overlapping edge segments.

Figure 3.7. An example with fixed port positions caused by its hierarchical structure.

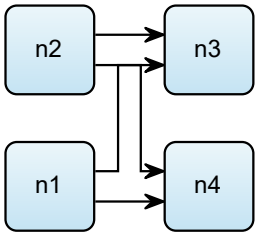
y-coordinate more likely, which is vital for an overlap to occur. This is a restriction that does not hold for every graph, but likely for an adequate number, since equal node sizes are indeed common. As for SCCharts it seems rather unlikely since the node sizes are often very different. However, even for different node sizes it is conceivable that the computation of port placements may set different ports at the same height and might thus cause overlaps. In any case, while trying out different graph layouts with equal node sizes I ran into those overlaps multiple times, even if not on purpose. For example I encountered the 4Nodes Case without intentionally seeking it. Hence finally I would definitely call this problem relevant, since it may indeed occur in common cases and further produces an overlap, which is rightfully rated as very severe violation of good layout characteristics.

3.3 Tasks and Challenges

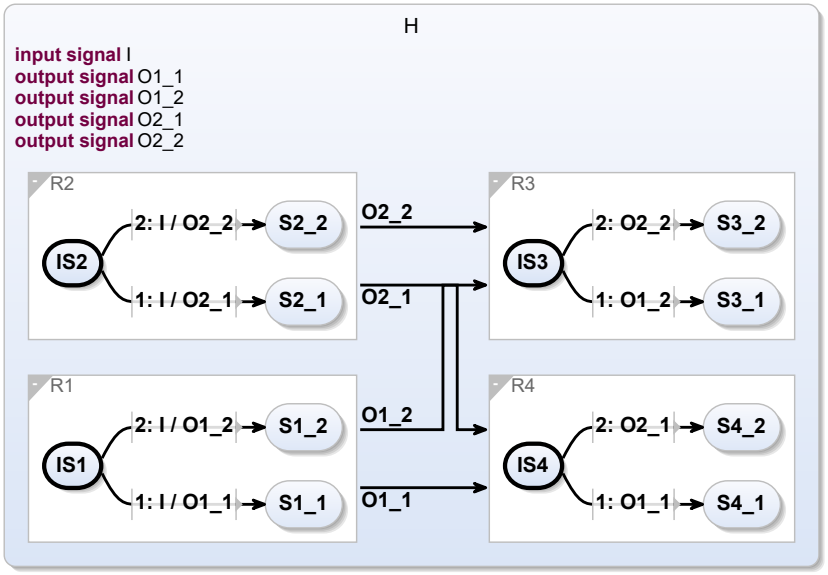
Giving edges the opportunity of using more than two vertical segments is a very serious breach to the edge router's assumption of every regular edge being able to be routed with at most three segments. I therefore have to prepare for a number of challenges and upcoming issues in the process. First, according to the conclusion of the relevance's discussion I value the problem as indeed relevant, but

```
1    node n1
2    node n2
3    node n3
4    node n4
5
6    edge n1 -> n3
7    edge n1 -> n4
8    edge n2 -> n3
9    edge n2 -> n4
```

(a) Code for reproduction provided in ELKT. There are no restrictions at all.



(b) Corresponding layout.



(c) A reproduction using induced data flow in SCCharts based on equally sized regions R1, R2, R3, and R4.

Figure 3.8. The *4Nodes Case* without explicitly defined ports.

mostly dependent on the criteria of an equal node size, which is not given for a moderate amount of graphs. Hence despite its relevance it is no common case. The problem's solution should consider this and before breaking the algorithm's assumptions of one vertical segment per edge it must be absolutely sure about its necessity. During implementation I try to stick to the *make the common case fast* paradigm, where *fast* here means to not increase the current runtime more than necessary. This is easy for most cases, since the detection of an actual overlap may be no big deal. The comparison of hypernode pairs during the creation of dependencies already checks for each pair of horizontal segments' distance falling below a specific threshold to detect conflicts. This check may be extended by an additional one for actually being zero in constant time. Furthermore a solution is only necessary if there is at least one pair of hypernodes for which holds that both partial layouts fail this new check.

This was the first naive approach before finding out about critical cycles introduced in Section 3.1.1, which must be looked for despite any occurrence of the more common overlaps coming with a two-cycle. Since finding cycles means additional computations for a common case, it may be wise to find a way to avoid this if possible, which I will discuss in Section 4.3.

In the following I will explain more tasks and a few initial approaches, where the most challenging topic is the question of what a general solution for overlaps should look like. How many segments do the algorithm actually need and where should it place them? But first I discuss the actual difference between overlaps and pure conflicts and how to classify them in theory.

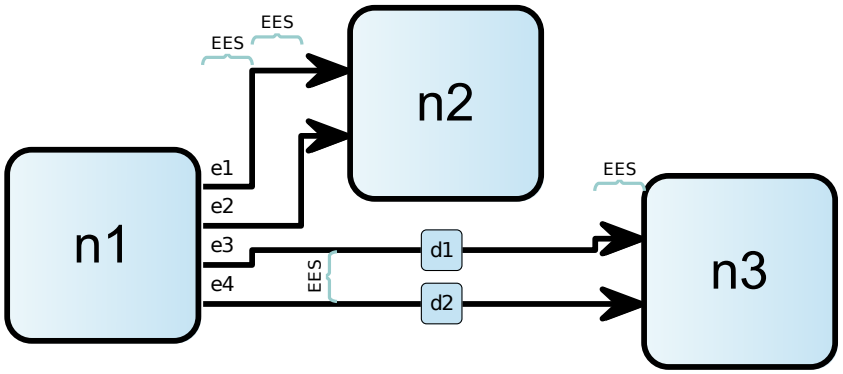
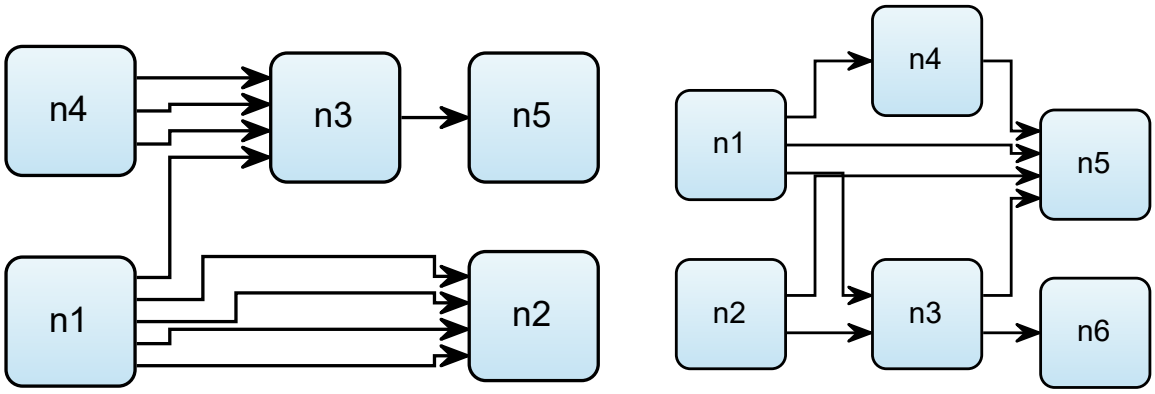


Figure 3.9. Orthogonal layout with occurrences of edge-to-edge spacing (EES) and explicitly rendered dummy nodes.

3.3.1 Thresholds and edge-to-edge spacing (EES)

I already mentioned the fact that the edge router does not distinguish between pure conflicts and overlaps. Instead it generally tries harder to avoid conflicts than crossings. First of all I explain this process in more detail and start by introducing a general value responsible for numerous characteristics of the final layout, the edge-to-edge spacing (EES). In Figure 3.9 I highlighted a number of its occurrences. Originally it was intended to be the minimal distance two edges should have at any point. Hence it is used as spacing between routing slots, so that vertical segments comply with it. Furthermore this means that the routing space between two layers always has a width of $(n_r + 1) * \text{EES}$ with n_r being the number of its routing slots. But the distances of all other edges' horizontal segments rely on their ports. As for the given example all four edges start at node n1. Hence n1 has four ports equally distributed over its size and hence the edges' horizontal source segments have the same distances. Those may fall below the EES in many cases. Thus the EES is impossible to adhere to for horizontal segments and they need an own threshold, the *conflict threshold*. It is an arbitrary value and for now defined as $0.2 * \text{EES}$. Further it is the reason why for instance e3 comes even closer to e2 than the already too close port distance, instead of adding a third routing slot. For this case one might consider this adequate, but it turns out to be too small in some cases. In Figure 3.10a is an example of two segments coming each other very close without detecting a conflict. Note that this is still a matter of perception and in the end a matter of proportion. This distance seems very small, but if n1 had for instance 16 ports with the same node size it may turn out to be absolutely fine, since all edges' distances would heavily decrease.

The conflict threshold is still bounded to the EES, which may lead to problems since it is modifiable via the layout options. If one for instance sets the EES relatively high, the difference between a pure conflict and an actual overlap would grow as well. Consider the comparison of two hypernodes v_1, v_2 in order to decide on a dependency. While the partial layout $v_1 < v_2$ may produce a conflict with a distance of 1, which is almost an overlap (see Figure 3.10b as contrast), $v_2 < v_1$ could produce a conflict with a distance of 5 being absolutely understandable in the final layout. But the algorithm is not able to distinguish them in their severity and adds a two-cycle which must be solved in the cycle breaking by deleting one of the two-cycle's dependencies. So it may be possible that the cycle breaking deletes the dependency containing the conflict with distance 5 and thus the final layout ends up with a pure conflict, which is nearly an overlap. Hence my first approach was to rank every conflict of size less or equal 1 as overlap. In Section 4.5 I will explain why I changed my mind on this while implementing the actual solution and instead introduced an additional *overlapping threshold* relying on



(a) The segments between nodes n1 and n2 come each other very close, but are not treated as conflict.

(b) A pure conflict of size 1 occurs.

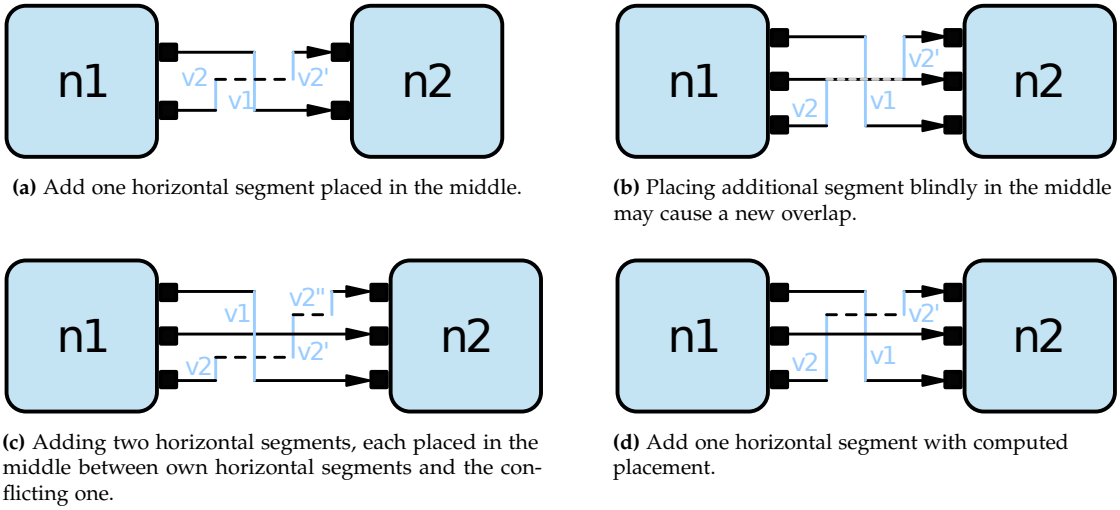
Figure 3.10. Two graphs with horizontal segments coming too close to each other.

the sizes between the ports.

Finally my conclusion is that using another value than the EES as notion of horizontal segments coming too close to each other is basically a good choice. It comes with the problem of finding a balance between EES, the actual space between ports, and a generally reasonable value to let the layout look good, though. During the implementation it is essential to distinguish between pure conflicts and overlaps and make their detection thresholds independent. Furthermore, since the conflict threshold would be no longer responsible for detecting overlaps I may set the threshold higher and decrease its severity in rating for a better partial layout between two hypernodes.

3.3.2 Additional Segments

In Chapter 1 I introduced a possible solution for the Simple Case, which one can see in Figure 3.11a with a visual distinction between the different segments. Here another routing slot was added, causing the routing space to grow by $1 * \text{EES}$ which I consider a small price for avoiding an overlap, but nevertheless it is a matter to mention. This new routing slot now includes a new vertical segment v2' bounded to v2. Such a new vertical segment is an "outsourced" part of the original one. Hence I call this division of one into two vertical segments a *split*. Those split parts must be linked together via an additional horizontal segment which I will refer to as *link segment*. It solves the overlap by giving the other edge's vertical segment a horizontal segment to convert it to a crossing instead. The first naive approach is to place this link segment directly in the middle between the corresponding source and target segment's height. This preserves the symmetry by maintaining equal distances between all horizontal segments, which can be desirable for a good layout. However, Figure 3.11b shows a problem: there may already be horizontal straight line edges which now cause a new overlap with the link segment. For solving this I considered two approaches. One is shown in Figure 3.11c and adds two new link segments on different heights and thus has three vertical segments in total. Another approach is given in Figure 3.11d which adds only one link segment, but with an adjusted position. An advantage of the first one is the still maintained symmetry, but one may easily recognize the disadvantages: the second additional vertical segment v2" has no purpose at all despite bringing symmetry. The same holds for the second link segment which reduces the distance between horizontal segments' for negligible benefits. Furthermore, on the new link segments' positions there could already be straight line edges as well. To solve this, further link segments would have to be added. Hence I



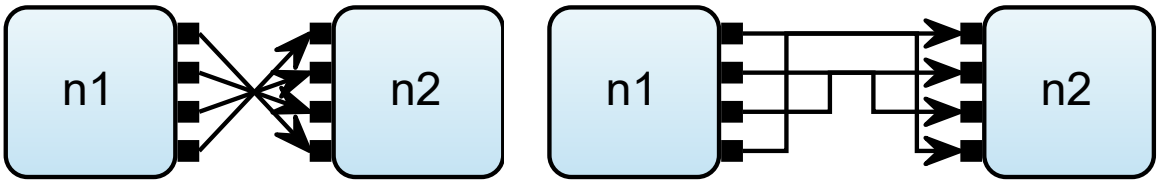
(a) Add one horizontal segment placed in the middle.

(b) Placing additional segment blindly in the middle may cause a new overlap.

(c) Adding two horizontal segments, each placed in the middle between own horizontal segments and the conflicting one.

(d) Add one horizontal segment with computed placement.

Figure 3.11. Different approaches of placing additional horizontal segments highlighted as dashed line and bounded to vertical segments v2, v2′ and v2″.



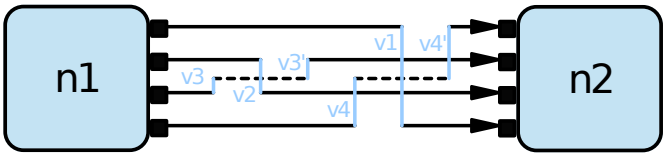
(a) With polyline routing.

(b) With orthogonal routing. Two overlaps occur.

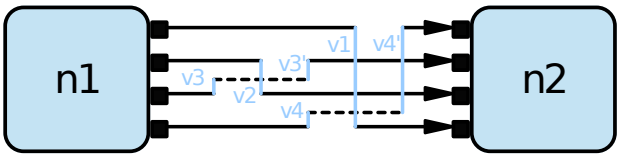
Figure 3.12. The *Nested Case*. The name derives on the fact that there is practically a Simple Case nested in another Simple Case.

decided for the second approach. Likewise this is the main reason I decided to use for every solution of overlaps exactly one link segment. In the following I will explain another one.

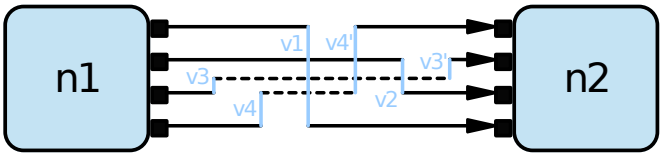
Before explaining I introduce the *Nested Case* in Figure 3.12 with two occurrences of overlapping. This time the algorithm has to add link segments on two edges which causes a new problem. In Figure 3.13a there is the first approach of a solution. Both link segments can be placed in the middle, because there is no straight line edge preventing this. They need space between them, though. If placing both link segments at the same y-coordinate they need more dependencies, since every horizontal segment has vertical segments on its ends shifting each other away. Hence there must be a dependency between v3′ and v4 to avoid a new overlap. However, in Figure 3.13b one can see a solution which places the link segments on different y-coordinates and makes it possible for v4 and v3′ to share a routing slot. This advantage holds not only for link segments. If possible the algorithm generally should not place a new link segment on any y-coordinate of other horizontal ones, but between them. This has to be mentioned, since it would be possible to do this, which is exemplary shown in Figure 3.13c. It is possible, because link segments are never bounded to ports and thus could be shifted to the left or right by any other horizontal dependency sharing the same y-coordinate, without to worry for any overlaps. This shifting causes new unavoidable dependencies to those horizontal segments, though. In the end no link placement should be placed on the y-coordinate of a horizontal segment, whether it is an original or another new link segment. This means for each link segment



(a) Using equal positions.



(b) Compute individual positions. Using only positions between original horizontal segments saves routing slots.

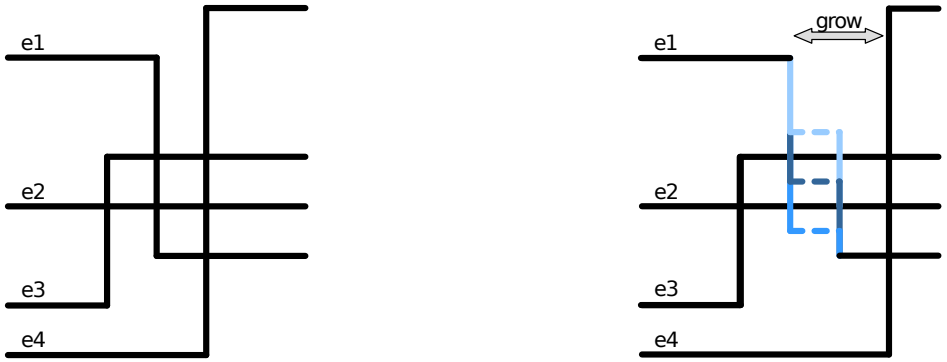


(c) Compute individual positions. Using horizontal segment's positions is possible, but causes new unavoiable dependencies to those.

Figure 3.13. Try to shorten the routing space of the Nested Case with individual placement for link segments highlighted as dashed lines.

to compute a very own individual placement and leads to the second reason to use only this one additional link segment: minimizing the computations for placement.

The algorithm already provides an adequate solution for placing and ordering vertical segments via its hypernodes. Hence I could use it to compute the arrangement of new vertical segments with already existing ones. Furthermore it is the reason I rejected the idea of improving the placements in a post processing step. But for the comparison of hypernodes to work properly it is paramount to have already computed the placements of the link segments, so the algorithm knows the exact shapes of the split vertical segments. This individual computation of placement comes with some convenient advantages, though. In most cases one does not have to care about additional crossings, except if there are hyperedges involved. First, consider the usage of only regular edges. Figure 3.14a shows a layout with six crossings and four edges in total. If splitting the edge e1 the assignment to routing slots cares for a good arrangement and there is no chance of getting additional crossings regardless of where the link segment is placed. In Figure 3.14b there are three examples of placements distinguished by different shades of blue. Each of these placements cause exactly the same number of crossings as before. This requires to add the new vertical segments before the hypernodes are compared or maybe comparing the new ones in a second run afterwards. Since the split vertical segments are partitions of the original one they bridge in total the same height-gap as before and thus both together cross at most the same number of horizontal segments as before. As for the link segment, since the routing space is growable, there is no chance for any crossing which could not be handled by applying new dependencies and a new routing slot, if necessary. Since the assumption is to split the nodes before actually computing the placements and to reorder all new introduced vertical segments, the only matter is the following:



(a) Layout without split edge. Six crossings occur.

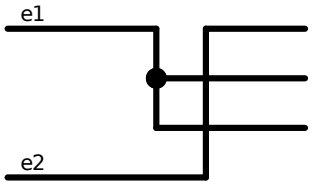
(b) Layout with split edge e1 for different placements of link segment hinted at by different shades of blue. There are still six crossings for each.

Figure 3.14. Relation between split regular edges and crossings with edges e1, e2, e3, and e4.

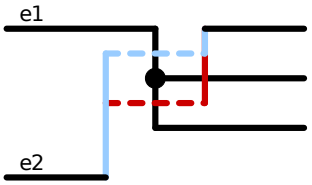
Assume there is an ordering for two hypernodes v_1 and v_2 considering only regular edges such that there are $i \in \mathbb{N}$ crossings. Is there an ordering for v_1', v_1'', and v_2, with v_1' and v_1'' being the split vertical segments of v_1, with at most i crossings? This holds, because the vertical segments v_1' and v_1'' are partitions of v_1. So every horizontal segment crossing v_1 can now either cross v_1' or v_1'', but not both. The original horizontal segments of v_1 are distributed to v_1' and v_1'' and crosses v_2 in total as often as before. The only new introduced horizontal segment is the link segment which would be only taken in consideration as potential crossing for the ordering $v_1' < v_2 < v_1''$. So there is in the end always an ordering with at most i crossings. Of course this excludes the crossing which is taken instead the overlap in the first place.

Unfortunately this does not apply if using hyperedges. For the setting of Figure 3.15a it is actually important where to set the link segment in order to avoid an unnecessary crossing. The edge e2 crosses the hyperedge e1 two times. As for Figure 3.15b there are two alternatives for placing. One is highlighted in red and results still in two crossings, while the other one is blue and only causes one crossing, which is even better than the original solution. The algorithm should of course be able to decide for the better alternative, since it is another characteristic for a good layout to minimize crossings if possible.

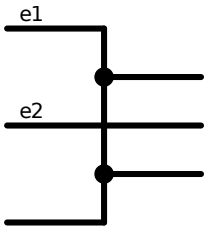
Another case considers splitting a hyperedge. For regular edges it holds that the split results in a partition, but since a hyperedge contains more than two horizontal segments, it may have a form like the one in Figure 3.15c. A split would require to pass all target segments to the new introduced vertical segment which must grow in the process. It results a form shown in Figure 3.15d with two crossings instead of only one. To solve this I could pass target and source segments to the new hypernode as long as those lie in its reach which looks like as shown in Figure 3.15e. But I consider the resulting form as rather strange and prefer to stick to the notion proposed by Sander of forking the hyperedge only right before it reaches its target or source ports [San04]. Hence I decided to avoid to split hyperedges, if there is an alternative regular edge to do it for instead. This way those cases may be at least minimized.



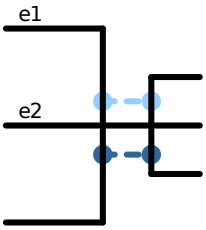
(a) Layout before splitting an edge. Two crossings occur in order to avoid the overlap.



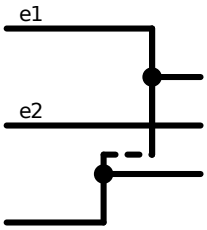
(b) Layout after splitting e2 for different placements of link segment. The placing is relevant to avoid the second crossing.



(c) Layout before splitting the hyperedge. One crossing occurs.



(d) Layout after splitting the hyperedge. There are now two crossings, regardless of the link segment's placement.



(e) Alternative splitting for hyperedges. There is now still one crossing.

Figure 3.15. Relation between hyperedges and crossings with hyperedge e1 and regular edge e2.

Chapter 4

Implementation

In Chapter 3 I described a number of approaches I try to stick to. As mentioned before I try to always adhere to the *make the common case fast* approach. Hence I try to not increase the actual runtime for a common layout without visible overlaps too much, while trying to let the algorithm cope with them as well. In the following I will describe the actual realization of the solution in its steps. Furthermore many new functionalities involve a newly introduced overlapping threshold as well as the updated conflict threshold. What those thresholds actually look like will be discussed in Section 4.5—for now it is only important to note that there are two adjustable values. Those represent the distance two horizontal segments *should* have (conflict threshold) as well as the one two segments *must* have (overlapping threshold). Of course the latter is smaller. The reason to introduce their actual computation at a later time is that this relies on a number of characteristics I want to introduce before.

4.1 General Structure

Before explaining the most important steps in detail, the initial question should be how and where to fit a detection of and solution to overlapping in the current algorithm. I sketched the new arrangement of functionality in Figure 4.1. The five nodes on the first level stand for the current orthogonal edge routing algorithm. The most interesting parts of the new features happen before breaking the cycles in the dependency graph. As mentioned in Chapter 3 the detection of overlaps caused by a two-cycle happens during the comparison of hypernodes, while the critical cycles must be checked extra afterwards. Finally the algorithm solves all overlaps by adding more horizontal and vertical segments. More vertical segments means updating the arrangement of hypernodes with new dependencies. Afterwards the algorithm resumes as usual, except for a few simple changes to the final placement's computation in order to be conform with the new structure.

The requirement for an individual computation of the horizontal segment's placement comes with another advantage. Since the algorithm must consider a number of characteristics anyway, such as additional crossings and routing slots, it assumes to always find an adequate solution in the end.

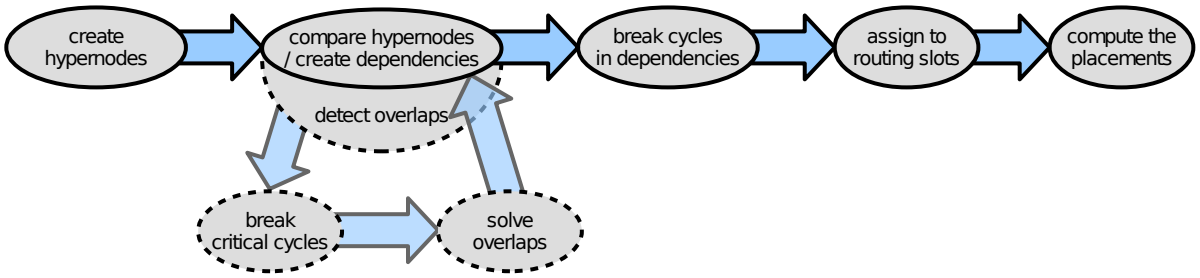


Figure 4.1. Integrating the new functionality in the given algorithm. The top level nodes represent the current algorithm whereas the dashed nodes are new features.

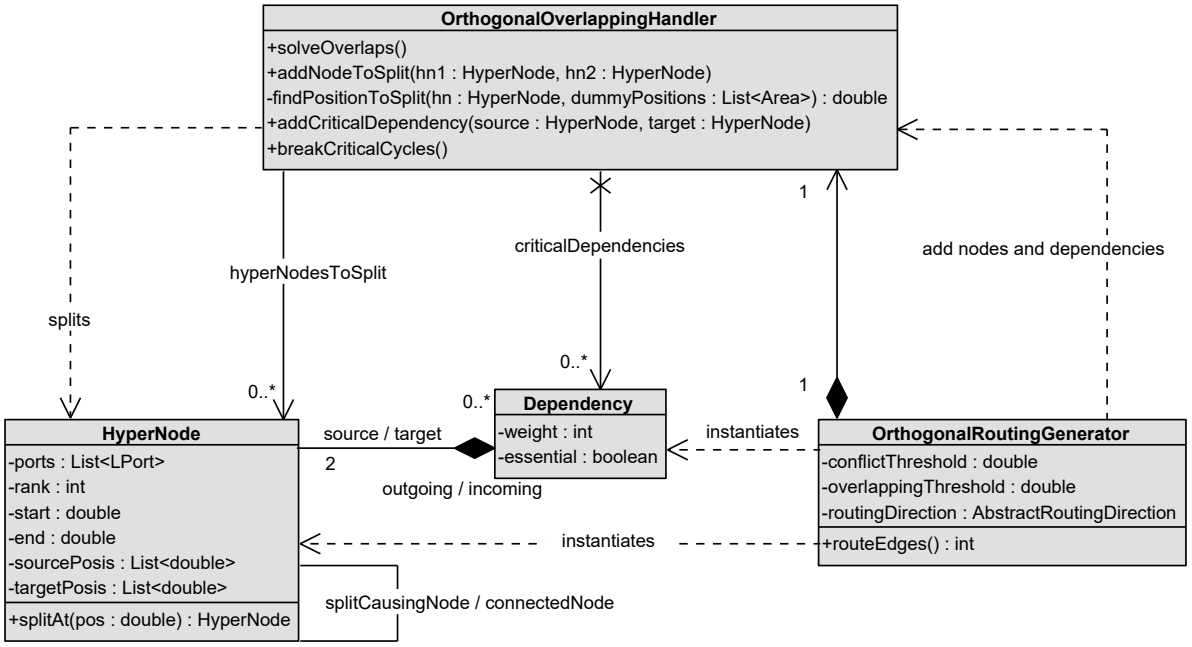


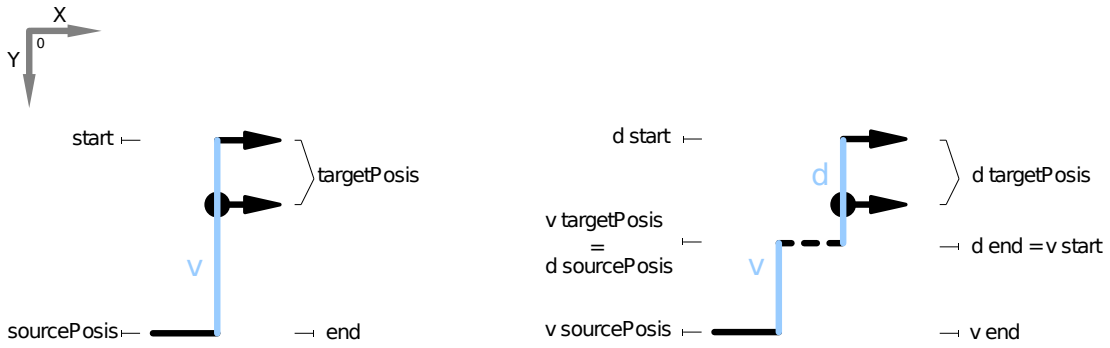
Figure 4.2. Simplified class diagram showing the most important components and their relations.

Hence the algorithm runs the solving part exactly once for every suffering hypernode. Consider for instance again the approach of Figure 3.11c (see page 30). Such a solution could start with the first approach of Figure 3.11a setting the link segment blindly in the middle and splitting again if the solution is not adequate. This may be faster in some cases, but in regards to all the matters discussed during Section 3.3.2, it could end up in a bad layout if not considering potential advantages of every other possible location to place the link segment.

The general class structure of the new edge router's version is shown in Figure 4.2. Most of the new features belong to the so-called `OrthogonalOverlappingHandler` (henceforth called *handler*). It is held by the `OrthogonalRoutingGenerator` which is supposed to lay out a pair of adjacent layers and implements the algorithm described in Section 2.3. While the handler cares for the actual solution, it still falls to the general algorithm to detect cases of overlaps. The solution consists of two vital steps: First, the algorithm computes an adequate position for the link segment and second, it splits the edge at exactly this position and rearranges all vertical segments. In the following I will explain the details of this solution and its detection.

4.2 Splitting of Hypernodes

As described in Section 2.3.2 the hypernode is the edge router's representation of an edge. In Figure 4.3a one can see the actual values the hypernode stores for this matter. First of all there are *start* and *end* positions indicating the vertical segment's reach. Further it has two lists of sorted positions: `sourcePosis` storing the source segments' positions and `targetPosis` doing this for the target segments. Those must be lists in order to support hyperedges with multiple source or target segments. Note that position only refers to the vertical dimension, since the horizontal position is not computed until the end. Each of those positions is vital for the detection of crossings and conflicts during the comparison



(a) Hypernode v before split.

(b) Hypernode v and its *dummy* d after split.

Figure 4.3. Description of hypernode's values, with `sourcePosis` and `targetPosis` being the list of source and target segments' positions.

of two hypernodes. One may see that the start value is the minimal horizontal segment's position and the end value the maximal one.

In order to solve an overlap the algorithm should be able to split the edges, which means in other words splitting the hypernodes. In the following I will explain how I extended the hypernode to cope with the new functionality. First of all, when introducing the hypernode I mentioned that it stands especially for the vertical segment. Hence understanding it as a vertical segment is obvious and represents the general algorithm's notion of at most one vertical segment per edge. Except for the overlapping issues discussed in this work, this notion is absolutely fine. Thus, instead of breaking it, I decided to solve the overlaps with a workaround. My method to split a hypernode is shown in Figure 4.3b. Basically the splitting creates another hypernode, the *dummy hypernode* (or *dummy*) d which is bounded by the link segment to the original one, the *split hypernode*. For a hypernode, splitting means handing over all target segment's coordinates to the dummy while keeping those of the source segments. In order to actually link both hypernodes the new link segment's position becomes v's new `targetPosis` as well as d's `sourcePosis`, which will of course always be of size 1, since there is only one link segment. As for the given example the dummy's start value is the original hypernode's start value and its end value is the position of the link segment. Note that this is not always the case when using hyperedges. Since the dummy gets all of the target segment's positions there may be cases of some being below the link segment causing the vertical segment to grow. Simultaneously some original hypernode's source segments may be above its start value. This is the reason the vertical segment's shape must always be recalculated after the split, instead of simply applying the old start and end values together with the position of the new link segment. An example was already shown and explained for Figure 3.15d (see page 33).

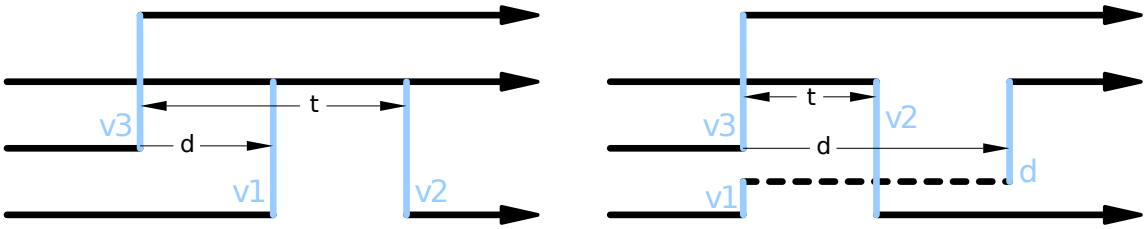
4.2.1 Rearrangements

As mentioned, the two elementary detections of overlaps take place while and after comparing hypernodes for creating dependencies. At this point it is necessary to clarify again that the algorithm does not modify a given layout, as it may seem if looking at those *before and after* comparisons provided as visual examples in this work. In fact the algorithm modifies the internal graph structure this layout relies on, thus causing a different arrangement of routing slots and a new layout. Hence when running the split phase, dependencies have already been calculated between all current hypernodes. The very first approach was to delete those, adding the new hypernodes, and starting from scratch to recalculate

(a) Two-cycle t caused by the overlaps.

(b) Adding three new *essential* dependencies d.

Figure 4.4. Applying new dependencies.



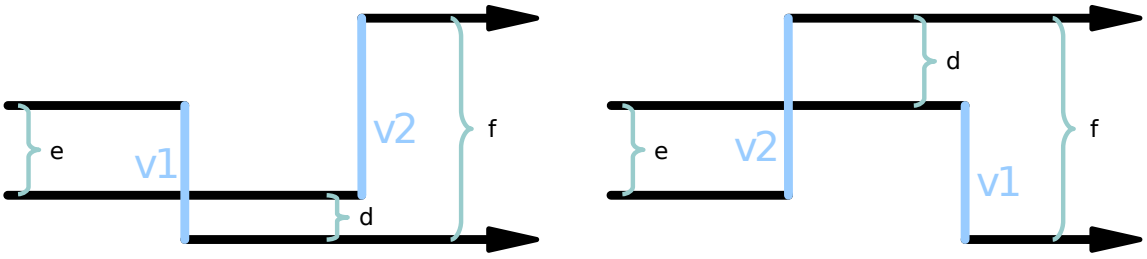
(a) Without splitting, there is a dependency between v3 and v1.

(b) With splitting, the dependency (v3, v1) is no longer necessary, since the link segment was placed providently. Instead there is need for a dependency (v3, d).

Figure 4.5. Updating dependencies d and two-cycle t after splitting v1.

them all. But actually extending the amount of vertical segments and incorporate those in the current ones is generally no problem, since this only means adding new nodes and edges to the internal dependency graph it relies on. Besides, the actual computation of the total number of routing slots and the assignment of hypernodes to those slots happens at a later time anyway. The actual extension relies now on adding new segments as well as updating old ones. Regarding the former, there is a process of splitting described in Figure 4.4. Every overlap, whether detected as two-cycle or in order to break a critical cycle, is caused by exactly two hypernodes. This is always the case, since in one routing space there could be at most two ports at the same height. As for Figure 4.4a it is caused by a two-cycle between v1 and v2. The decision of which is to split is made by the handler. In this case it is v1, while v2 is referenced as v1's *split causing* hypernode. Remembering this during detection is important in order to later add three *essential* dependencies caring for the partial layouts $v1 < v2$ and $v2 < d$, which is out of the question in order to create the distinctive order shown in Figure 4.4b. The third one, $v1 < d$, seems to be redundant due to the transitivity of the other two, but reassures the structure additionally to an extraordinarily high weight for every essential dependency for being broken by the cycle breaking.

As discussed during Section 3.3.2 the link segment is placed at an individually computed position in order to minimize crossings and routing slots. For those minimizations to take effect all vertical segments must be rearranged. This means that after adding new dependencies, the algorithm has to update the original ones. The example in Figure 4.5a shows a hypernode v3 with a dependency to the hypernode v1 going to be split, as well as to the other hypernode v2. After the split in Figure 4.5b v3 no longer needs a dependency to the node v1, but instead one to the newly introduced dummy hypernode. So the dependency (v3, v1) must be deleted. After splitting it must be decided for each split hypernode's dependency whether it must still be applied to it, just to its dummy hypernode, or to both. During this update there is now a new check for horizontal segments coming each other too close. However, this time it depends on the new introduced overlapping threshold, since the new distances between current horizontal segments and the new link segment placed between them may



(a) For the partial layout v1 < v2.

(b) For the partial layout v2 < v1.

Figure 4.6. Checking d for being a conflict. However, the distances e and f are equal for both orderings and thus do not care the arrangement.

fall below the conflict threshold anyway. The two-cycle between v3 and v2 is maintained, because no dependency is manipulated as long as its source or target is *not explicitly involved* in a split. "Explicitly involved" means being actually part of the split. One may note that the two-cycle of Figure 4.4a is gone for Figure 4.4b. Note that this was not deleted, but never added to the dependency graph when using the handler, which will be explained as part of the following section's discussion.

4.3 Problem Detection

In this section I will explain the detection of overlaps. As aforementioned the algorithm needs in fact two detections. The first one takes place directly during the comparison of hypernodes and finds overlaps caused for both alternative orderings of the two compared hypernodes. The second one must look up specific cycles in the internal graph, after all original hypernodes are compared and created their dependencies. Such a cycle must contain only those dependencies avoiding an overlap. I will begin my explanations with the former one.

As already hinted at during Chapter 3 the first detection should be relatively easy which turns out to be true during implementation. The original comparison of two hypernodes v_1, v_2 computes for each partial layout $v_1 < v_2$ and $v_2 < v_1$ a value which represents the layout's badness. Such a value is the sum of crossings and conflicts caused if choosing this partial layout, with the latter multiplied by an arbitrary constant. The constant is intended to put more weight on conflicts, since for the original algorithm there is the chance of conflicts being an overlap. This detection of conflicts is of particular interest for finding overlaps as well. For now the algorithm checks for $v_1 < v_2$ whether v_1's target segments come closer to v_2's source segments than the conflict threshold which I introduced in Section 3.3.1. Why the comparison checks exactly those values to find conflicts is demonstrated in Figure 4.6. The difference d is the only one of interest, because e and f remain the same for every arrangement. Hence I extended the check of d for being smaller than the conflict threshold by another one checking the same with the overlapping threshold, a value which will be explained later on.

How I fit this detection into the original comparison is shown as pseudocode in Figure 4.7. As usual, dependencies are not applicable to a straight line, because an arrangement has no effect on it. The comparison begins in lines 7 and 8 with counting every pure conflict and detecting overlaps in the aforementioned manner. If both alternatives `v1 < v2` and `v2 < v1` failed the check for overlaps the detection is triggered and the algorithm passes both hypernodes to the handler. The decision which one is actually split is up to the handler. After deciding on a node I will refer to it as *registered* for being split during the actual phase of solving overlaps. Note that there is no dependency added in this case, which is the reason there is no two-cycle in Figure 4.4b. If no direct overlap is detected, there

```
1   void createDependency(v1, v2) {
2      if (v1 or v2 is straight line)
3         return;
4
5      // count number of pure conflicts and overlaps for both variants
6      con1 = count number of pure conflicts and overlaps of v1 < v2;
7      con2 = count number of pure conflicts and overlaps of v2 < v1;
8
9      // input for the handler
10     if (con1 includes overlap && con2 includes overlap)
11        // register one hypernode for splitting
12        store v1 or v2 in handler;
13     else if (con1 includes overlap)
14        // reject v1 < v2
15        create critical dependency (v2, v1) and store it in handler;
16     else if (con2 includes overlap)
17        // reject v2 < v1
18        create critical dependency (v1, v2) and store it in handler;
19     else
20        resume as usual;
21  }
```

Figure 4.7. Pseudo code for detection of overlaps during the comparison of hypernodes `v1`, `v2` in order to create a dependency.

are additional checks regarding the second detection of critical cycles.

Despite the fact that critical cycles are a very rare phenomenon they must be precluded for every graph. Since finding a cycle can be generally tedious work, it would be wise to find other ways of confirming their absence without checking every possible path in the dependency graph. First of all, such a cycle consists of only *critical* dependencies. Those are dependencies avoiding an overlap. This is why the algorithm would for instance add in line 16 such a critical dependency from `v2` to `v1` if `v1` $<$ `v2` causes an overlap. After creation it is stored directly by the handler which cares for a potential cycle breaking once the comparison is done. In Section 3.2 I explained that every overlap, whether it is avoided by a dependency or not, relies on the fact that a source port is placed at the same y-coordinate as another edge's target port. It may happen in some cases, but it is rather unlikely for a real life example. This implies that in many cases we could avoid the cycle detection anyway, because there are less than three critical dependencies. And even if there were more, it may be a moderate number in most cases.

Upon finding a critical cycle the algorithm simply chooses one of its hypernodes, deletes its critical dependencies, and registers it for splitting. Afterwards the process is repeated until no cycle is detected any more. Despite that there is no need to care for the dependencies' weights, since I let every critical dependency be weighted equally, I admit that for the current state there is still room for enhancements. The algorithm could use for instance a minimal Feedback Vertex Set (FVS) to split the fewest possible number of hypernodes in order to break the cycle. The FVS is a set of nodes breaking the cycle on deletion, which relies on an NP-complete problem [GJC79] and corresponds to the FS described in Section 2.2.1. But since this issue is so rare I decided to reduce the invested effort and to concentrate

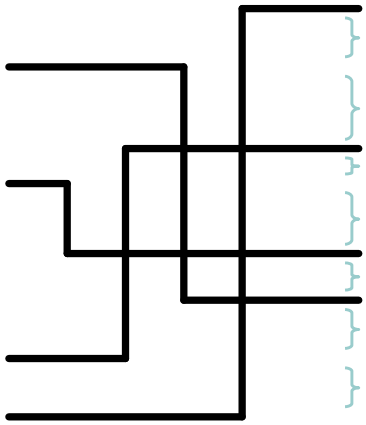


Figure 4.8. Detecting potential areas to place link segments in.

on more common problems.

4.4 Solve Overlaps

After both phases of detection finished and registered a number of hypernodes for being split, the actual process of finding a solution begins. Such a solution consists of a number of tasks. The approach starts with finding possible locations to place link segments at and continues by deciding for each registered hypernode which position may be the best one for its split. Afterwards the hypernode gets split and its dependencies experience an update before proceeding with the next one. In the following I will explain each of these tasks and hint at the notions they rely on.

The very first thing to do is to find possible locations for a link segment to being placed at. As discussed in Section 3.3.2 the link segments should not be placed at the same y-coordinate as other horizontal segments to avoid unnecessary dependencies. So the handler looks for spaces between all current horizontal segments high enough to place a new link segment in. A space is high enough as long as the overlapping threshold fits into it two times, because this way a newly introduced segment does not have to worry about any conflict. This relies on the fact that new calculated dependencies uses this new threshold for the conflict detection. Hence, in these detected spaces it is possible to place the link segments without creating new dependencies in the process, except for those created by the update mentioned in Section 4.2.1. For more compatibility the handler provides the spaces as one dimensional areas, an example is given in Figure 4.8. To avoid that multiple link segments are placed at the same y-coordinate, every area is consumed by using it for a split. This way every area is used only once. Since hypernodes of smaller size have fewer possibilities of placing their link segment, the algorithm sorts all registered hypernodes by their vertical segment's size before the placement. Consider for example again the example Figure 3.13 (see page 31). If the algorithm placed the link segment of the bigger hypernode in the middle, the smaller one would have no possible area left. Note that the overlapping threshold's flexibility reassures that there are always areas being considered high enough, regardless of the placement of ports. This characteristic will be further explained in Section 4.5.

4.4.1 Placing the Link Segments

After computing a number of possible areas to place the segments in, the solution must assign those. For this matter it now works through the sorted list of registered hypernodes and decides on a position for each. So for a registered hypernode the handler starts to select all areas which are actually in its vertical segment's reach. For all those it creates a rating. Relying on Section 3.3.2 I assume that there could be no new dependency for the split hypernode or its dummy, except those regarding the mentioned update. This is the reason the handler takes only the current dependencies into consideration when rating the specific position. Such a rating checks for potential new crossings and dependencies a splitting would cause on this position. Relying on this rating the handler decides for a position. The rating depends on the following criteria sorted by their relevance, with every criterion only taken in consideration, if the previous one is equal for two compared positions.

1. The minimal number of caused crossings.
   This is the prime directive. Regarding the introduced notions for good layouts presented in Section 2.1.2 it is necessary to avoid as much crossings as possible. Since in order to avoid overlaps there are already additional segments and bendpoints anyway, the crossing minimization is the only basic characteristic of a good layout this rating may actually influence. Differences in the number of crossings should be only the case for hyperedges involved in the overlap, as I discussed in Section 3.3.2.

2. The minimal number of caused dependencies.
   This way the handler tries to minimize new dependencies in order to shorten the resulting routing space. For many graphs the avoidance of stretched layouts is a common characteristic as well. Since the layered approach relies on a notion of direction laying out nodes subsequentially, these kinds of layouts tend to be rather wide anyway. This is the reason to not list this as a common criterion in Section 2.1.2, but still it is a matter to avoid unnecessary wide layouts if possible.

3. The area's size.
   Since in the end placing a link segment means halving the original distance between two horizontal segments, it is indeed a wise decision to choose the area of biggest size.

4. The minimal distance to the middle between the registered hypernode's source and target segment.
   If nothing else matters, a link segment should be placed as near to the middle as possible, in order to provide symmetry.

Note that the handler does not have to worry about any conflict for the rating, since it computed areas with adequate distances previously.

After deciding that the hypernode should be split, the handler adds the essential dependencies and distributes the original ones over the split hypernode and its dummy. The area is now consumed and the handler proceeds with the next registered hypernode.

Unfortunately this whole process suffers from one general problem. The rating relies on all vertical segments' shapes, while simultaneously those rely on the result of the rating, at least for all registered hypernodes. This means that dependencies to other hypernodes registered for a split which is not performed yet may be inaccurate in some cases, since their vertical segment's shape is not the final one. However, this only affects the enhancements of saving routing slots, and even this only in a little number of cases, hence I value this issue for now as tolerable. It seems to be a very rare case of having multiple unavoidable overlaps in a graph anyway.

4.5 Adjustments to Thresholds

The last and most challenging topic is the discussion of reasonable values for the new overlapping threshold which was mentioned a few times by now. Its purpose is the strict differentiation between overlaps and pure conflicts. I also hinted at the reason to use a threshold instead of simply checking the segments for having actually no distance at all. The example shown in Figure 3.10b (see page 29) shows a pure conflict relying on a distance of 1, hence the segments touch and seem to overlap as well. The first naive approach was to use a constant overlapping threshold of 1, so every distance less or equal is interpreted as an overlap. In the following I will explain why I changed my mind on that during implementation and how I adapt the conflict threshold in the process. The actual computation will be explained at last.

ELK comes with a high amount of variety for different options and placement restrictions which could be applied to a graph. The layered algorithm itself adds even more. For example it distributes all defined ports equally over a node's source and target's sides, but one can define exact positions as well. Every solution must consider this as well as every other optional characteristic which makes it hard to adhere to everything. As for the ports' placements, there are cases of port distances falling below any reasonable value. This could be caused by an equal distribution of far too many ports. If the port distance falls for instance below 1, a constant overlapping threshold would detect overlaps almost everywhere. Why this would be bad was already explained in Section 3.3.1 for the conflict threshold. After all, the whole notion of finding better partial layouts and recommending those via the dependencies relies on a balance, which could be disturbed by adding high-valued dependencies for almost everything. The reason is the cycle breaking solving an NP-hard problem in a good, but indeed not perfect way. An increasing number of cycles makes inconvenient decisions of the cycle breaking more likely. Hence, one approach could be to deny the solution of overlaps for those unreasonable distances, because a solution may end up in a layout which is still confusing. But since the whole layout process is bounded to predefined node sizes and thus on predefined distances between horizontal segments, my opinion is that the algorithm should be able to handle even those cases adequately. Thus I decided for a variable overlapping threshold which could fall below any reasonable value as well, since in the end two touching edges are still slightly more comprehensible than two overlapping ones. Chapter 5 will provide a number of examples for very low port distances.

So the overlapping threshold depends not on the EES, but on the distances of ports. Further it takes only those ports in consideration that have actually an edge linked to it, whose positions in other words correspond to all original horizontal segments involved in the routing. This comes with an inconvenience: For now only the hypernodes know their horizontal segments, but there is no general overview of all of them. Such an overview is vital to compute for example the areas for placing the link segments. Unfortunately it requires to iterate over all hypernodes, store every horizontal segment's position in a list, and sort it afterwards which means additional runtime. This is why an earlier version of my algorithm performed this computation only to find the areas, if actually solving an overlap and spared the common case. But since introducing the new threshold I must extend this feature for the common case as well, in order to properly detect the overlaps. Now there are generally two ordered lists of source and target segments which can be used to compute all distances. Their minimum is in the end the intended port distance to use. Only now the algorithm can use it to compute the overlapping threshold before the comparison of hypernodes. Nevertheless I value this as necessary and accept the additional runtime in the common case for now as unavoidable.

There are two other characteristics necessary to mention when dealing with port distances. First, for a graph with high spaces between the horizontal segments, the minimal distance and thus the overlapping threshold turns out to be relatively big, since I consider the distance between all ports,

not only those of the same node. Second, the conflict threshold is computed for every routing process between two adjacent layers. Hence, those may differ for the routing spaces in a graph. The former comes with the requirement to limit the threshold to a maximum which I define as 2. This value stems from the fact the original algorithm tried to avoid overlaps with a threshold of $0.2 * \text{EES}$ with the latter being 10 unless defined otherwise. So I figured out $0.2 * 10 = 2$ as an adequate maximum. Simultaneously, this maximum may not solve the differing thresholds per routing space, but makes them hard to notice, since an overlapping threshold t is now always a value with $0 < t \leqslant 2$.

As for the conflict threshold, since its purpose of avoiding conflicts is now decoupled from considering overlaps, there are a few characteristics I adapted. First of all, it is generally increased. This was avoided in the past, since a conflict was always valued as potential overlap and thus as very severe. Bringing us to the next point, it is not severe any more. Overlaps are avoided elsewhere and in the end a pure conflict only indicates two edges coming a bit closer than they should be. Instead now I value a crossing as more severe, because a basic notion of good layouts is to avoid those, not to maintain a specific distance between edges. Anyone who might disagree with that may simply adapt the constant every crossing is now multiplied with. Another topic is a reconsideration of deriving the conflict threshold from the EES. One approach was to use the minimal distance, since the algorithm computes it anyway in order to set the overlapping threshold. But in the end I abandoned this idea, because of the deviations between the routing slots I already discussed for the overlapping threshold. For this threshold they would be much worse, since the constant to multiply the distance with is bigger and has no maximum. Instead I decided to leave it as it is, computed by using the EES, and just to adapt the constant.

Finally, after explaining the minimal difference between horizontal segments, I introduce the actual computation of both thresholds. This explanation is accompanied by an exemplary setting in Figure 4.9. This example shows a layout rendered with the old algorithm, but with appended hints for all new and generally relevant notions of distance. First of all, both thresholds rely on their own constant. The conflict threshold's computation remains the same, but I modified the constant from 0.2 to 0.5 in order to shorten the accepted distances, which were in my opinion too small in some cases. Note that this constant is arbitrary and found experimentally. It may be adapted in the future if it turns out to be insufficient. Since differentiating it from overlaps it benefits from much more freedom and less relevance, though. In the example there are actually two cases of too small spaces visible. To show that the new conflict threshold (CT) would solve this, if using the updated algorithm, I appended it to those cases. Note that the conflict threshold remains the same for both routing spaces. However, the overlapping threshold is computed with $0.2 * m$ with m being the aforementioned minimum distance between source or target segments. In the example those are m1 and m2 computing the overlapping thresholds OT1 and OT2 for each routing space. To demonstrate the slight differences between both overlapping thresholds they are contrasted with each other below the general computations in the example. Furthermore the computations are given in the example as well, together with markers representing their actual sizes.

The aforementioned constant for computing the overlapping threshold is for now defined as 0.2. As final topic of this chapter I want to explain why. Contrary to the arbitrary constant for pure conflicts, there are actual reasons this value *must* remain lower than 0.5 and *should* be lower than 0.25. Since those reasons are much more comprehensible with an example, there is a hypothetical setting given in Figure 4.10. The edges are hidden, but the vertical positions of their horizontal segments are hinted at as lines. Those connected to n1, the source segments, are dashed lines, while the target segments, connected to n2, are solid ones. The algorithm looks up the minimal distances between all source segments m and all target segments n which are equal in this case. Hence, the minimal distance to compute the overlapping threshold is $m = n$. Assume now a constant of 0.5 halving this minimal

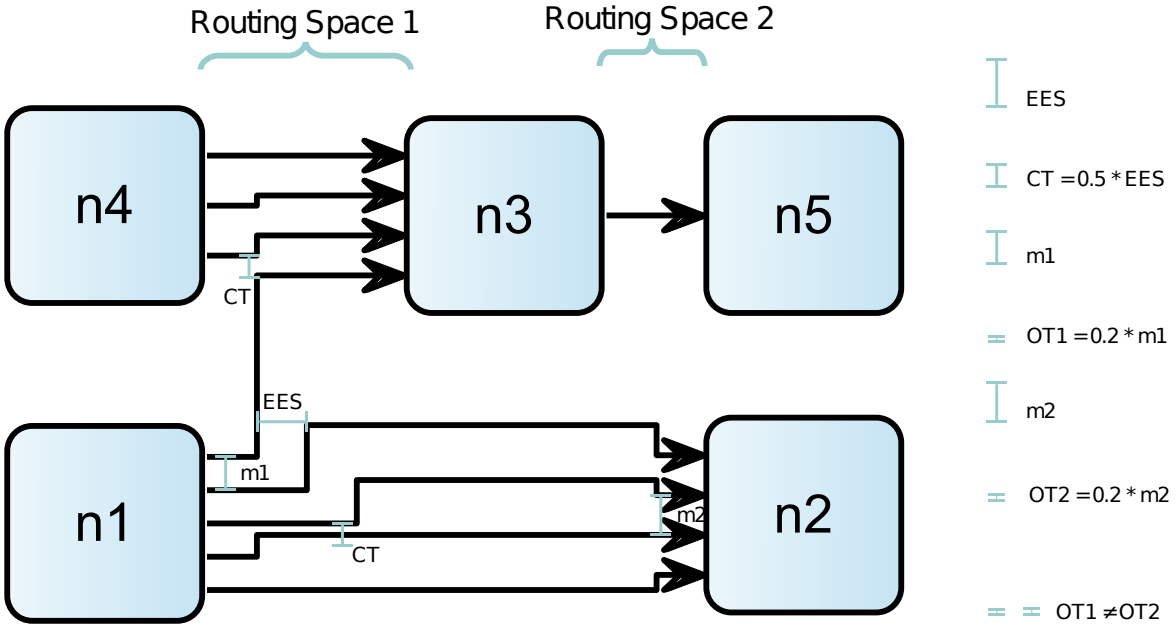


Figure 4.9. Layout rendered with the old algorithm and showing specific calculation for the two thresholds. Overlapping threshold (OTi) is computed with the minimal difference between horizontal segments mi for routing space i with $i \in \{1, 2\}$. The conflict threshold (CT) is computed using the EES.

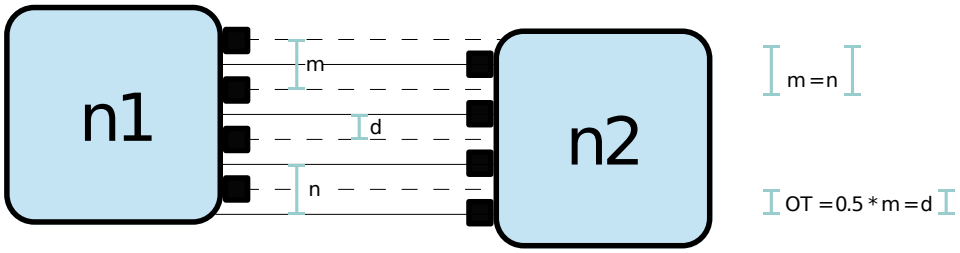


Figure 4.10. Setting for error-prone configuration of overlapping threshold when using a constant $\geqslant 0.5$. Source segments' vertical positions are hinted at by dashed lines, target segments' positions by solid ones, with m = n being their vertical distances used to compute the overlapping threshold (OT). The maximum space between horizontal segments is named d.

distance to use it as actual overlapping threshold. Since the port positions are determined such that every area between a source and a target segment is half the minimal distance, there may be cases of found overlaps. For example if there are applied edges similar to the Nested Case (see Figure 3.12 on page 30). After detecting those overlaps, the handler searches for areas to place the link segments. As described in Section 4.4 it uses the overlapping threshold to validate all areas with this fitting in twice, in order to let a segment, which is placed in it, adhere to this value. But all areas have the same size of $d = m * 0.5 = t_o$ with t_o being the overlapping threshold. The handler would find no area to place the segments in. To actually fit in those areas of size d, it must hold that $t_o < \frac{d}{2} = 0.25 * m$. However, for a threshold constant below 0.5 the algorithm should find no overlaps in this example. This relies on the fact that it picks the minimum of distances and all source ports have vertical distances to all target ports of at least $d = 0.5 * m > c * m = t_o$ with c being a constant with $0.25 \leqslant c < 0.5$. So still there is no chance of finding an area, but probably no chance for causing overlaps as well. To be honest, I am not absolutely sure if the latter discussed case with this constant c could actually happen with any setting or additional option applied to the algorithm. Hence I decided to be on the safe side and thus let it be below 0.25. Besides, 0.2 seems to me like an adequate value, since in most cases it leads the threshold to vary between 1 and 2. The former value was my initial intended threshold while the latter represents the default conflict value of the original algorithm used to avoid overlaps.

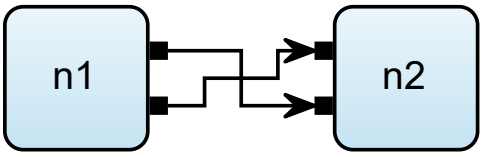
Chapter 5

Evaluation

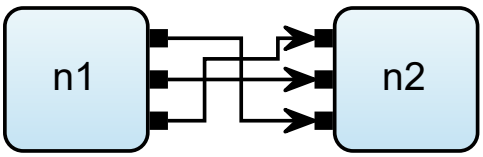
This chapter is meant to analyze the algorithm proposed in this thesis and to compare it with the old one. For this matter I will introduce a number of new examples as well as a comparison of layouts produced by both. Furthermore I will check whether I accomplished the actual goal of avoiding overlaps. I start with the solutions to a few problems I introduced during my explanations and theoretical assumptions in the chapters before.

5.1 General Examples

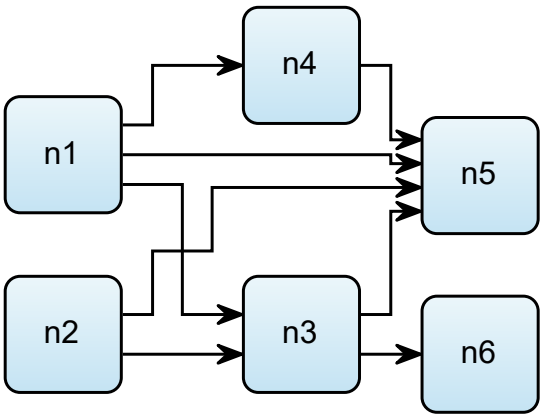
The avoidance and detection of the general problem represented by the Simple Case in Figure 5.1a is actually not much additional effort. It is detected during the comparison of hypernodes. Since in this case there are only two of them, the whole comparison of hypernodes cares for only one pair. As described in Section 4.3 the detection leads to no additional dependency, since an overlap is found. Hence the solution resumes with splitting one of the hypernodes with only one possible area to place the segment in and no dependencies to update. The resulting structure relies on the essential dependencies added by the solution, which are described in Section 4.2.1. As for the graph in Figure 5.1b, there are actually three hypernodes since one stands for the straight-lined edge. But such straight lines do not create dependencies and thus the only difference to the Simple Case is the number of possible areas to use. For this case it does not matter which area is chosen, though. In Figure 5.1c there is the case of a pure conflict of size 1, introduced in Figure 3.10b (see page 29). Edges with a distance of this size seem to actually overlap, which is the reason I value them as overlaps as



(a) The Simple Case (see Figure 3.1a on page 21).

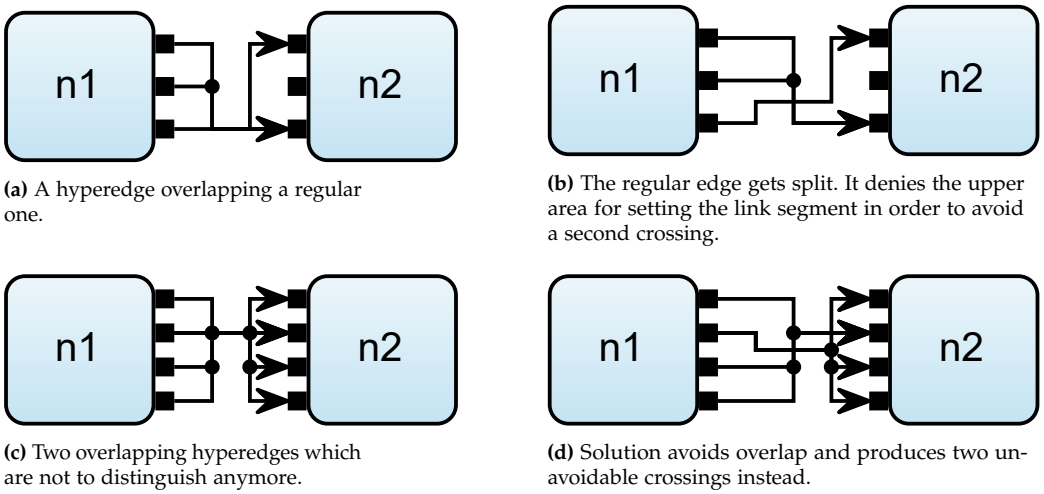


(b) The graph described in Figure 3.11 (see page 30) cannot place its link segment in the middle, because there is already a segment.



(c) The graph from Figure 3.10b (see page 29) with a pure conflict of size 1 between the edges (n1, n3) and (n2, n5).

Figure 5.1. Three graphs rendered with the new algorithm solving already introduced cases.



(a) A hyperedge overlapping a regular one.

(b) The regular edge gets split. It denies the upper area for setting the link segment in order to avoid a second crossing.

(c) Two overlapping hyperedges which are not to distinguish anymore.

(d) Solution avoids overlap and produces two unavoidable crossings instead.

Figure 5.2. Two graphs containing hyperedges, each represented by a layout computed with the old algorithm (left) as well as with the new one (right).

well, as explained in Section 3.3.1.

5.2 Hyperedges and Crossings

In Section 3.3.2 I hinted at the problems if dealing with hyperedges. When splitting, those could cause new crossings if not handling this split adequately. The rating process, explained in Section 4.4, is supposed to decide on an area to place the link segment in, which is of particular interest here. Figure 5.2 provides two examples of a graph containing hyperedges. In Figure 5.2a there are two edges, a regular one and a hyperedge overlapping each other. The regular one gets split and has now two potential areas to choose from. It computes a rating for both, with the result of the upper one causing two crossings and the lower one causing only one. Since crossing minimization is the prime directive for the rating, it chooses the latter as shown in Figure 5.2b. The second graph in Figure 5.2c considers two hyperedges. The split hyperedge looks as in Figure 5.2d, since I decided for a split to divide source and target segments in general. In this case the rating denied the lowest area since this had caused three crossings.

5.3 Saving Routing Slots

The next topic is the saving of routing slots. A first example is the solution to the Nested Case in Figure 5.3. The resulting layout is slightly different to my theoretical setting given in Figure 3.13b. However, those differences rely only on the actual arrangement which remains equivalent in its number of crossings and routing slots. In the end the rating saved one routing slot, hence it has in total only one more as for the layout produced by the old algorithm causing two overlaps. I explained in the final part of Chapter 3 why there cannot be more crossings after splitting than before, as long as there are only regular edges considered. Of course this excludes the additional crossing taken instead the overlap in the first place. That is, because it relies on the X-shape discussed in Section 3.1 and is thus unavoidable, as long as adhering to the routing spaces. So while the original layout of the Nested Case,

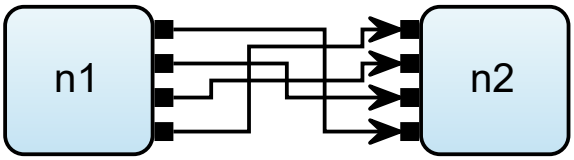
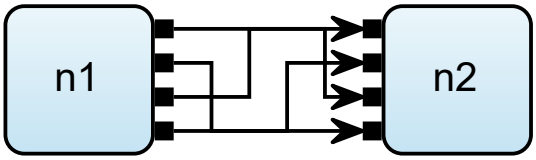
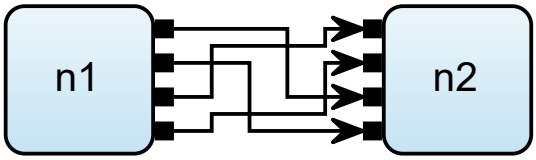


Figure 5.3. The Nested Case with a layout by the new algorithm. Similar to the theoretical solution hinted at in Figure 3.13b (see page 31).



(a) There occur two overlaps with the old algorithm.



(b) Computation of placement leads to no additional routing slot when using the new algorithm.

Figure 5.4. The *Cascade Case*.

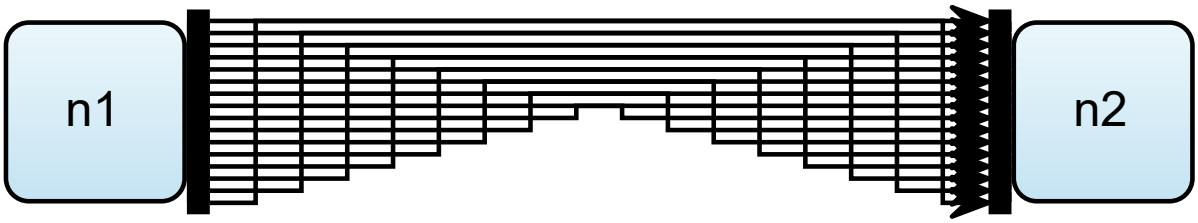
presented in Figure 3.12 (see page 30), included four crossings and two overlaps, the new algorithm produces six crossings and no overlap.

As for the *Cascade Case* given in Figure 5.4, there is actually no new routing slot needed. The secondary directive for the rating is the minimization of resulting dependencies. One may see in the the original layout Figure 5.4a that there are in total three potential areas between the ports to place a link segment in, whereas both vertical segments to split could reach two of them. The middle area is a possible area for both. Further it can be seen in Figure 5.4b that neither uses it in the actual solution. This relies on the additional dependency which were caused between both split edges if one of their vertical segments' link would be placed in the middle area. Of course this characteristic is only taken in consideration since there is for each placement the same number of crossings.

5.4 Minimizing the Port Distances

As explained in Section 4.5 I value it as necessary for the algorithm to produce correct layouts, even for port distances getting very small. For this matter I introduced the new conflict threshold. It relies on the minimal distances between all ports and is the reason the algorithm is able to handle complex layouts such as the one given in Figure 5.5. Note that this case is only a theoretical one, since it is based on fixed port orders and node placements as explained for the Simple Case in Section 3.2.1. The resulting layout is syntactically correct, though it is highly recommended to increase the node sizes for this to be more comprehensible. However, an early approach of mine was to leave such a layout as it is, since the solution would bring many computations, while it results in a layout which is in the end not at all more comprehensible. I decided against this for two reasons. First, this solution is still more reasonable than overlaps. Second, it gives a better notion for solving the problem with increased node sizes. Consider a user getting my solution, it would directly know that it could be more comprehensible with greater node sizes. A user getting the old algorithm's result may rather be bothered by the overlaps than actually realize the node sizes as too small.

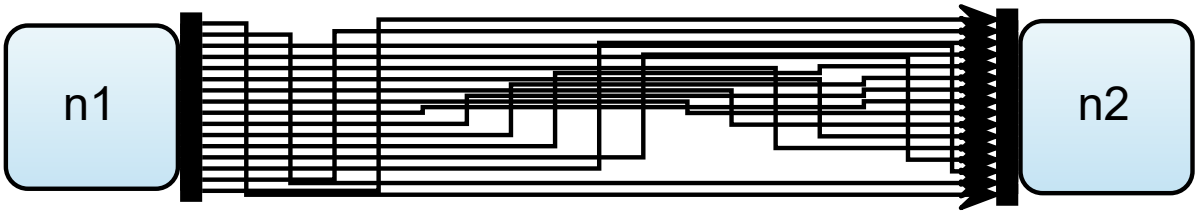
Unfortunately, since the conflict threshold scales with the very small port distances, it is now very small as well. This turns out to be reasonable for the aforementioned layout, but may produce a number of bad examples if not all pairs of opposing ports share the same y-coordinate. To demonstrate



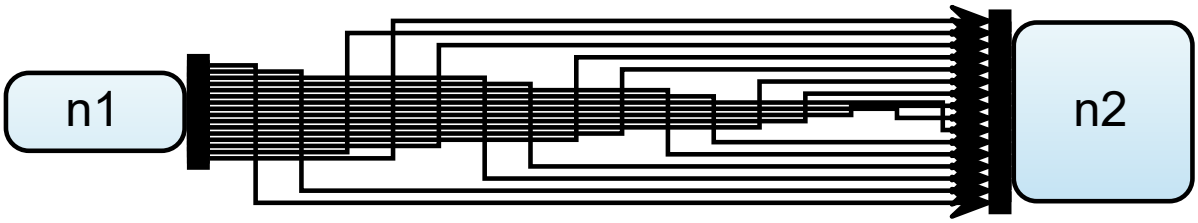
(a) There are eight overlaps in total with using the old algorithm.

(b) The layout remains unreasonable due to way too small distances between segments if using the new algorithm.

Figure 5.5. The Nested Case in a more complex variant. Now there are practically eight Simple Cases nested in one graph while maintaining the nodes' size.



(a) With slightly different node sizes.



(b) With significantly different node sizes.

Figure 5.6. The graph from Figure 5.5b with different node sizes.

this I modify the node sizes in Figure 5.6a in order to produce slight differences between the port placements. This results in some conflicts being valued as overlap and some as just that, a conflict. However, despite being inconvenient, those are still no real overlaps. Nevertheless one should note that the conflict threshold is computed for every routing space relying on the minimal distance between all ports lying in it. Hence cases like the one shown in Figure 5.1c could suffer from this. In case there is a node in the same routing space having a number of connected edges as those in Figure 5.6 minimizing the conflict threshold below 1, the conflict would be not valued as overlap.

Chapter 6

Conclusion and Future Work

In this chapter I draw a conclusion and summarize the work discussed in this thesis. Furthermore I will go into more detail about where my proposed solution provides further room for enhancements and sketch a few approaches for what it could be used for.

6.1 Conclusion

The orthogonal edge router used to lay out every edge with at most one vertical segment. This characteristic relies on the fact that each edge, which would be diagonal if drawing it straight, can be drawn with three orthogonal segments instead. My thesis proved that it could lead to problems with overlaps and thus violate the general notion of a good layout by producing ambiguity. Such overlaps make edges impossible to distinguish and thus the graph incomprehensible. Besides proving the problem's existence it declared its relevance, especially for graphs using equal node sizes which is a popular characteristic. Therefore, the final goal of my work was to solve this issue by introducing an extension for the given algorithm relying on additional segments for edges as a last resort to avoid an overlap.

This goal is accomplished by the new `OrthogonalOverlappingHandler` which realizes those additional segments with linking two hypernodes semantically and hence the vertical segments visually. Furthermore it turned out that the general addition of segments comes with the necessity of considering the potential production of additional crossings and spaces as well. I solved this problem with an approach which relies on checking out every possible location for every registered hypernode. This means serious additional expenses in the runtime and is thus a matter I would like to be reconsidered in the future and maybe to improved. But for now I value both approaches, the actual solution and its included crossing avoidance, as an appropriate first step towards a new domain of considering features based on the orthogonal edge router's new capability of handling overlaps.

6.2 Future Work

Despite a number of problems to care for, the proposed solution came, of course, with a lot of new possibilities as well. The purpose of this work was bringing edges more flexibility due to additional segments. At the current state this is only used as a last resort for avoiding overlaps. This chapter will describe a two hints at new features which might or might not benefit from my overlapping solution.

6.2.1 Reconsider Relevance of Crossings

Despite the discussed problem of overlapping segments the algorithm's notion of at most one vertical segment per edge is absolutely fine. As now proven, though, there are cases of this lack of flexibility standing in the way of a good layout. Besides overlaps, there are other problems in regards to this.

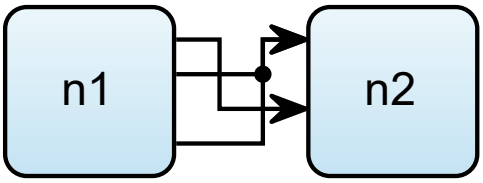


Figure 6.1. Layout rendered with the new algorithm. One of the crossings would be avoidable if using splitting.

My proposed solution is supposed to break the algorithm's general assumption only if absolutely necessary, that is to avoid overlaps. Thus the avoidance of overlaps is its prime directive. Since there are only two possible partial layouts for two vertical segments, an overlap included by one of them forces the algorithm to apply the second one. In other words now the algorithm will avoid overlaps at any cost. Such a cost may be a crossing which could be avoided if deciding on a split instead.

In Figure 6.1 is a layout produced with the new algorithm. There occur two crossings and one would be avoidable with a split edge. The regular edge could be split and the link segment placed above the upper source segment of the hyperedge. As described there are only two possible alternatives. One causes an overlap, but the other one does not. Hence the algorithm decides for the other one, despite the fact that it could avoid the crossing with a split. The algorithm would be able to handle this case if one adapts the comparison of hypernodes. For instance the whole comparison could try to avoid crossings as prime directive. If both partial layouts cause equal numbers of crossings, the pure conflicts may decide for a dependency. Hence, a partial layout $v_1 < v_2$, which number of crossings is bigger than its alternative's, forces a dependency (v_2, v_1). Before adding this dependency the partial layout $v_2 < v_1$ could be checked for causing an overlap and if doing so, the algorithm performs a split using my approach.

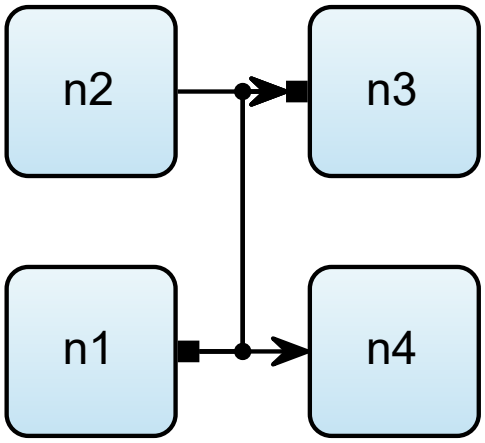
Of course this is only a raw idea. Implementing this would bring a few new challenges. For instance modifying the crossing detection, which for now also counts two crossings for pure overlaps with a distance of zero. The reason is that a hypernode having a horizontal segment's coordinate on another hypernode's horizontal segment's coordinate crosses its vertical one as well, which holds for both. Besides, there may be a few adaptions to prevent segments from coming too close, when primarily deciding on the least number of crossings. Maybe the general notion of distinguishing between pure conflicts and overlaps may actually become obsolete in the process, since now both are in most cases only avoidable with a split anyway.

6.2.2 Uniting Hyperedges

The next topic was initially under consideration to be part of this thesis and thus should be worth to discuss. In the current state the orthogonal edge router does not provide a proper handling for hyperedges. Instead one must build a hyperedge by letting a number of edges share the same port. So in order to define a hyperedge between three nodes n_1, n_2, and n_3 with n_1 being the source and $\{n_2, n_3\}$ being the target set, there must be an explicitly defined port $n_1.p$ as well. This port makes it possible to define the hyperedge as two regular edges $(n_1.p, n_2)$, $(n_1.p, n_3)$. When creating hypernodes the orthogonal edge router will add initially one port to a hypernode and crawls recursively through every other port connected to it in order to add them as well. In the end the hypernode has more than two ports appended to it as well as multiple source or target segments. For this case it is three ports, one source segment, and two target segments. The whole ordering process happens to this one hypernode, while every regular edge it consists of finally gets routed on its own using the hypernode's rank.

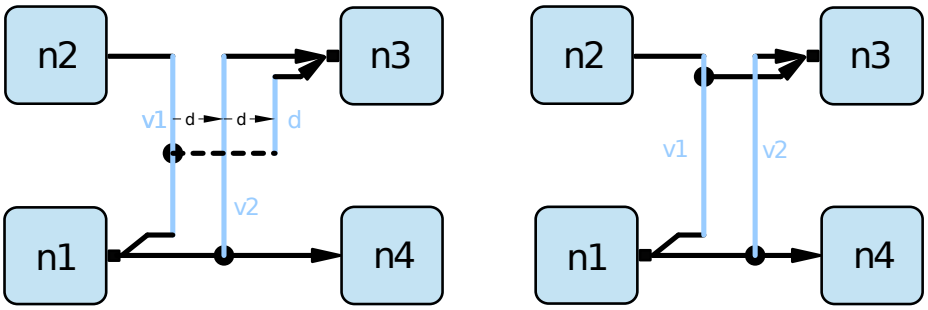
```
1   node n1 {port p}
2   node n2
3   node n3 {port q}
4   node n4
5
6   edge n1.p -> n3.q
7   edge n2 -> n3.q
8   // divide semantically?
9   edge n1.p -> n3.q
10  edge n1.p -> n4
```

(a) Representation in ELKT. Two hyperedges sharing the same ports p and q.



(b) The final layout. Both hyperedges unite to a bigger one and are impossible to distinguish.

Figure 6.2. Try to define two semantically different hyperedges sharing a number of ports.



(a) Solving approach with split.

(b) Solving approach without split.

Figure 6.3. Two approaches for solving the case introduced by Figure 6.2.

This workaround does the trick, but comes with a few inconveniences. One example is uniting hyperedges. For some cases one may want to define two semantically different hyperedges sharing the same port. This is not possible, since the edge router would put both in the same hypernode and define them as one big hyperedge. A possible scenario is showcased in Figure 6.2. An early approach to fix this was the application of additional vertical segments. With my new algorithm this would be possible. There is still the question of necessity, though.

In the current state there is no possibility to divide those two hyperedges semantically and my algorithm could not change that. But after solving this problem, for instance with a simple property appended to the edges, it may help to distinguish both visually in the final layout. A hint of a solution is shown in Figure 6.3a. As aforementioned, this solution requires a semantic differentiation in order to provide separate hypernodes v1 and v2 for both hyperedges. If doing so, one of them could be split to cross the other one instead of overlapping it. Unfortunately it does only affect the inner routing space. Hence I would recommend some kind of angle to fan out the outer port connecting segments, as it is shown in the example. Those angles should lead to another horizontal segment having at least the overlapping threshold's distance to the other one. Similar angles are already provided in *Ptolemy II*[1], a

[1] http://ptolemy.berkeley.edu/ptolemyII/

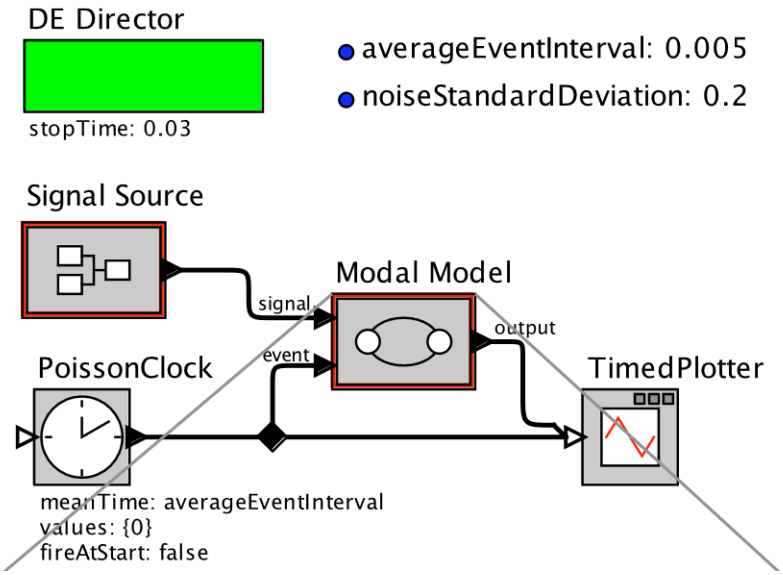


Figure 6.4. A detail of a modal model in Ptolemy II. There is a *fan segment* appended to the `TimedPlotter` in order to divide both edges connected to its port. Source: [Lee09].

framework for actor-oriented design [Bro16], as shown in Figure 6.4. They could be injected during the final routing process as additional bendpoint. The adjusted horizontal segment should be used already during the comparison, though, and I will refer to them as *fan segments*. I would highly recommend to use such a segment. Otherwise there is still an overlap which is bad for the layout whereas one must force the algorithm to ignore those overlaps which leads to bad code style, since it would be a workaround in a workaround. With consideration to the fan segments, simultaneous vertical segments lose a lot of potential as solution to uniting hyperedges. In Figure 6.3b the same solution is shown without splitting being absolutely fine.

This is only an example, though. It could be a matter to consider possible advantages of additional vertical segments. And even without them, a possible approach using the aforementioned fan segments would at least benefit from the overlapping threshold I introduced in this thesis. Since it is computed with the least possible space between any horizontal segments, it provides the possibility of placing those segments without worrying about any overlap.

Bibliography

[BET+98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing: algorithms for the visualization of graphs*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. ISBN: 0133016153.

[BK01] Ulrik Brandes and Boris Köpf. "Fast and simple horizontal coordinate assignment". In: vol. 2265. Sept. 2001. DOI: 10.1007/3-540-45848-4_3.

[Bro16] Christopher Brooks. *Ptolemy ii: an open-source platform for experimenting with actor-oriented design*. Poster presented at the 2016 Berkeley EECS Annual Research Symposium. Feb. 2016. URL: http://chess.eecs.berkeley.edu/pubs/1166.html.

[ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. "A fast and effective heuristic for the feedback arc set problem". In: *Inf. Process. Lett.* 47.6 (Oct. 1993), pp. 319–323. ISSN: 0020-0190. DOI: 10.1016/0020-0190(93)90079-O. URL: http://dx.doi.org/10.1016/0020-0190(93)90079-O.

[GJC79] M.R. Garey, D.S. Johnson, and Michael S. Mahoney Collection. *Computers and intractability: a guide to the theory of np-completeness*. Books in mathematical series. W. H. Freeman, 1979. ISBN: 9780716710448.

[Har87] David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90035-9. URL: http://www.sciencedirect.com/science/article/pii/0167642387900359.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquın Aguado, Stephen Mercer, and Owen O'Brien. "Sccharts: sequentially constructive statecharts for safety-critical applications: hw/sw-synthesis for a conservative extension of synchronous statecharts". In: *SIGPLAN Not.* 49.6 (June 2014), pp. 372–383. ISSN: 0362-1340. DOI: 10.1145/2666356.2594310. URL: http://doi.acm.org/10.1145/2666356.2594310.

[Lee09] Edward A. Lee. *Finite state machines and modal models in ptolemy ii*. Tech. rep. UCB/EECS-2009-151. EECS Department, University of California, Berkeley, Nov. 2009. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html.

[PFJ95] Helen Purchase, Robert F. Cohen, and Murray James. "Validating graph drawing aesthetics". In: *Graph Drawing* 1027 (Sept. 1995). DOI: 10.1007/BFb0021827.

[San04] Georg Sander. "Layout of directed hypergraphs with orthogonal hyperedges". In: *Graph Drawing*. Ed. by Giuseppe Liotta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 381–386. ISBN: 978-3-540-24595-7.

[Sch] Christoph Daniel Schulze. "Text in diagrams: challenges to and opportunities of automatic layout". submitted. PhD dissertation. Faculty of Engineering, Kiel University.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (Feb. 1981), pp. 109–125. ISSN: 0018-9472. DOI: 10.1109/TSMC.1981.4308636.