Interactive Tree Layout

Niklas Carstensen

Bachelor's Thesis
October 5, 2020

Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Sören Domrös

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 graph layouting is crucial for Model Driven Development (MDE) in Integrated Development Environments (IDEs) as users would otherwise have to draw a graph representation by hand every time the graph changes, which is time consuming. However, textual graph definitions only define nodes and their relations through edges with each other. The layout algorithm gets no information about how nodes should align beyond that. Therefore, results may not line up with the semantic meaning of the nodes and with what the user envisioned the graph to look like.

For this purpose the concept of *Intentional Layout* from earlier works by Petzold and Schönberner is applied on the tree graph drawing algorithm MrTree in this thesis. Furthermore, the structure of MrTree is revised and a new edge router is added to the algorithm as the current one routes edges in a straight line from node to node, which can result in edge-node overlaps. Finally *one dimensional compaction* is added to MrTree using the *Scanline algorithm*, which allows to minimize the area the graph takes up and to use screen real estate more efficiently.

The implementation of the proposed features was successful. However, interactions between the compaction and the edge routing algorithms can be hard to handle and may need future work. The results were shown to members of the Real-Time and Embedded Systems Group of Kiel University and feedback was implemented. Informal interviews and a small survey validated that the UI is intuitive but only slightly favor the new edge routing.

Acknowledgements

First I want to thank Prof. Dr. Reinhard von Hanxleden for giving me the opportunity to write this bachelors thesis. I also want to thank my advisor Sören Domrös for showing me how to navigate through the eclipse projects and KEITH modules I worked with, for answering my questions and for building the electron app of KEITH for the user evaluation. Finally I want to thank the entire working group, everyone who participated in the user evaluation, my friends, family and Kreia.

Additionally, I want to thank the Internet and Discord for allowing me to communicate during these times.

Contents

List of Figures

Chapter 1

Introduction

Graphs consist of nodes and edges that connect two or more nodes together. The nodes can represent any objects, and the edges can represent any relations those objects posses. Graphs are widely used in computer science due to their versatility and are studied in a subfield of mathematics called graph theory. Graphs are used in other scientific disciplines such as linguistics for modeling language syntax and compositional semantics, which follow tree-based structures or sociology where graphs are used to analyze and to model the behavior and influences of people onto each other[1].

A graph can be represented textually or visually. Both of those representations have advantages and disadvantes. A textual representation is easily editable in any text editor and version control is easier as established version control software supports merging of text files. However, the advantage of the visual representations of a graph is that the graphs structure is easier to understand and so semantic mistakes and relationships between larger groups of nodes or node clusters become much easier to see. However, drawing such a representation manually takes a lot of time for large or many graphs. Changing a graph later on becomes frustrating since a small insertion might require to change the whole graph. Algorithms for automatic graph layouting were developed to solve this problem. When converting a graph from the textual form, which is usually already generated or saved, to a visual one or in other words layout it, the positions of the nodes have to be generated and the edges have to be routed as this information is not provided in the textual representation. There are infinitely many ways to layout a graph but there are certain ways that should be avoided, such as layouts that place nodes on top of each other or route edges over nodes. Furthermore, there are more aesthetics criteria, which our algorithm should satisfy. Those aesthetics criteria aim to precisely formulate what it means for a graph to *look pleasing* to humans.

There are many different classes of layout algorithms such as the force-directed layout algorithm [Fru91], which works like a particle system of graph nodes with certain inter-particle attractions. This algorithm works well to visualize node clusters but it does not prevent overlap between nodes and edges and does not minimize overlap between edges.

Another algorithm that does not show node clusters but prevents overlaps is the layered algorithm, which is based on Sugiyama [STT81]. It has five phases: Cycle Breaking, Layer Assignment, Crossing Minimization, Node Placement and Edge Routing.

For trees there are a number of usable algorithms such as Lefty, Inorder [Knu71], WS [Cha79] and RT [Edw81]. However, here John Walkers Algorithm [II90] is most relevant, since it is used by MrTree. It combines the approaches of the WS and RT algorithms and handles out-of-bounds conditions, various node sizes and tree orientations while layouting subtrees in the same way regardless where they occur in the tree, producing reflected drawings for mirrored subtrees.

While those algorithms are made for drawing an optimal graph in regard to an aesthetics criterion or metric, they often fail to include the semantic context or the users mental map [LLY06] into the representation. To achieve this, user interaction is needed, since the graph does not carry the necessary information for algorithms to do it automatically.

[1] https://en.wikipedia.org/wiki/Graph_theory

1.1 Problem Statement

Using KEITH it is already possible to interactively set constraints for the layered [Pet19] [Sch19] and rectpacking layout algorithms that allow users to implement their mental map into a graphs representation. However, interactivity is not yet supported for a tree layout algorithm within KEITH.

In this work I will adapted the approach from Petzold [Pet19] and Schönberner [Sch19] to the tree layout algorithm in ELK, which is called MrTree. I will also construct a User Interface (UI) in KEITH, rework and append to parts of MrTree to improve its usability within KEITH and KIELER as well as in its current real world use case, which is rendering DomTrees from Sequential Constructive Statecharts (SCCharts).

1.2 Outline

Chapter 2 introduces important definitions and technologies, which are used later on. After that Chapter 3 describes the ways other works tackled similar problems. Chapter 4 convers used work such as the current architecture and inner workings of MrTree, which are important for the implementation. Chapter 5 is about introducing concepts for the three main topics child node ordering, edge routing and compaction. In Chapter 6 the concepts from Chapter 5 are implemented into MrTree. Chapter 7 describes how this work was evaluated, as for example in the user evaluation or in the expert feedback that was given. Finally the thesis is concluded in Chapter 8 and currently not yet solved problems that may be topic of future work are introduced.

Chapter 2

Preliminaries

The later chapters of this thesis are based on a number of technologies and concepts, which is why important definitions and terms are explained here.

2.1 Graph Definitions

Many of the definitions here are from university lecture notes. However, some tree related terms were added, which are important for the following chapters.

Graph A *graph* $G = (V, E)$ is a set V of *vertices* and a set E of *edges*, in which an edge joins a pair of vertices. If the pairs of vertices within the set of edges are ordered the graph is called directed graph. Vertices can be called nodes.

Path A *path* of length $n \in \mathbb{N}$ of a graph $G = (V, E)$ is a tuple $(v_1, .., v_n)$ with $v_1, .., v_n \in V$ and $(v_i, v_{i+1}) \in E$ for all $1 \leqslant i < n$.

Connected A graph G is *connected* if $\forall (v_a, v_b) \in V \times V : \exists p = (v_a, .., v_b)$ with p being a path.

Cycle A *cycle* is a path with $v_1 = v_n$.

(A)cyclic A graph is called *acyclic* if it does not contain any cycles, inversely if a graph does contain cycles it is called *cyclic*.

Tree A *tree* is an acyclic connected graph.

Child/Parent A vertex v is a *child* of a vertex p and p is the *parent* of v if $(p, v) \in E$.

Siblings Vertices with the same parent p are called *siblings*.

Root *Root nodes* or just *roots* are nodes without a parent node. A tree graph may only have a singular root node.

Leaf *Leaf nodes* or just *leaves* are nodes without a child node.

Depth The *depth* of a vertex v in a tree is the length of the path p from v to the root of the tree.

Level A *level* of a vertex is its depth + 1 and the level $n \in \mathbb{N}$ of a graph is the set of nodes with the level n.

Ancestors The *ancestors* of a vertex a in graph $G = (V, E)$ are all vertices $v \in V : (v, x) \in p \vee (y, v) \in p$ with p being the path from a to the root node of the graph and $x, y \in V$.

Curve A *curve* is a subset of $\mathbb{R}^2$ of the form $\alpha = \{y(x) : x \in [0, 1]\}$ where $y : [0, 1] \to \mathbb{R}^2$ is a continuous mapping from closed interval $[0, 1]$ to the plane. $y(0)$ and $y(1)$ are the *endpoints* of curve α.

Simple curve A *simple curve* is a curve without repeating points except possibly the first and the last.

Drawing A drawing Γ of a graph is a function that maps each vertex v to a point $\Gamma(v) \in \mathbb{R}x\mathbb{R}$ in a plane and each edge (u, v) to a simple curve $\Gamma(u, v)$ with endpoints $\Gamma(u)$ and $\Gamma(v)$.

Aesthetics We use the term *Aesthetics* to denote the criteria that concern graphic aspects of readability [TDB88].

2.2 ELK

The ELK framework provides a set of layout algorithms, such as MrTree, and an infrastructure that bridges the gap between layout algorithms and diagram viewers and editors[1]. It defines a common graph data structure that consists of `ElkNodes` and `ElkEdges` with `ElkNodes` being the graphs vertices and the `ElkEdges` being the graphs edges. Beyond their relation as vertices and edges these ELK objects contain additional information. `ElkNodes` have a position, a width, a height, a label and more associated with them, while the `ElkEdges` consist of a set of bend points and more. The bend points of an edge are the points between which the line segments of the edge are drawn. This data structure supports properties for edges and nodes, which can be used to set graph layout options or save certain information about a graph element. It supports a hierarchical graph structure, which means that another graph can be defined inside of an ElkNode. The entire graph is always inside a parent ElkNode.

Each Layout algorithm in ELK has to define an ELK Metadata File (melk) file, which contains information about the algorithm such as the algorithms id, a label, a description, a preview image, features and layout options. There are options that are predefined within ELK so they can just be imported but it is possible to create new options. In that case one has to define the options name, its datatype such as int, boolean, an enum or something else, a label, a description, a default value, a lower/upper bound and a target. The target of the option describes whether the option is set for nodes, edges or parents. Parents are nodes with child nodes inside of them. The options from the melk file are then written into an options class as properties and can be used by the layout algorithm from there. Besides the options class generated by the melk file, the layout algorithm can define internal property classes, which contain properties that can not be set by the user or from the outside but are used to store information during the run of the layout algorithm.

The output layout for the graph that ELK produces has to be rendered in another framework.

2.3 Language Server Protocol (LSP)

The LSP is a open source protocol, which was originally designed by Microsoft for VSCode[2]. It is based on JavaScript Object Notation - Remote Procedure Call (JSON-RPC) and allows language servers to supply IDEs with language specific features by defining a common API. This solves the m IDEs n languages problem [KPE16]. Language specific features do not have to be reimplemented on every IDE but instead can be implemented on a language server, which can then be used in every IDE with support for language servers. This brings down the development time for m IDEs and n languages from $n \cdot m$ to $n + m$.

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

[2] `https://microsoft.github.io/language-server-protocol/`

```
"jsonrpc":"2.0",
"method":"textDocument/didChange",
"params":{
   "textDocument":{
      "uri":"file:///c%3A/Users/One/...keith_workspace/test_size.elkt",
      "version":9
   },
   "contentChanges":[
      {
         "range":{
            "start":{
               "line":55,
               "character":18
            },
            "end":{
               "line":55,
               "character":18
            }
         },
         "rangeLength":0,
         "text":"3"
      }
   ]
}
```

Listing 2.1. Example LSP request from KEITH that is sent on file change.

When an user changes a text document in KEITH, a request JavaScript Object Notation (JSON) with three elements is sent to the language server. The first two are the id and the kind. The kind is a string that specifies the request. For most requests it is set to *open*, *ready* or *data*. If kind is set to *data*, the request contains a third element called content, which is a JSON-RPC string such as the one seen in **Listing 2.1**. Furthermore, there are other events such as text hover or text highlight that the language server can respond to.

2.4 Theia

Theia[3] is a cloud IDE framework that was developed by TypeFox[4] using the TypeScript language and is based on VSCode. It was build on the LSP and therefore supports over 60 languages through language servers[5]. Theia is split into front-end and back-end processes, which both use Node.js. The back-end of Theia may run on any sever while the frontend can run inside a browser or an electron app. Alternatively the front and back-end together with the language servers can all be build into one electron app. The processes communicate through JSON-RPC messages over WebSockets or REST APIs over HTTP[6]. Theia can be extended using Node Package Manager (npm) package extensions and ships with some build in extensions.

[3] https://github.com/eclipse-theia/theia

[4] https://www.typefox.io/

[5] https://theia-ide.org/

[6] https://theia-ide.org/docs/architecture/

Extension developers can use the contribution interfaces as the *CommandContribution* or *MenuContribution* interfaces of Theia to interact with the IDE. Theia then loads those extensions using the dependency injection modules defined in the extensions and at startup generates a global dependency injection container for the frontend and the back-end processes[7].

2.5 Sprotty

Sprotty is a open-source web-based diagram rendering engine that is already integrated into Theia[8]. Like most web graphics frameworks it uses Scalable Vector Graphicss (SVGs) to display objects on the screen. It is built to handle asynchronous state changes and to render the resulting animations without artifacts. This improves the User Experience (UX) and makes it easier for the user to follow changes in the diagram. Sprotty is split into client and server. When drawing a diagram, the server will only send the relevant data from a possibly large dataset to the client via JSON-RPC and the client will render that information to a SVG using web-native technologies[9].

2.6 KIELER

KIELER is an open source IDE based on Eclipse, which aims to enhance MDE by automatically drawing a graph of the code the developer is writing and supports a variety of modeling languages such as SCCharts[10]. KIELER generates its drawings by performing a synthesis, getting the node sizes from Kieler Light Weight Diagrams (KLighD)[11] [SSv13], getting the graph's layouting from ELK [Pet19] and rendering the graph on the screen using the Java Version of Piccolo2D[12], which is based on Java2D[13].

2.7 KEITH

KEITH is a port of the KIELER framework to the cloud based IDE Theia by Domrös [Dom18] and Rentz [Ren18] and therefore can be used from a browser or as an electron app.

While porting existing technologies to the cloud it aims to be more modular due to less language specific code in the IDE, and to be easier accessible with fewer buttons and dialogs[14]. It uses Sprotty as its rendering engine instead of Piccolo2D.

The Graphical User Interface (GUI) of KEITH is designed similarly to VSCode and KIELER. All views in the GUI can be placed to the positions the user desires. In **Figure 2.1** and **Figure 2.2** on the left sidebar are views such as a file browser for the currently active workspace folder and a text search view. In the middle of **Figure 2.1** and **Figure 2.2** are the currently opened text files to the left and the diagram view on the right. Under both are the Problems and KIELER compiler views. On the right sidebar there are the Diagram Options and Outline views. In contrary to KIELERs GUI, the diagram options are a separate view to the actual diagram view here.

[7] https://theia-ide.org/docs/authoring_extensions/
[8] https://www.typefox.io/blog/sprotty-a-web-based-diagramming-framework
[9] https://www.eclipse.org/community/eclipse_newsletter/2018/october/sprotty.php
[10] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview
[11] https://github.com/kieler/KLighD
[12] http://piccolo2d.org/
[13] https://docs.oracle.com/javase/8/docs/technotes/guides/2d/spec/j2d-intro.html
[14] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KEITH

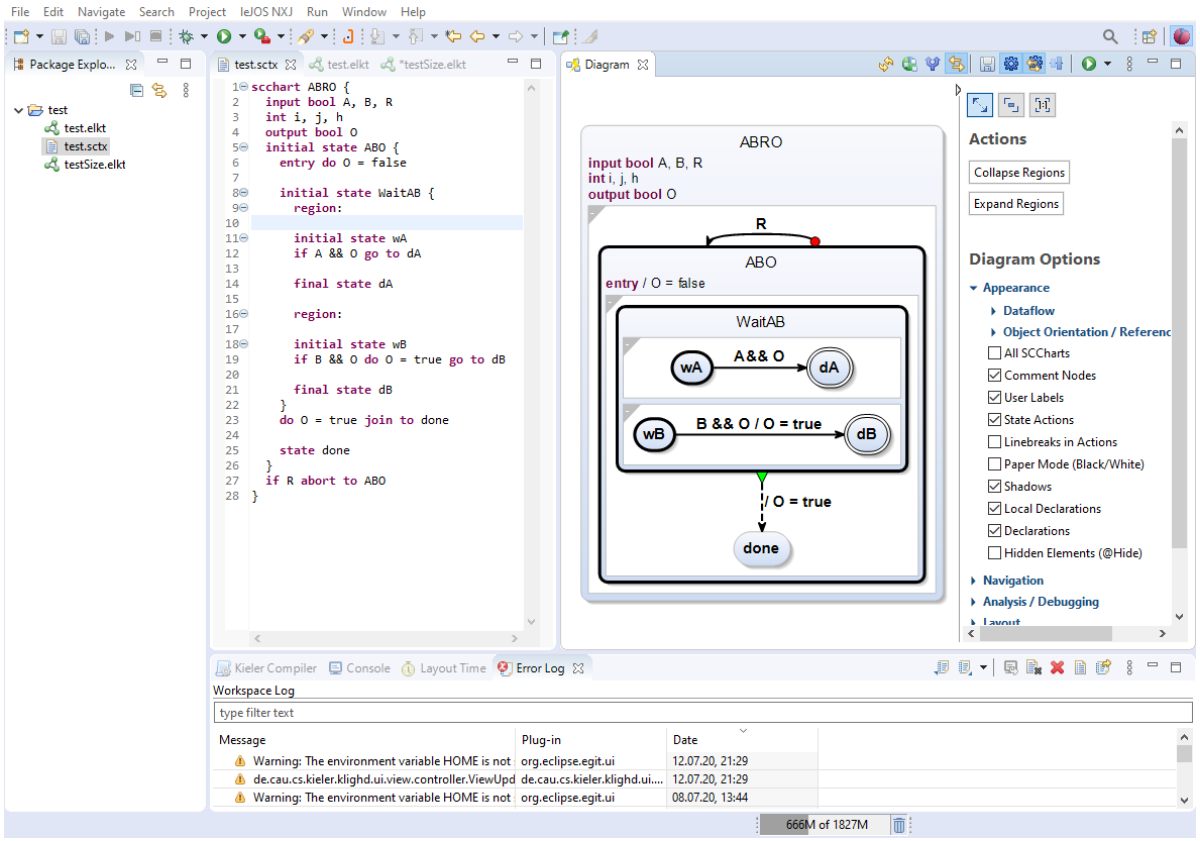


Figure 2.1. A visualization of a sample SCChart in KIELER

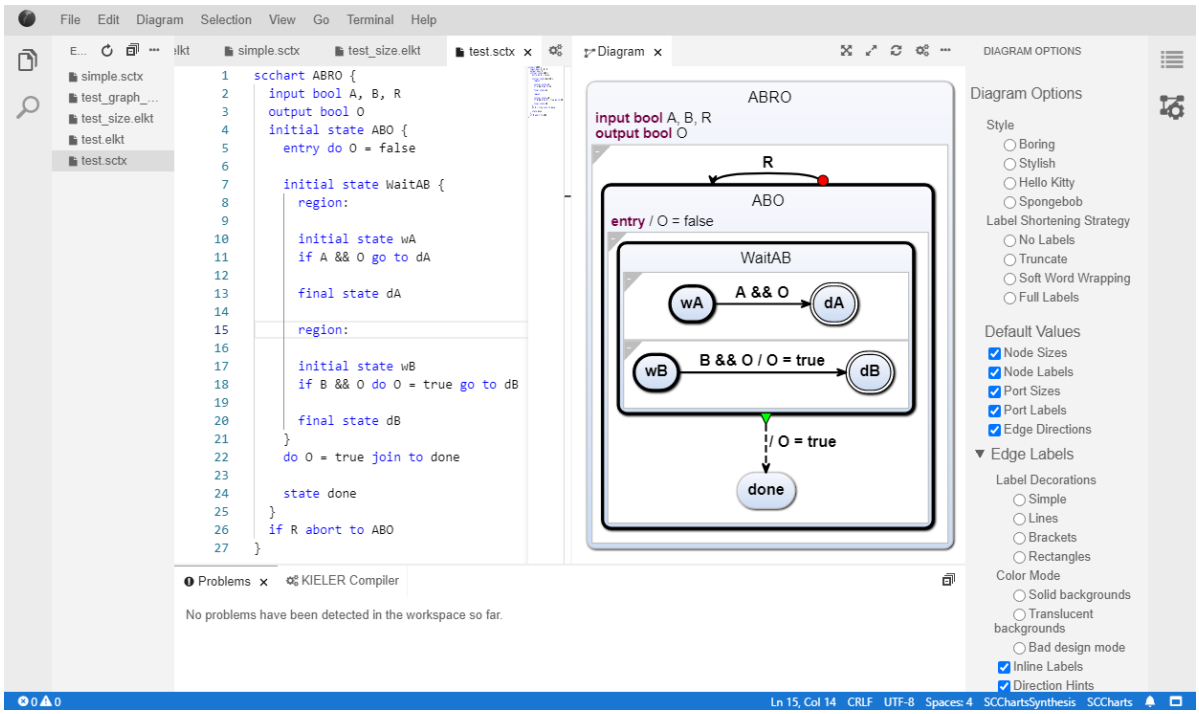


Figure 2.2. A visualization of the same sample SCChart in KEITH in Light Theme

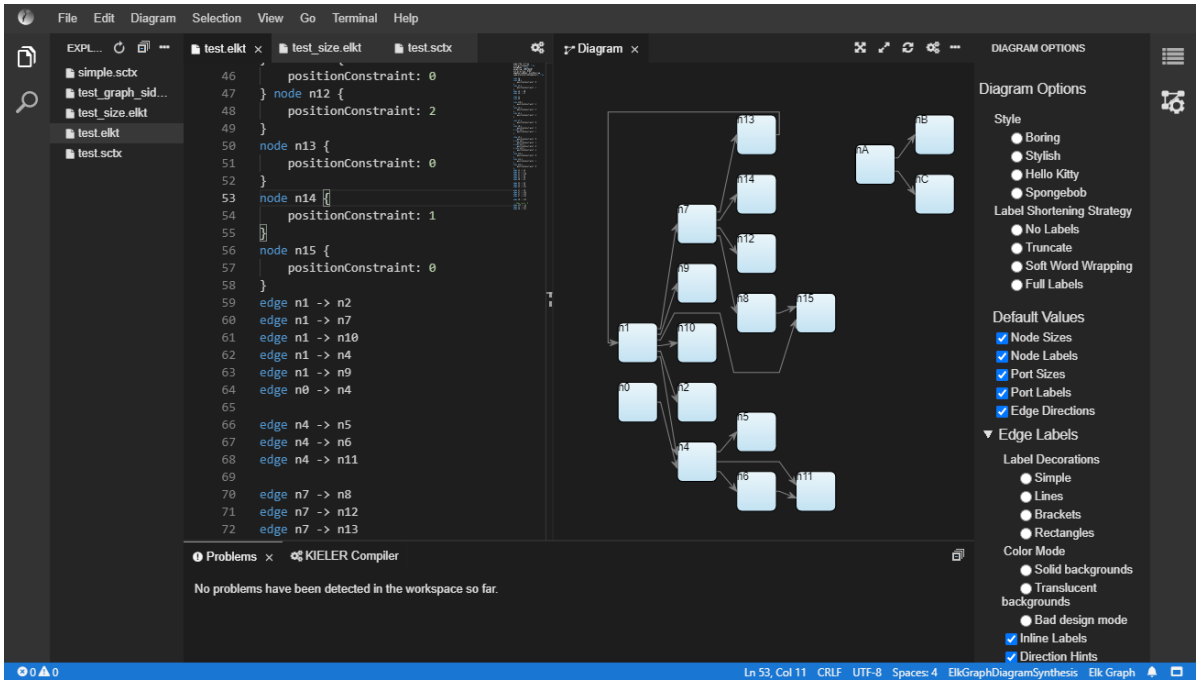


Figure 2.3. A visualization of an ELK graph in KEITH in Dark Theme

Chapter 3

Related Work

Other works already address interactivity in automatic graph layouting. Here they are shortly introduced.

3.1 Layered Layout Constraints

Böhringer and Paulisch [BP90] define layout constraints as a means to combine the advantages of automatic layout, which can generate a drawing that is aesthetically pleasing, and manual layout, which can implement secondary notation [Pet95] into the drawing.

As an example they show the implementation of this concept on Sugiyama's layout algorithm [STT81]. For this three low level constraints are introduced.

1. *Absolute Positioning* constrains the node to a level and a position or range of positions.

2. *Relative Positioning* constrains the node left, right, over or under another node.

3. *Clusters* constrain nodes to be drawn close together.

The low level constraints, which are defined as linear equations, are enforced by constraint managers for each dimension. All constraint managers communicate with a 3-D constraint manager for three dimensional layouting.

3.2 Tree Layout Constraints

He and Marriott [HM98] propose constraints for tree layout. Similarly to Böhringer and Paulisch [BP90] they use linear equality and inequality constraints for each dimension. The problem of satisfying all set constraints is then handled as a quadratic programming problem.

Such constraints allow for *Absolute Positioning* by constraining a node's x and y position to certain values. For example $x_1 = 10$ and $y_1 = 10$ constrain a node to the point (10, 10). However, *Relative Positioning* are possible too by using inequalities such as $x_1 > x_2$, which constraints node 1 to the right of node 2. Furthermore, nodes $n, m \in \mathbb{N}$ can be constrained to be maximally $z \in \mathbb{N}$ apart from each other in one dimension by adding the constraints $x_n - x_m < z, x_m - x_n < z$. Doing this for multiple nodes and all dimensions of the drawing results in a cluster constraint.

While this approach is similar as it is about constraints in a tree layout algorithm, here constraints are not implemented as equations but as absolute constraints, which means the quadratic programming approach is not transferable.

3.3 Intentional Layout in Sprotty Diagrams

The two theses from Petzold [Pet19] and Schönberner [Sch19] propose a concept for setting layout constraint interactively in the layered layout algorithm and show an implementation on KEITHs Sprotty diagrams. Absolute constraints that allow to set a node's layer and position within the layer were defined. Furthermore, principles for the constraint reevaluation [Sch19] were proposed and implemented:

1. No unwanted changes of the layout
2. No constraints for not affected nodes
3. No avoidable constraint changes

Furthermore, a UI was defined and implemented into KEITH [Pet19] that shows valid positions the currently dragged node may be dragged to using drawn circles as well as visualizing layers by using dashed rectangles. Forbidden layers, which are layers, in which adjacent nodes of the dragged node, are inside of, are marked using red rectangles. To identify where the currently dragged node is from, a shadow of the original node at its original positions is displayed. To identify the constraints of a node, lock icons are displayed next to the nodes.

In the context of trees a layer constraint would be a constraint that forced a node to be placed into a level of the tree that it normally would not be placed in. However, since most tree layout algorithms set the height of a node based on its depth in the tree, users will expect this behavior here too.

Chapter 4

Used Algorithms

4.1 MrTree

MrTree is the tree layout algorithm of ELK and consists of four layout phases, some of which are similar to phases from the layered algorithm from Sugiyama [STT81]. While MrTree is meant to layout trees, the algorithm can layout graphs that are not trees, too, by using cycle breaking and by choosing a primary parent node for nodes with multiple parents.

Here the original version of MrTree is shown. The new version is developed in Chapter 6. An overview over the phases and intermediate processors of the original MrTree algorithm is shown in **Figure 4.1**.

4.1.1 Graph Import

The entry point of MrTree is, like for every layout algorithm within ELK, its layout method in its layout provider class. The layout method receives an `ElkNode`, which contains an ELK graph and a progress monitor, which can be used to measure the performance of the algorithm.

The first bit of code of the layout algorithm converts the ELK graph into a `TGraph`, which is the graph format the rest of the algorithm works with.

Then the graph is split into components using the `Components Processor` class. It identifies graph components by starting DFS searches from each node of the graph. The `Components Processor` can be turned off using the SEPARATE_CONNECTED_COMPONENTS option. While this class has processor in its name, it is not part of the layout phases and is not an intermediate phase.

After that the layout phases are applied in their order to each component.

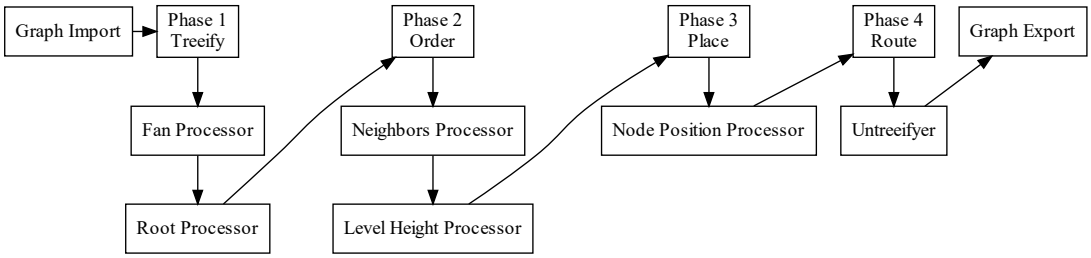


Figure 4.1. The phases of the original MrTree Algorithm

4.1.2 Intermediate Processor

Intermediate processors may be added before or after layout phases. The `process()` method of the processor will then be called on the graph at the set point in time between the layout phases.

4.1.3 Phase 1: Treeify

This phase is somewhat similar to the cycle breaking layout phase from the layered layout algorithm. During this phase the graph is stepped through using either DFS or BFS and edges back to nodes that have already been visited are removed from the graph and saved into an internal property of the graph.

This phase adds the intermediate processor DETREEIFYING_PROC after the fourth layout phase, which adds the removed edges back to the graph.

4.1.4 Phase 2: Order

The original MrTree algorithm contains two classes that can handle this phase, *NodeOrderer* and *OrderBalance*. Only the *NodeOrderer* class is registered in the algorithm so *OrderBalance* is not used. When trying to swap out the *NodeOrderer* class for the *OrderBalance* class the graphs child nodes were not sorted. Due to this only the *NodeOrderer* class is relevant here. The *NodeOrderer*, however, does not react to the algorithms weighting option, which should change the parameter that the nodes are sorted by, so this option does not work in the current version of the algorithm.

NodeOrderer steps through the graphs levels and sorts their nodes by the value of their FAN property. Child nodes can be sorted by manipulating the outgoing edge list or their parents. The order of the nodes in that list determines the order the nodes will be placed in in the third phase and with that in the final graphs drawing.

This phase sets the intermediate root processor before the second phase, which sets the ROOT property to true for root nodes. This intermediate processor adds a SUPER_ROOT dummy node to the graph if there are multiple root nodes or a DUMMY_ROOT dummy node if the graph is empty.

The fan processor is set before the second layout phase.

4.1.5 Phase 3: Place

This phase implements the algorithm proposed by Walker [II90]. It utilizes the concept of building subtrees as rigid units and the concept of using two variables for the positioning of each node, the PRELIM and MODIFIER variables. The algorithm does two DFS walks over the graph.

The first walk traverses the tree in postorder and sets the two fields. The PRELIM property is the preliminary x-coordinate of a node. For leaves it is set to the sum of the PRELIM value of the left sibling, the node node spacing and the mean size of the left sibling and the current node. If a node does not have a left sibling, the PRELIM value is set to 0. For subtree roots PRELIM is set similarly but with the subtree midpoint and the right contour in mind. The MODIFIER variable is only set for subtree roots and is based on the midpoint of the subtree and the prelim value. A subtree midpoint value in this case refers to the average between the prelim values of the rightmost and leftmost nodes in the tree.

The second walk traverses the tree in preorder and sets the final x position of each node based on the PRELIM and MODIFIER values of its ancestors. The final y position of the node can not be set solely based on the nodes level here. It has to take the sizes of the levels before it into account. The size of a level is the size of the largest node in it so the current height has to be saved as the graph is traversed.

However, the final position of the node in this implementation is not set yet but the XCOOR and YCOOR properties of each node. These values are then later applied to the actual node positions in the `Node Position Processor`, which this phase adds before the fourth phase.

This phase sets the intermediate `Root Processor` before the second phase too, but that does not change anything as the processor is already set to that position by the second phase.

Before the third phase the `Level Height Processor` and `Neighbors Processor` are added. The `Level Height Processor` walks through the graphs levels and sets the LEVELHEIGHT property of each node in the level to the height of the tallest node in the level. The `Neighbors Processor` similarly walks through the graphs levels and sets the LEFTNEIGHBOR and RIGHTNEIGHBOR properties of the nodes. If the parents of two nodes are the same the right nodes LEFTSIBLING property is set the left node and the other way around. The parent of a node in this case means the node at the beginning of the first edge in the `incomingEdges` list of the node. This distinction has to be made because in MrTree a node may have multiple parent nodes and a graph with such a node should still be layouted like a tree.

4.1.6 Phase 4: Route

This layout phase's `process()` method only removes the bend points for each edge in the `outgoingEdges` list of each node in the graph. However, since MrTree does not add any `bendPoints` to the `edges`, this layout phase does not change the graph. It does not add any intermediate layout processors to the configuration.

4.1.7 Graph Export

After all layout phases and intermediate processors have been applied, the graph components are combined into one whole again by the `Components Processor`.

Each component occupies a rectangular space and these spaces have to be combined. The order of the components is determined by their size and the value of the priority property of the graph. After that the maximum row width is calculated from the graphs aspect ratio property. With the maximum row width and a sorting computed the components are placed iteratively into the graph.

Now the graph needs to be aligned with the axes of the coordinate system so there can be no nodes with negative position coordinates and there has to be at least one node with a x coordinate of zero and at least one node with a y coordinate of zero. After that padding is applied. This process is visualized in **Figure 4.2**.

Now that the graph is positioned correctly, the edge's bend points, which are still missing, are added. This is done by adding the position of the source node and the position of the target node to the `bendPoint` list of the edge and then moving those to the border of the node.

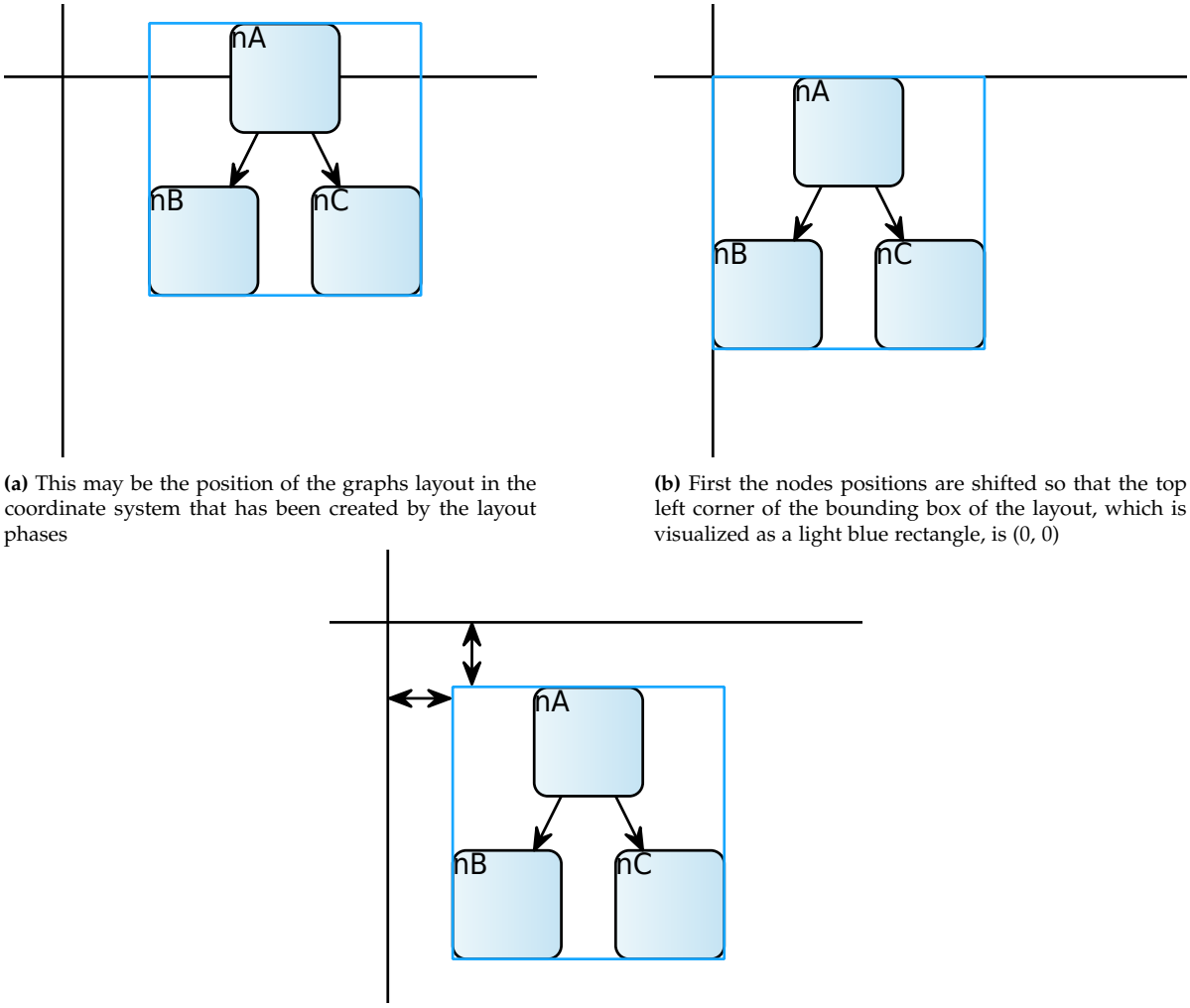
After that is done the graph only has to be converted back to a ELK graph and returned.

4.2 Compaction

In this section compaction and a key algorithm that is used in Chapter 6 is introduced.

4.2.1 One Dimensional Compaction

Compaction is about moving the nodes of a graph so that the total area of the graph is minimized without letting two nodes overlap. While the problem of two-dimensional Compaction is NP-Complete, there are efficient solutions for the one-dimensional compaction problem that work in $O(n \log n)$.



(a) This may be the position of the graphs layout in the coordinate system that has been created by the layout phases

(b) First the nodes positions are shifted so that the top left corner of the bounding box of the layout, which is visualized as a light blue rectangle, is (0, 0)

(c) Then padding is applied

Figure 4.2. The graph alignment process after the layout phases of MrTree

For compaction a constraint graph $CG = (R, C)$ is needed [Rüe18]. It consists of a set of rectangles R and a set of constraints C. The rectangles are the nodes of our graph, while the constrains are tuples of two rectangles (r, s) that are added iff r is above/left/right/under s depending on the direction the constraint graph should be constructed for.

Assuming a downward layout direction and compacting upwards using a constraint graph, the rectangles can be moved as far up as possible under the node above them. However, grouping has to be considered. Some nodes such as the child nodes of another node for trees should always be under their parent node even if they could be moved further upwards without causing overlap.

4.2.2 Scanline Algorithm

The first solution that comes to mind when wanting to construct a constraint graph is going through all nodes n for each node m and checking if n is above m. However, that solution would be in $\mathcal{O}(n^2)$.

A faster solution would be the scanline algorithm [Len90]. It sweeps along the graph in a positive direction, which for our coordinate system means a downward or rightward direction, perpendicular to the direction the constraints are calculated in, stopping at each point of interest, which are the beginnings and ends or for example left and right corners of the rectangles.

If the point of interest for the scanline is the beginning or upper left vertex of the rectangle r then r is added to s, which is a ordered set, sorted by the y coordinate of r. The constraint candidate of r is set to the element in s that is under r and the constraint candidate of the element in s above r is set to r.

If the point is the end or lower right vertex of the rectangle, then (r, t) is added to c, with t being the element in s under r if t is the constraint candidate of r, or (u, r) is added to C with u being the element in s above r if the candidate of u is r.

The scanline algorithm is in $\mathcal{O}(n \log n)$ due to the sorting of the points at the start of the algorithm.

Algorithm 1: Scanline algorithm from [Rüe18]

Input: R: set of rectangles
Data: $cand[r]$: constrains candidates indexed by rectangle, S: sorted set of rectangles
Output: C: set of constraints

```
1  points ← all begin and endpoints of all rectangles;
2  sort points ascendingly (prioritizing the rectangle end points);
3  foreach p in points do
4      if p is start point then
5          r ← r(p);
6          put r into S;
7          cand[r] ← S.left(r);
8          cand[S.right(r)] ← r;
9      else
10         if S.left(r) exists and S.left(r) = cand[r] then
11             add(r, S.left(r)) to C;
12         else
13         end
14         if cand[S.right(r)] = r then
15             add(S.right(r), r) to C;
16         else
17         end
18         remove r from S;
19     end
20 end
```

Chapter 5

Interactive Tree Layout

Since MrTree did not implement a form of constraints of the position of child nodes, I have added this feature as described in the following sections. Furthermore, the changes to KEITH and the LSP and the addition of the edge routing and compaction are described there.

5.1 Software Architecture

One of the first changes to the original MrTree Layout Algorithm as described in **Chapter 4** is moving the edge routing from the postprocessing, described in the **Graph Export** section, to the fourth phase of the layout algorithm so that the edge routing phase does what its name implies.

5.2 Layout Direction

Since the layered algorithm implements a layout direction option, which is defined in ELK, the same option is implemented into MrTree.

Nonetheless for simplicity the DOWN layout direction is assumed for the rest of this chapter.

5.3 Child Node Ordering

This feature is about the adaption of the concept of intentional layout that was developed and implemented on the layered layout algorithm by Petzold [Pet19] and Schönberner [Sch19] to MrTree. The concept of the *Position Constraints* was kept here and applied to MrTree's second phase, the Node Ordering Phase. However, the `Layer Constraints` were not adapted as explained in Chapter 3.

5.3.1 MrTree Position Constraint Sorting

The algorithm should not just move nodes to their assigned positions, it should also manage multiple nodes with the same index or nodes with indices bigger than the size of the siblings sensibly as those may be set textually be the user. Instead of changing non valid constraints as proposed by Schönberner [Sch19], they are not changed here. Instead I have implemented a robust way for the layout algorithm to handle them as reasonable as possible as they can only be changed from within KEITH and the feature for MrTree may be used from other views like KIELER too. To achieve that the nodes are sorted into an array of the siblings size based on index priorities.

1. Priority: **Valid indices** for positions within the array that are still unassigned.
2. Priority: **Duplicate indices** for nodes that could not be assigned as their specified position was already taken are now assigned as close to their desired position as possible.

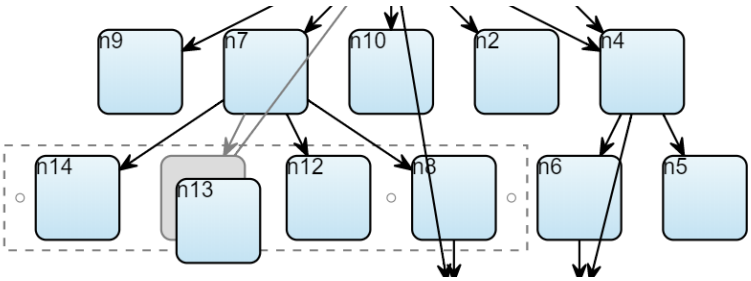


Figure 5.1. An example of the rectangle around the nodes siblings of the dragged node `n13`, visualizing the range a node may be ordered in

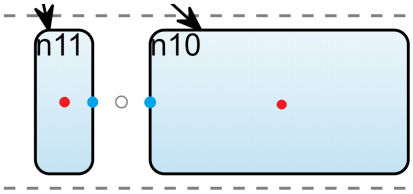


Figure 5.2. An visualization of the calculation of the circles between nodes

3. Priority: **Out of bounds indices** are assigned to the still unassigned indices inside of the array and preferably closer to the end of the array.

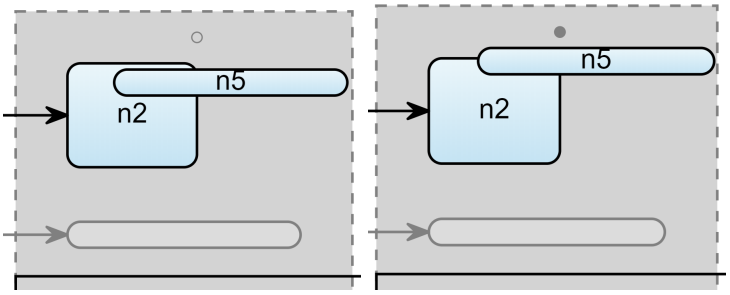
4. Priority: **Undefined indices** are indices below zero that are sorted into the still free slots of the array at the end. Undefined indices are not resorted from the original list ordering so if all nodes of a set of siblings only have undefined indices, the siblings will not be resorted from the original outgoing edge list ordering. The default value for a node constraint is -1 so nodes have an undefined index by default.

5.3.2 Sibling Visualization

The siblings of a node, which are the nodes that may be sorted using this feature, are not always equal to the nodes of the entire level. Therefore, the nodes cannot be sorted completely and node siblings will always be clustered together. To visualize this limitation a dashed rectangle was added around the nodes that can be ordered by the user based on feedback, as seen in **Figure 5.1**.

Circles are placed to the available locations between the nodes and at the sides of the siblings if moving the node to those positions would actually change the ordering of the siblings. Furthermore, space has to be left in the dashed rectangle for the circles at the sides of the siblings.

There are many ways to place the circles that show possible node positions. For example they can be placed to the mean position of the middles of both nodes as marked in **Figure 5.2** by the red dots. However, if both nodes are of different sizes that would lead to the circle ending up inside the larger node. So another way to deal with this is to take the mean position of the closest sides of the rectangles middle position of both nodes, as marked by the blue dots in **Figure 5.2**.



(a) On average n5 is higher than n2 but the layered algorithm's KEITH UI indexes n5 is below n2

(b) Here n5 is recognized as being above n2

Figure 5.3. Examples of the dragged node position index calculation in the current layered algorithm

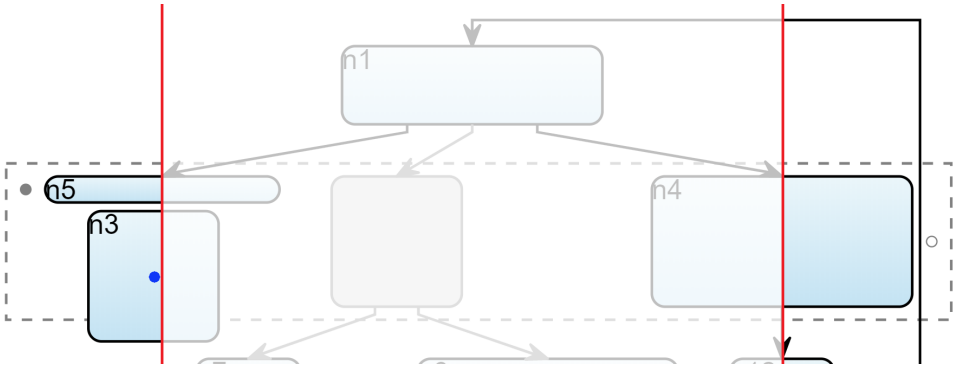


Figure 5.4. A visualization of the hitboxes of n5 and n4 left and right of the red lines and a blue dot in the middle of the dragged node, which symbolizes the point the hitbox areas are checked against

5.3.3 Drop Hitbox

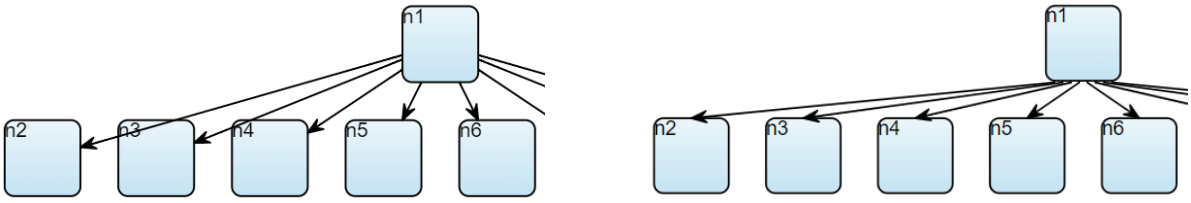
The GUI within KEITH of the layered layout algorithm determines the current node index by checking the relation of the two nodes positions, as seen in **Figure 5.3**. However, the mean position of all points within the rectangle of n5 in the left figure is already higher than the mean position of all points within the rectangle of n2. The position of the upper left corner of the node is the position of the node because of convention of the coordinate system. The mean position of a rectangle, however, is the same independent of the direction of the y axis of the coordinate system.

A visualization of the hitboxes that are the result of using the mean for the current index calculation can be seen in **Figure 5.4**.

5.4 Edge Routing

The current edge routing is quite simplistic, which suffices in most cases. However, it can quite often, in real world use cases, result in overlaps between edges and nodes, reducing the readability of the graph depending on how much information is visualized inside the nodes.

One solution for this problem would be to implement the orthogonal edge routing that the layered layout algorithm already uses into MrTree. While that would solve the problem of edge node overlaps,



(a) Wide edge collision in MrTrees basic edge router

(b) Prevented wide edge collision in the AvoidOverlap edge router

Figure 5.5. Examples of wide edge collision in both edge routers

it would mean that the resulting graphs would look similar to graphs layouted by the layered algorithm. Furthermore, trees are graphs that can be layouted using the layered layout algorithm since they are layered graphs too. So there would be no reason to use MrTree if it produced results that would look completely similar. Thus, I did not implement a completely different edge routing algorithm, but augmented the current one by adding bend points in certain scenarios.

Collision checks between the edges and the nodes force the runtime to be in $\mathcal{O}(n \cdot m)$, with n being the number of nodes and m being the number of line segments, because they require checks for line-rectangle collision for every edge line segment and every node. Therefore, this would be computationally expensive for large graphs and impair the interactive usability of the algorithm. Instead bend points are placed generally on easy to check and handle scenarios.

This is a trade off between aesthetics criteria. As for example stated by Purchase [Pur02] for a graph to be considered aesthetic, the edge bends should be minimized, but this routing adds more edge bends. However, depending on the situation, expert users agreed that preventing overlaps between edges and nodes is a more important aesthetics criterion for the graph than the number of bend points. But due to the design choices made earlier, the algorithm will add bend points if there are no overlaps, which will in total reduce the graph drawing aesthetics according to Purchase [Pur02] without benefits. However, the edge routing style is uniform over the whole graph.

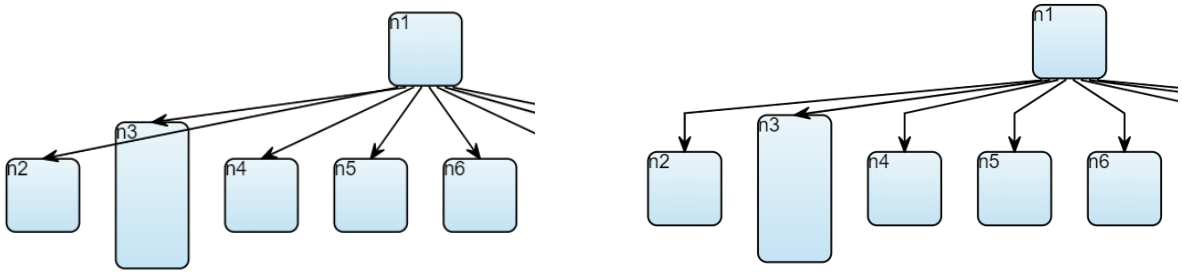
The edge router, which implements the features proposed here, is called the AvoidOverlap edge router.

5.4.1 Wide Edge Collision

Whenever the source and target nodes of an edge are far apart horizontally in their neighboring levels, edges may intersect with other nodes in those levels because edges are routed between the middles of the nodes. An example for this behavior can be seen in **Figure 5.5 (a)**.

A solution for this problem would be to route the edges not from and to the node's middle point but to force the ports of the edges to be at the lower side of the source node and at the upper side of the target node of the edge. However, this would mean that most edges would be routed through the vertices of the rectangle if the nearest point on the nodes side is always chosen. Therefore, the bend points are spread over the available space on the side of the nodes rectangle while not letting the edges start or end near the rectangles vertices, as they are rounded in the default diagram style and bend points that are close to them are moved to match this.

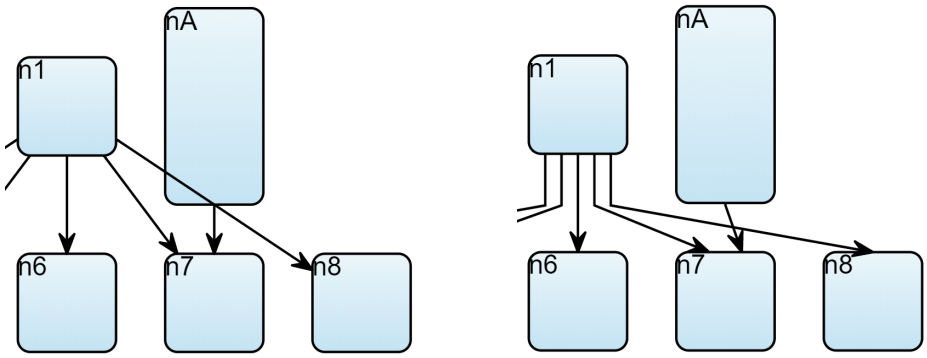
A drawback of this kind of edge routing is that if a node has a lot of outgoing edges those edges can end up being close together, as shown in **Figure 5.5 (b)**.



(a) Size difference edge node collision in MrTrees basic edge router under a level gap

(b) Prevented size difference edge node collision in the AvoidOverlap edge router

Figure 5.6. Examples of size difference edge node collision handling in both edge routers



(a) Size difference edge node collision in MrTrees basic edge router above a level gap

(b) Prevented size difference edge node collision in the AvoidOverlap edge router

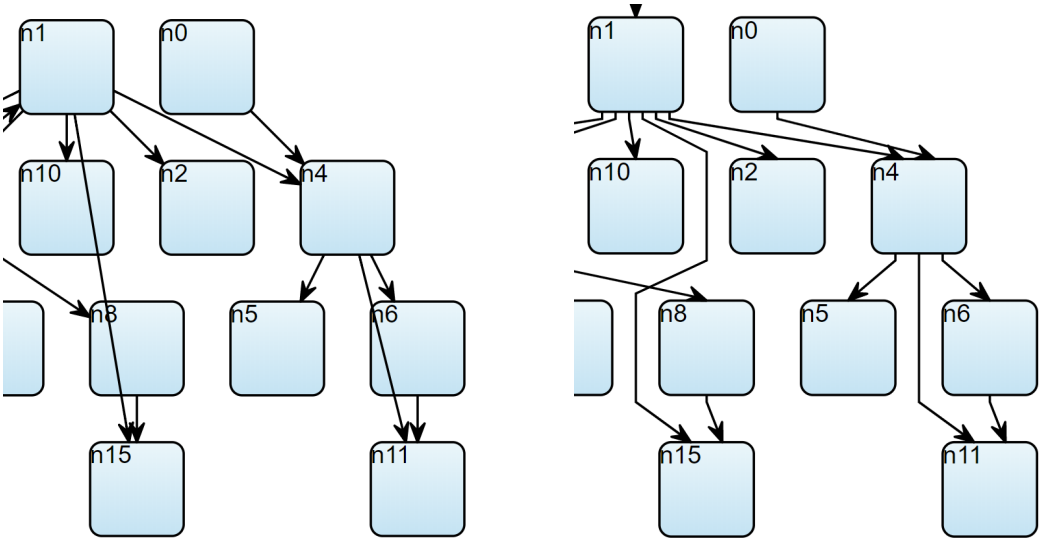
Figure 5.7. Examples of size difference edge node collision handling in both edge routers

5.4.2 Size Difference Edge Collision

However, moving the edge ports on the source and target nodes is not enough to stop basic overlaps. If a node is larger than the node behind it when viewed from the parent node, then there are still overlaps at the bottom of the level gap, as seen in **Figure 5.6 (a)** with `n1` and `n3`. If there is a larger node next to the parent node then there can be overlap between the large node and an edge that is routed far towards the side the big node is relative to the parent node, as seen in **Figure 5.7 (a)**. A solution for this would be to add bend points at the end and start coordinates of the level with the start coordinates of a level being in our case the largest nodes y position and the end coordinates of a level being the largest nodes y position plus its height.

The incoming and outgoing edge's ports should be distributed on both sides of the nodes so that the edges remain distinguishable, as seen under `n1` and above `n7` in **Figure 5.7**.

However, due to the angles at which the edges leave `n1` in **Figure 5.7** edges may end up being routed quite close together near the new bend points.



(a) Multi level edge collision in MrTrees basic edge router

(b) Prevented multi level edge collision in the AvoidOverlap edge router

Figure 5.8. Examples of multi level edge collision handling in both edge routers

5.4.3 Multi Level Edge Collision

Another type of collision that appears when using MrTrees original edge router is multi level edge collision. Multi level edges are edges that do not go from level n to level $n+1$ but to levels greater than $n+1$.

If an edge connects two nodes that are more than one level apart in the tree then the edge will most likely traverse through other nodes, as seen in **Figure 5.8**.

We can solve this collision by finding the right spaces or gaps between nodes and then placing bend points there in the right order. Finding the right node gaps is possible by following the line the edge from the original edge router takes and checking at which X position it is at the height of the middle of the level and then snapping the X position to the closest gap.

For that the algorithm interpolates between the sources x position and the target nodes x position based on the y position of the current level. The gap between two nodes that is the closest to the resulting x value is then chosen as the gap the edge will be routed through, as shown in **Figure 5.9**.

5.4.4 Cycle Inducing Edge Collision

Cycle inducing edges are edges that do not go from level n to level $n+1$ but to levels $< n$. Such an edge is removed by the first phase of MrTree and then added in later, and will go against the flow of the tree.

There are many possible ways to route this edge. For example, it could be routed similar to multi level edges. The end of the edge on the target node could be positioned like the rest of the ports for outgoing edges is positioned on the target node. However, because cycle inducing edges are a breach of the tree definition and because users may want to clearly see which edge is cycle inducing. I chose a different method. Furthermore, within a graph the number of incoming edges is usually more evenly distributed over the nodes than the number of outgoing edges since with the exception of the root

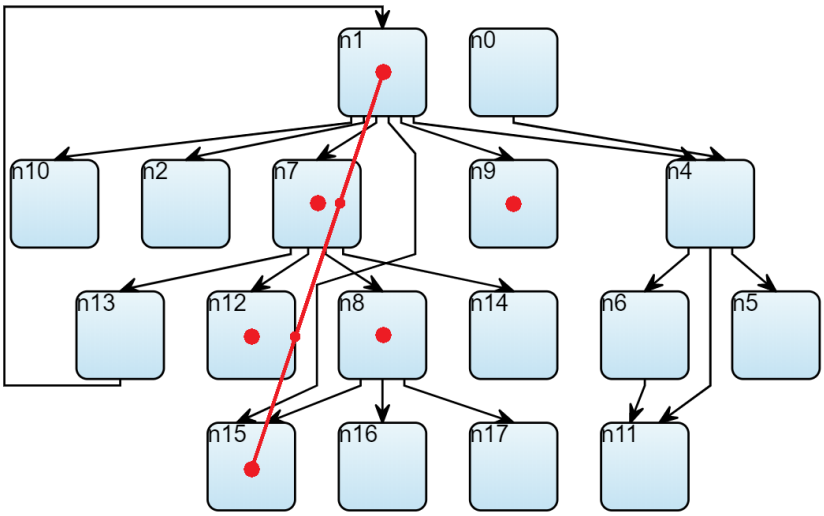


Figure 5.9. Visualization of the node gap choosing of the multi level edge routing

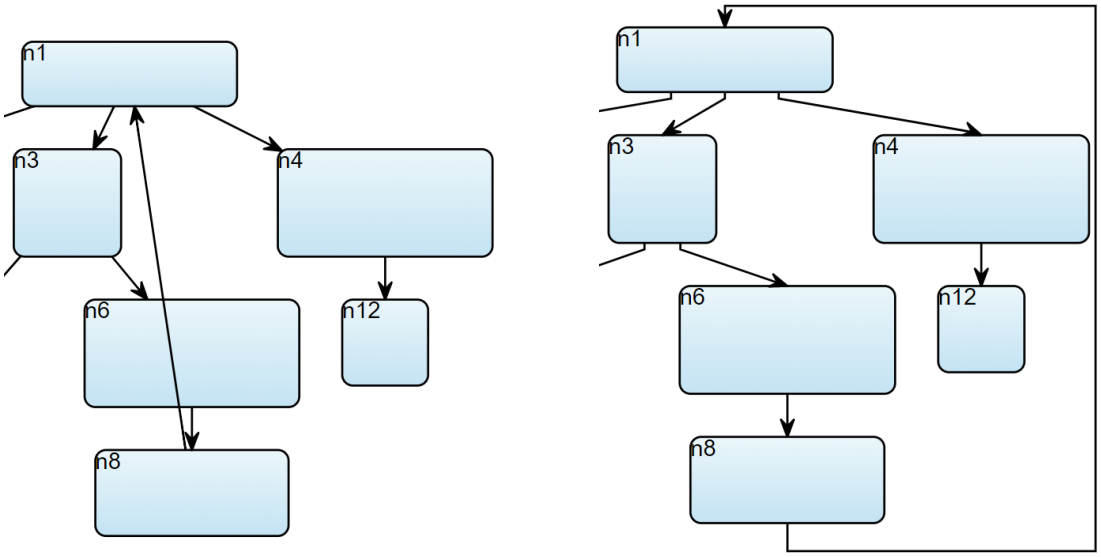
node(s), all nodes have at least one incoming edge, but there are more leaf nodes which do not have any outgoing edges. Having many outgoing edges is easily visualizable as outgoing edges are a thin line, but incoming edges additionally have the arrow head drawn onto them, which means they have to be further apart than incoming edges.

Therefore, the edge is routed around the tree and back up into the target node, as seen on the right side of **Figure 5.10**. This way all incoming edges remain on the top of the node and all outgoing edges remain on the bottom of the node. This maximizes the flow direction of the graph, which is another aesthetics criterion [Pur02]. However, if the cycle inducing edge does not collide with any node, the routing around the graph might seem excessive and the original routing more sensible, as seen in **Figure 5.11**.

To avoid the edge collisions, the number of nodes at the start, end and at the two sides of the graph are counted and new edges are routed with and offset of their normal location based on those counters.

5.4.5 Edge End Texture Collision

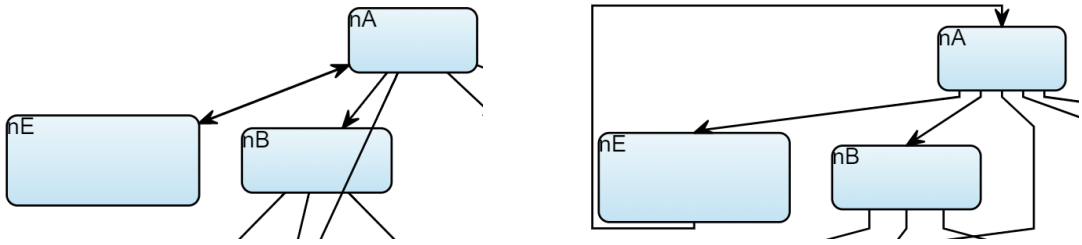
Many representations of the graph draw an arrow head texture at the end of the edge to indicate its directionality. However, those arrow heads, if they are drawn, can overlap with the node, as can be seen in **Figure 5.12 (a)**. Furthermore, the steepness of an edge, coming into or going out of a node, is an aesthetics criterion, with steep edges reducing the diagrams readability as stated by expert users. To fix this, edges with a steep angle between their last line segment and the node side the edge enters have to be identified. Furthermore, a bend point has to be added. Since only the angle of the last line segment counts for the angle the head is drawn in, even if the line segment starts and ends under the head as shown in **Figure 5.13 (b)**, the bend point is added under the arrow head. This way the angle of the head can be changed slightly without changing the rest of the look of the graph. Because the length and the existence of such end heads differs between visualizations of the layouted graph, and because the elk layout process is not informed about the edge heads normally an elk option that controls the positioning of the bendpoint has to be added. The result of those changes can be seen in



(a) Cycle inducing edge collision in MrTrees basic edge router

(b) Prevented cycle inducing edge collision in the AvoidOverlap edge router

Figure 5.10. Examples of cycle inducing edge collision handling in both edge routers



(a) Cycle inducing edge routing in MrTrees basic edge router

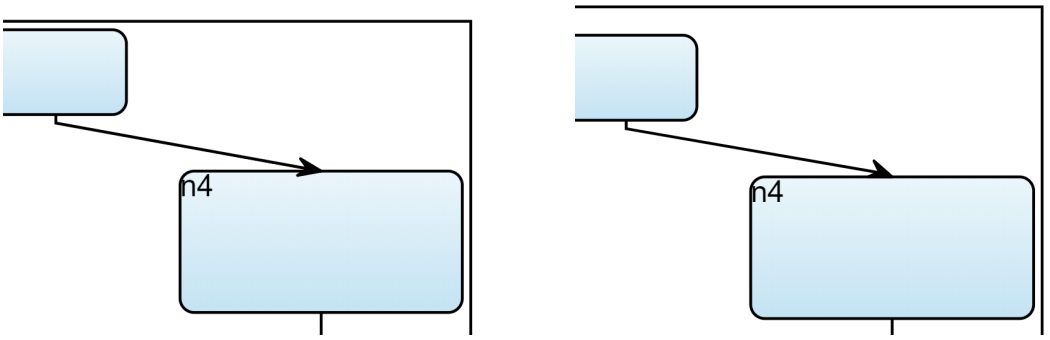
(b) Cycle inducing edge routing in the AvoidOverlap edge router

Figure 5.11. Examples of unnecessary cycle inducing edge collision handling

Figure 5.12 (b).

5.4.6 Hourglass Bendpoint Omitting

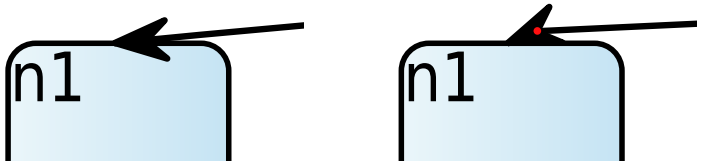
This edge router trades one aesthetics criterion, which is having as few as possible bend points, for another, which is having no overlaps. However, it is important to identify algorithmically easy to check situations, in which bend points can always be omitted without reintroducing overlaps. One of those scenarios would be if a multi level edge is routed around the right side of a level that extends quite far to the right to a level that is further left and then to a level that is further right again, as can be seen in **Figure 5.14**. There is no need to route the edge to the side of the inner level. This scenario is easy to check as it is needed to compare coordinates of the sides of the levels, which have to be computed for the edge to be routed around them anyway.



(a) Edge end texture collision without collision handling

(b) Edge end texture collision with collision handling

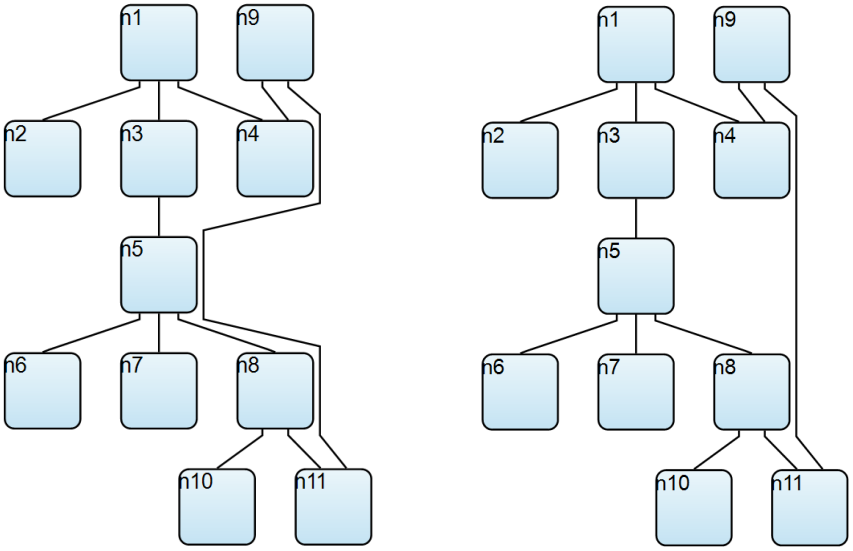
Figure 5.12. Examples of edge end texture collision handling in KEITH



(a) Edge end texture collision without collision handling

(b) Edge end texture collision with collision handling and the inserted bend point visualized as a red circle

Figure 5.13. Visualization of the bend point under the edge end texture



(a) Without bend point avoidance

(b) With bend point avoidance

Figure 5.14. Example of bend point avoidance for multi level edge routing around a hourglass tree contour

5.5 Compaction

Another important aesthetics criterion for a graph is it size. As stated by Di Battista [TDB88], a graph should not waste space and keep the area occupied by the drawing small. To assure that this aesthetics criterion is met, the graph needs to be compacted.

For this feature the *one dimensional compaction* as described in Chapter 4 is used. However, groupings of the tree's nodes have to be considered because a node should never be moved further than below its parent without violating the node spacing option. For example, in **Figure 5.10** the node `n4` should not be moved upwards even though it could be, because moving it further up would impair the readability of the graph as users expect a tree graph drawing that places the child nodes below their parent with some spacing.

The dimension chosen for *one dimensional compaction* here is the layout direction, since trees will most likely be routed downwards and most screens that are used are wider than higher, which means less height is available than width. Compacting the height of the graph can therefore improve the usage of the screen real estate[1].

5.5.1 Node enlargement

Enlarging the nodes by half of the node spacing in each direction when computing the constraint graph using the *scanline algorithm* prevents the compaction algorithm from breaking the node spacing as specified by the user. If this step was skipped a node with its right side slightly left of another node's left side and above it may be compacted next to it, breaking the node spacing. For example, if `n11` was wider in **Figure 5.15 (a)** so that it comes closer to `n6` than allowed by the specified node node spacing, the node would still be compacted downwards into the same spot without node enlargement and would violate the node node spacing set by the user.

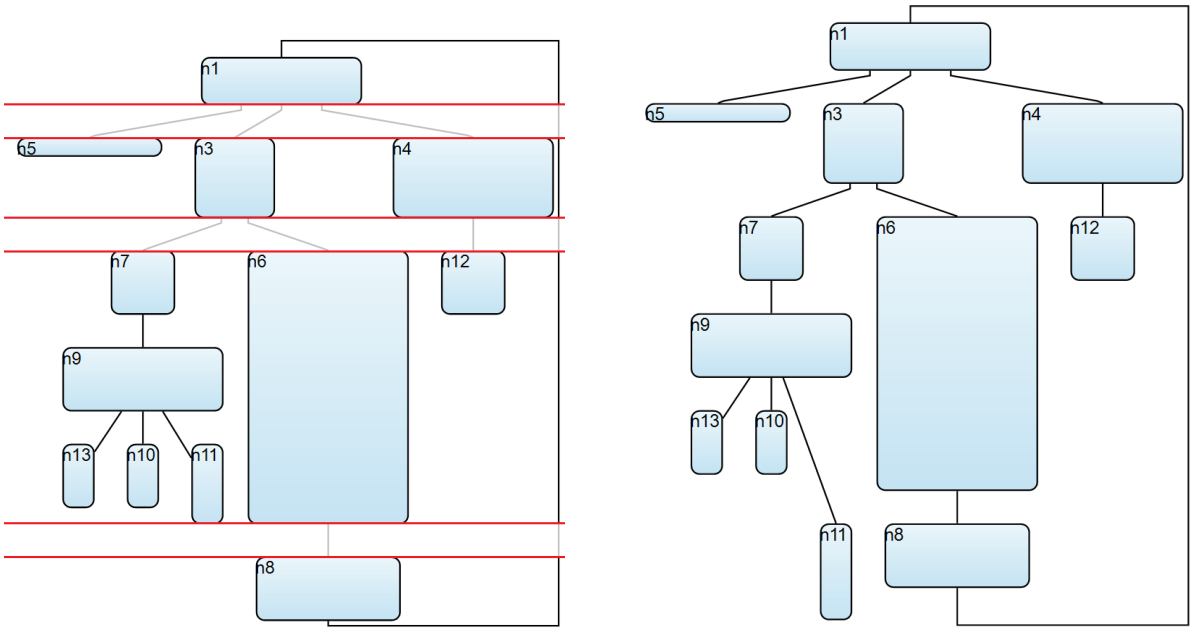
5.5.2 Level preservation

If the AvoidOverlap edge router is used, the nodes have to be kept out of the spaces between the tree levels because the edge router needs those spaces for the edge routing. To achieve this, the algorithm has to check whether the node intersects with the level gaps after it has been moved. If an intersection occurred the node is moved back downwards until it lies completely within one level again. It has to be moved downwards because the node is already as far up in the graph as possible from the compaction that was applied earlier. An example of this can be seen in **Figure 5.15 (a)** with node `n11`. If `n11` was a bit larger and would reach into the level spacing, it is moved one level downwards, as seen in **Figure 5.15 (b)**. It is important to do the node shifting and level preservation together for each node from the top to the bottom of the graph to not move nodes into each other.

When compacting with the AvoidOverlap edge router, nodes that fit into the next level are put into the next level, as seen with node `n9` and its child nodes in **Figure 5.15**, as opposed to splitting the level and assigning node `n6` to multiple levels so that the spacings between those intermediate levels would not be obstructed by `n6` during the edge routing. The layered algorithm had a similar problem with big nodes that span over multiple layers[2][3].

However, the AvoidOverlap edge router will fall back to using the simple edge routing for edges that start and end in the same level because it is not built to handle a graph that is not divided by levels.

[1] `https://www.onlinesolutionsgroup.de/blog/glossar/s/screen-real-estate/`
[2] `https://github.com/eclipse/elk/issues/523`
[3] `https://github.com/eclipse/elk/issues/439`



(a) The level spacings of a tree drawing marked by red rectangles

(b) The graph from **Figure 5.15 (a)** but with a slightly higher n11 node that is forced to be placed into one level under its siblings

Figure 5.15. Visualization of a compacted graphs levels and example of level preservation

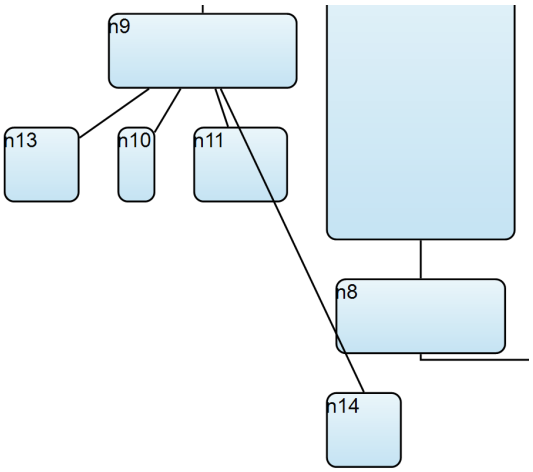


Figure 5.16. Example of problems with the current compaction in combination with the edge routing

The one dimensional compaction does result in problems where sibling nodes might get separated into multiple levels, as seen in **Figure 5.16**. As a fallback the edges going out of `n9` are routed using the basic edge routing.

5.5.3 Circle Placement

The node preview position circles from the child node ordering feature between `n11` and `n10` would end up next to `n8` in **Figure 5.16** if they would be calculated as described earlier. For scenarios as this I have changed the algorithm, which places the circles at the mean positions between the two middles of the facing sides of the nodes, to one that uses the middle y position of one node as the middle y position for the circle. In the case of **Figure 5.16** this would mean that the circle would be placed at the height of `n11` and therefore where the dragged node would be placed at if it would be released at the position of the circle. If the constraint graph was computed without enlarged nodes, as described earlier, circles may even end up inside other nodes. For example if `n11` was wider in **Figure 5.16** the circle between `n11` and `n10` would be placed inside `n8` if the old placement was used. However, the new circle placement is not without flaws either, as it would result in the circles not being placed in the same height among siblings even if all children would be in the same level, compaction was enabled and not all siblings had the same height.

5.6 Cohen–Sutherland based Edge Routing

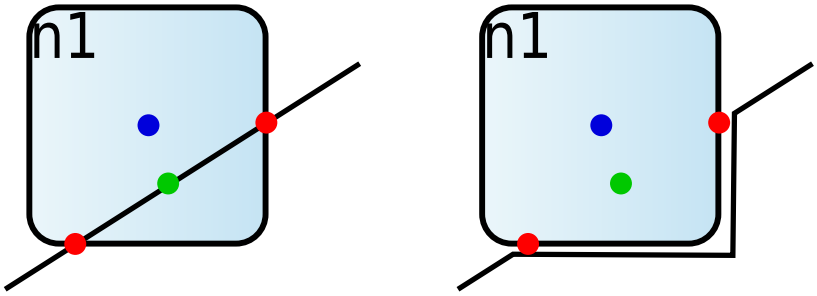
In cases in which the runtime of $\mathcal{O}(n \cdot m)$, with n being the number of nodes and m being the number of line segments, is acceptable an edge router using collision checks is possible. The advantage of such edge routers would be that they only change the edges from their current path if necessary.

One way to implement that concept would be to use the Cohen–Sutherland algorithm [Fol96]. The Cohen–Sutherland algorithm[4] computes the clipping points between lines and Axis Aligned Bounding Boxes (AABB) efficiently using the lines slope-intercept form. In our case the line is a segment of an edge and the AABB is a node's rectangle. Then the clipping points can be used to route the overlapping edge around the node. For that the mean of the clipping points can be checked against the middle of the AABB. If the mean is for example under and right of the middle of the node, as seen in **Figure 5.17**, then the edge can be routed around the bottom right vertex of the nodes rectangle. A special case would be if the mean's x or y coordinate was the same as either the middle's x or y coordinate, as seen in **Figure 5.18**. In that case the clipping points are on opposite sides of the rectangle and the edge has to be routed around two of the rectangle's vertices.

Currently three or four bend points are added per collision, depending on the case, but one bend point can be enough in most cases, as shown in **Figure 5.19**. However, setting just one bend point does change the path of the rest of the edge, which means the two new line segments would have to be checked for collision again.

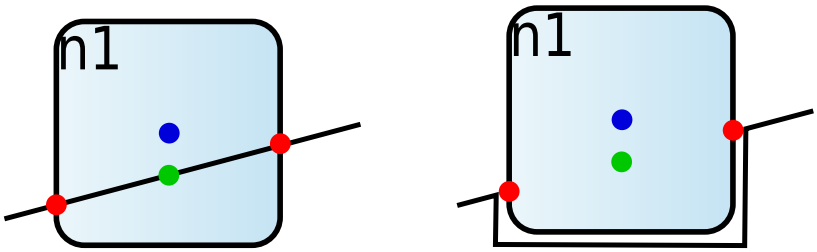
Because this technique works without using any tree-specific properties of the graph it could be used on other types of graphs.

[4] `https://en.wikipedia.org/wiki/Cohen-Sutherland_algorithm`



(a) An edge node collision **(b)** A resolved edge node collision

Figure 5.17. Illustrations of the proposed edge routing with the red points being the clipping points calculated by the Cohen–Sutherland algorithm, the green point being the mean of the two clipping points and the blue points being the middle of the node



(a) An edge node collision **(b)** A resolved edge node collision

Figure 5.18. Illustrations of the proposed edge routing with the red points being the clipping points calculated by the Cohen–Sutherland algorithm, the green point being the mean of the two clipping points and the blue points being the middle of the node

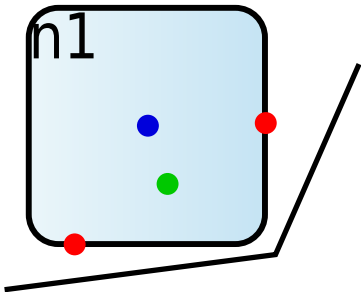


Figure 5.19. Simpler collision handling of the case from **Figure 5.17**

Chapter 6

Implementation

The last chapter mostly focused on the concept of the changes to MrTree and what was added to KEITH to achieve interactivity. This chapter is about how those concepts were implemented based on the technology described in earlier chapters. The main three areas that were modified are KEITH, which is based on the Typescript language, KLighD's LSP, which uses the xtend language, and the layout algorithm MrTree within ELK, which is written in Java.

6.1 Software Architecture

The new overall phase structure of MrTree after the implementation of all features is visualized in **Figure 6.1**.

Moving the edge routing code from the graph export into the edge routing phase does result in a problem with the positioning of the bend point coordinates of the edges and the node positions.

When the edge routing was done during the graph export, the nodes were moved to their final positions while converting them to `ElkNodes`. Note that only the nodes were moved, so the edges were routed on the already moved `ElkNodes`. If the edge routing is now done in the edge routing phase, the nodes are moved without their corresponding bend points during the graph export, which

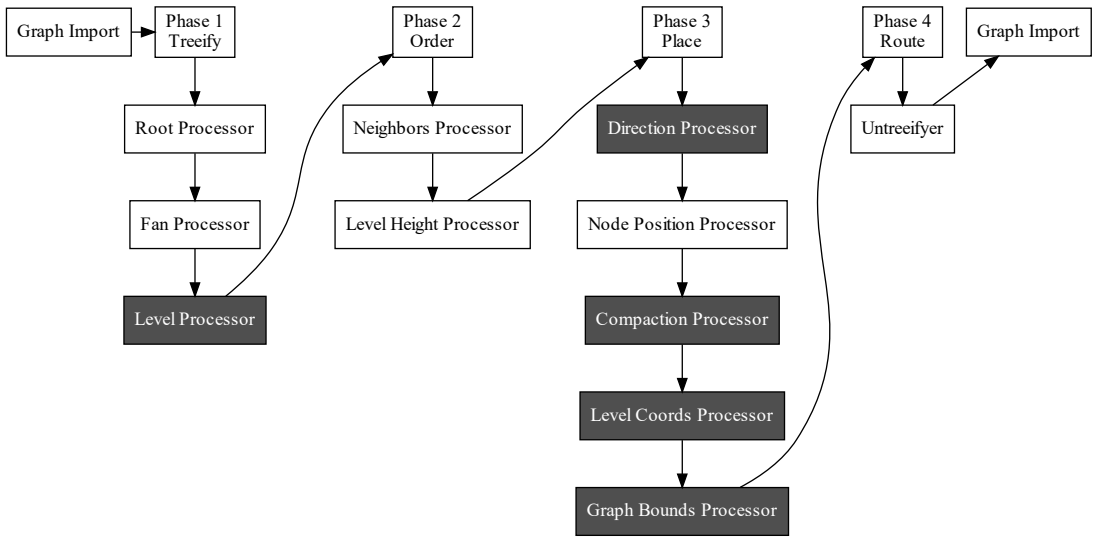


Figure 6.1. The phases of the MrTree Algorithm after the implementation

results in an offset between the coordinates for the bend points specified on the edge routing phase and the node coordinates. The edges will therefore not be positioned within the graph as specified within the edge routing phase. To fix this, the nodes and the edges have to be moved by the same offset. The `Components Processor`, which does move the edges of graph components when packing the components, already defines a method which moves nodes and edges by the same offset. Because of that and because it is sensible to apply the graph's padding in a phase that is about positioning the graphs components, the application of padding was moved to the `Components Processor`.

6.2 Layout Direction

ELK already defines a layout direction option using the `Direction` enum, which contains the following set instances: $\{DOWN, LEFT, RIGHT, UP\}$. I have implemented his feature by adding a `Direction Processor` intermediate layout phase to MrTree after the third layout phase, in which the nodes are placed, and specifically before the `NodePosition Processor` intermediate phase, as can been in **Figure 6.1**. The edges are routed on the final node positions here. The reason why the `Direction Processor` was placed before the `NodePosition Processor` in the `Intermediate Processor Strategy` enum is that before the `NodePosition Processor` the node positions are not actually the positions of the upper left corner of the node but for the middle of the node. This transformation is done in the `Node Position Processor`. If the layout direction is applied before that the node's size does not have to be taken into consideration, and the position variables can just be swapped or inverted or both to apply the desired layout directions.

6.3 Child Node Ordering

This feature was implemented as a new `OrderWeighting` option. To ensure that old definitions for MrTree would still result in the same tree when layouted with the new algorithm, the already existing `OrderWeighting` options that defined how the child nodes should be sorted were not changed semantically.

As described in Chapter 5 the already defined `OrderWeighting` option from MrTree is used to implement this feature. A new weighting mode that allows defining the index within the list of node siblings the current node should be moved to is defined.Then GUI interactions are added to KEITH that allow us to update the textual definition of the graph interactively.

6.3.1 MrTree Constraint-based OrderWeighting

Firstly a way to order siblings in MrTree is implemented. In the original MrTree algorithm two classes are within the package of the second phase as described in Chapter 4. This feature is implemented on the class *NodeOrderer* and the CONSTRAINT object is added to the `OrderWeighting` enum, which if activated sorts sibling nodes as described in Chapter 5.

The new sorting of the nodes is then applied by sorting the *outgoingEdges* list of the parent node of the siblings. The node placer uses that sorting in the third phase to position the nodes.

In addition to the constraint option, the DESCENDANTS `OrderWeighting` option is implemented into the `NodeOrderer` class as it was implemented in the not anymore functional `OrderBalance` class. The DESCENDANTS option sorts the nodes based on the number of nodes in their subtree. The option was still available in the `melk` file but was not implemented as described in Chapter 3.

MrTree now supports four ORDERWEIGHTING options in total:

1. NONE, which does not change the ordering. Using this option will result in the edges in the outgoing list being sorted in the order they were defined in in the graph definition. Because this option does not do anything it is the fastest to execute.

2. DESCENDANTS, which was reimplemented and sorts nodes based on the number of nodes in their subtree.

3. FAN, which was already defined in `NodeOrderer` and sorts the nodes by the maximal number of child nodes that are in the same level.

4. CONSTRAINT, which sorts the nodes using their position constraint as described in Chapter 5.

6.3.2 Interactivity - KEITH

The rest of the implementation for this feature works quite analogously to the implementation of the position and layer constraints by Petzold [Pet19] and Schönberner [Sch19].

First the GUI for this feature has to be defined by adding code to the `keith-interactive` module of KEITH. Most importantly a `renderHierarchyLevel` function has to be defined for MrTree, which is called when the user drags a node and can be used to draw SVGs to the sprotty diagram, as well as a `setTreeProperties` function, which sends information about the drag and drop action of the user to the language server once the user lets go of the left mouse button (LMB). The payload, which is sent to the language server, contains the node ID of the dragged node and the new position index of the node.

Visualization

The node preview position circles and the dashed rectangles are drawn onto the sprotty diagram of the finished graph layout using the `renderHierarchyLevel` function in the `tree interactive view` that was defined in KEITH. The returned HTML tag <g> contains the SVG data that is drawn. The tag is defined at the start as `let result = <g></g>` and appended to by wrapping the current `result` into another <g> tag together with the new object as shown in Listing 6.1. This way the returned SVG is always valid even if nothing is added to it.

```
result = <g>{result}<rect
         x={x}
         y={y}
         width={width}
         height={height}
         stroke={color}
         fill= 'rgba(0,0,0,0)'
         strokeWidth={2 * boundingBoxMargin}
         style={{ 'stroke-dasharray': 4 } as React.CSSProperties}>
     </rect></g>
```

Listing 6.1. Appending the svg tag

In addition to placing the circles at the right positions, a dashed rectangle is drawn around the siblings of the dragged node to visualize that only those nodes can be resorted.

Setting Tree Properties

To send the language server what position the node was moved to, the KEITH client has to first find out what the new position is.

For that the client has to compute the siblings of the dragged node using the `getSiblings` function, which computes the same siblings of a node as the MrTree algorithm to keep the indices consistent. In both cases the primary parent of the node is chosen by following the first incoming edge in the `incomingEdges` list of the dragged node to its source. The siblings can then be calculated by filtering the list of all nodes for nodes that have the same primary parent. After that the siblings are sorted based on the current layout direction and the new position of the dragged node is sent to the language server as a `TreeSetPositionConstraintAction` containing an identifier for the dragged node and the new position within the dragged nodes siblings.

6.3.3 Interactivity - LSP

The second step is to implement a `MrTreeInteractiveLanguageServerExtension` into the LSP so that it knows what to do with the payloads sent from KEITH. First the `TreeSetPosition ConstraintAction` is received and the dragged node is selected from the diagram using the ID from the action. Then the siblings of that node are calculated using the `getSiblings` function, which returns the same siblings as the method in MrTree. Those siblings are sorted by their x position or y position depending on whether the current layout direction is vertical or horizontal. The dragged node is put to its new index, and finally position constraints are assigned to all siblings based on their new indices.

6.4 Avoid Overlaps in Edge Routing

Here the implementation of the edge routing concepts from Chapter 5 is described. All features of this section are implemented with the layout direction in mind because of the architectural decisions made earlier.

6.4.1 Incoming and Outgoing Edges

To place the bend points as described in Chapter 5 all nodes are visited. For each node to each edge in its `outgoingEdges`, two new bend points are appended that are the edges start points. Two new edge end bend points are appended to each edge in the `incomingEdges` list.

6.4.2 Multi Level Edges

Here the concept from Section 5.4.3 is implemented. However, the problem of having multiple edges being routed through the same gap and therefore the same position resulting in edge edge overlap has to be handled. The edges have to be counted and spread evenly throughout the gap while making sure they are sorted correctly so they do not intersect unnecessarily. This problem is solved by adding the `MultiLevelEdgeNodeNodeGap` class and a `HashMap`, which takes the index of the gap within the level and the level number itself as a key, and contains values of the type `MultiLevelEdgeNodeNodeGap`. If the first edge is routed through a gap, a new `MultiLevelEdgeNodeNodeGap` object is created and added to the `HashMap` for the gaps level and index. All the `MultiLevelEdgeNodeNodeGap` constructor gets here are two references to both the bend points of the edge and references to the two nodes that are left and right of the gap. If the gap is at the very left or very right side of the level, either the left or right node

will be `null`. The `MultiLevelEdgeNodeNodeGap` object places the two bend points. If another edge is then routed through the same gap, the gap will get the same key because it consists of the same index and level and the `HashMap` therefore returns our `MultiLevelEdgeNodeNodeGap` for the gap. Using the method `addBendPoints` of `MultiLevelEdgeNodeNodeGap` the algorithm then adds references to the two new bend points of the edge, which should be routed through the gap, and the `MultiLevelEdgeNodeNodeGap` object updates their positions. If the gap is between two nodes, the bend points are spread evenly between them to use the space most efficiently. If the the gap is next to the last or first nodes of the level, the bend points are stacked next to them and spaced by the edge node padding. However, if bend points are routed around the very left or right side of the graph, they have to be counted to make sure they do not collide with the cycle inducing edges that are routed along the sides of the graph drawing too.

This feature was implemented in this way because it keeps the code, which is targeted to do a specific job, in the phase that job belongs to. Adding dummy nodes in earlier phases, through which the edges may be routed, would have been another possible way to implement this feature.

6.5 Compaction

Compaction is implemented as an intermediate layout phase, which is positioned before the `Level Coords Processor` and `Graph Bounds Processor` as shown in **Figure 6.1**, so that it can use the final node positions from the third phase. The `Compaction Processor` is placed after the `Node Position Processor` so that available position values here are the positions on the top left of the node, which is useful here as the sides of the node are more important here as the middle position of the node, which we would have access to before the `Node Position Processor`.

To apply level preservation, the beginning and end coordinates of the levels along the layout direction have to be saved into an array of `Integer Pairs` before moving any nodes. During downward layouting the beginning and end coordinates of a level would refer to the highest and lowest y coordinate that a corner of a node occupies in the drawing.

After that is done the scanline algorithm [Len90] [Rüe18] can be invoked on the graph as described in Chapter 4. The points list, as seen in the pseudo code, is here sorted ascendingly based on the resulting value of the dot product of the point and the direction vector, where the direction vector is a unit vector that points in the currently set layout direction.

Using this projection is a way to check the coordinate of a point in any layout direction without writing code for each direction. The *cand* array from the pseudo code is implemented as a `HashMap` because Java's arrays only accept integer indices and *S* is implemented as a `SortedSet` again using the projection onto the direction vector as the sorting function. After defining these variables the code is implemented without many deviations from the pseudo code. However, it is important to check if the left or right elements in *S* exist before attempting to use them in the else case. The constraints *C* are saved into the internal property COMPACT_CONSTRAINTS.

Chapter 7

Evaluation

To evaluate how understandable the new UI is and to check if there are more useful features that could be implemented, I have shown the project to members of the Real-Time and Embedded Systems Group of Kiel University during development. People with prior knowledge about graphs were interviewed at the end of the development.

Personal interviews as they have been conducted in prior works were not possible due to the outbreak of the novel corona virus this year.

7.1 Expert Feedback

While the UI was in development it was shown to the Real-Time and Embedded Systems Group to check if features were implemented sufficiently well and to answer questions.

One edge case were nodes with multiple parents. Because the child node ordering feature is based on being able to sort siblings of a node it is important that MrTree, the KIELER language server and KEITH all had the same understanding of the range of a siblings a node may be sorted within. If that node had multiple parents, it would per definition be part of multiple sibling groups. However, there is only one position constraint index. It would be possible to confine nodes with two parents to be in the middle of both sibling groups, acting as its own sibling node group if multiple nodes had the same two or more parents. A primary or real parent for the node could be chosen so that it ignores all other parents when being ordered. However, it is still not clear how the primary parents of nodes should be chosen. The working group recommended using the first parent of the node in the list of nodes that the three different parts of this work get, which is the order in which the nodes were defined in the `elkt` file.

One occurrence in the AvoidOverlap edge router that was criticized was that while the routing algorithm mostly avoided edge node overlap, the arrow heads at the end of the edges could still overlap with the nodes and that the edges may enter nodes at a steep angle. A concept for a fix to that problem was shown in Chapter 5.

Another part of the UI that was criticized was the lack of features that marked the nodes that could be sorted because it may not always be clear which nodes may be swapped and which are part of different sibling groups, possibly due to having multiple parent nodes.

7.2 User Study

The interviews were held with 10 people, including 4 university faculty and 6 students. The interviews consists of a tutorial and two experiments on two different ElkGraphs with two tasks each for the participants. As described earlier the interviews were held entirely using Discord Screen Share[1] due to the outbreak of the corona virus. The interviewees would download the electron build of our KEITH

[1] `https://support.discord.com/hc/de/articles/360040816151-Share-your-screen-with-Go-Live-Screen-Share`

version for their operating system. The supported platforms were Windows and Linux based operating systems. I sent the participants a zip archive containing the three `elkt` files, in which the three graphs are defined, and which the experiments were conducted on. One problem, which was encountered during the experiments, was that some interviewees did not have a java runtime or not the right java runtime installed for the language server of the electron build to work properly. However, after installing the right JRE and updating the PATH variable so that the newly installed JRE gets called, the language server within the KEITH build worked and the graph could be drawn.

Due to the low count of participants I mostly focused on direct verbal feedback. However, a short survey sheet was still send to the participants to get general feedback. The sheet was inspired by Masoodian *et al.* [MLB+15]. Three questions on the sheet may be answered by the users using a 5-point Likert scale with different labels for each question. The final question asks which edge routing mode from which file made it easier to read the graph. In the paper two new diagram types for the visualization of electricity usage were tested in a user study similarly to visualization of the MiddleToMiddle edge router and the AvoidOverlap edge router that are being tested for their readability. Furthermore, the number of participants is similar to the user evaluation conducted by Masoodian *et al.* [MLB+15] too. Therefore, the survey sheet was designed similarly, while removing Table I, since it is hard to design concrete data to fill the tree with, and shortening the remaining tables. The questions were translated to German since German was the native language of all participants. Interviewees would get the sheet as a `.png` file and fill it out using a graphics editing program such as GIMP or Paint. The participants were split in two groups. One group had compaction activated on the graphs they were working on, and one group did not have the compaction feature activated. This was done to potentially identify differences in feedback. Both groups had the same size of 5 participants. After the survey sheet was filled out I had some free conversations with some of the participants and for example showed them the differences between enabled and disabled compaction and got additional feedback.

7.2.1 Tutorial

The interviews start with a simple graph from the file `a_tutorial.elkt` shown in Listing 9.1. The letters in front of the filename indicate the order in which the files should be opened, since most filesystems automatically sort them by name, and the participants should see the graphs in the same order every time. No experiments were conducted on this file, this simple graph's purpose is entirely to check whether the basic child node ordering UI operations are intuitive and to ensure the participants understand the concept of child node ordering.

7.2.2 AvoidOverlap

The second file, which is called `b_tasks.elkt`, contains a larger and more complex graph, which is routed using the AvoidOverlap edge router, as shown in Listing 9.2. The first task the participants would have to complete on this graph would be to order the nodes by the number in their labels to check whether the UI interaction is still intuitive on larger graphs. After that users were given a more abstract task. They should try to maximize the symmetry of the contour of the tree as much as possible. To work on this task one has to think about the hierarchy of the graph and what moving nodes in the upper levels would mean for nodes in lower layers since they a part of a subtree of a upper node, which is moved with the node if it gets reordered. The solutions the users came up with in this task differed widely, but users had to interact with and think about the whole graph in this task, not just neighboring nodes.

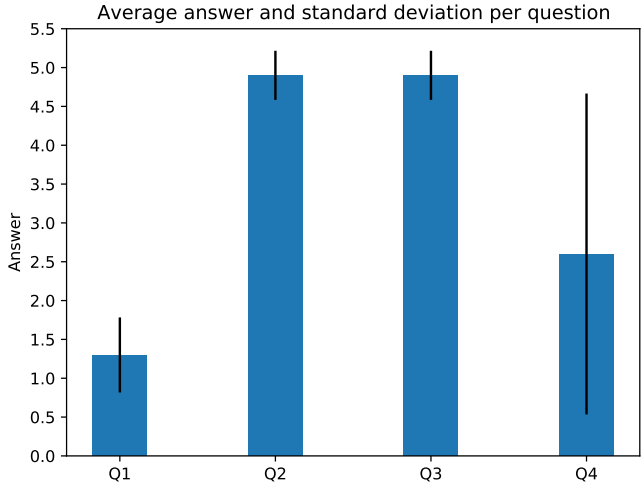


Figure 7.1. A bar chart of the results of the user evaluation

7.2.3 MiddleToMiddle

The third file, which is called `c_tasks.elkt`, contains a graph similar to the one in `b_tasks.elkt`, as shown in Listing 9.3. In fact they are the same, apart from node labels and initial ordering. However, all participants that were asked whether they noticed similarities between the graphs did not notice that both graphs were the same. The purpose of this file is to check whether it is easier for the users to work on the graph using the MiddleToMiddle edge router. Therefore, they had to complete the same tasks on this graph as on `b_tasks.elkt`. However, instead of ordering the nodes by their numbers they were ordered by their letters here as the labels are different to make the graphs seem different. The second tasks purpose here is to force the participants to look at the edges of the graph to identify subtrees to predict which nodes will move with the dragged node.

7.2.4 Results

The average rating and the standard deviation is calculated for the questions.

The first question was "The UI was easy to use" (translated to German) and may be answered using a 5-point Likert scale (anchors 1: strongly agree and 5: strongly disagree). The average result here was a 1.3 with a sample standard deviation of 0.48.

The second question was "The UI was unnecessarily complex" and may be answered using a 5-point Likert scale (anchors 1: strongly agree and 5: strongly disagree). The average result here was a 4.9 with a sample standard deviation of 0.32.

The third question was "I had to learn a lot before I could use this UI" and may be answered using a 5-point Likert scale (anchors 1: strongly agree and 5: strongly disagree). The average result here was a 4.9 with a sample standard deviation of 0.32.

The fourth question was "Which Edge Routing mode made it easier to read the graph" and could be answered by two options, either the MiddleToMiddle edge router or the AvoidOverlap edge router. Here 60% of participants chose the AvoidOverlap edge router over the MiddleToMiddle one. However, during the open feedback most interviewees stated that it depends on the graph the edge routing modes were used on. Most participants stated that on a larger graph with a lot of edge node overlap, the MiddleToMiddle edge router would probably result in a more readable graph, while on a normal

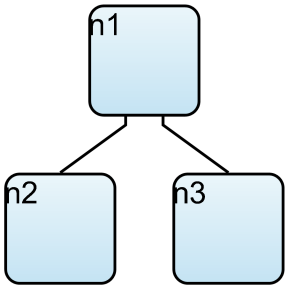


Figure 7.2. A graph that does not change its size if compaction is applied

tree graph without multi level edges the MiddleToMiddle edge router would produce more readable results. Specifically the routing of the cycle inducing edges was considered to be excessive in a scenario such as in the one in the graphs of the user evaluation. However, participants stated that it would seem more reasonable if the middle to middle routing resulted in overlap.

Additional feedback from one interviewee was that the level preservation during the compaction process actually improves the graph's readability and that they prefer the edge routing with the AvoidOverlap router for that reason. Level preservation was not actually built to improve readability, but to let the compaction and the new edge router work together. Nevertheless it does seem to satisfy an aesthetics criterion. Based on the feedback the grouping and alignment aspects seem to be most important as all nodes at the top of a level get the same y position and even nodes that that were pushed into the level during the compaction process got pushed clearly into the visible boundaries of the level. As levels are an important part of the structure of trees, keeping the spacings between levels may be an aesthetics criterion that is more important than the compactness of the graph.

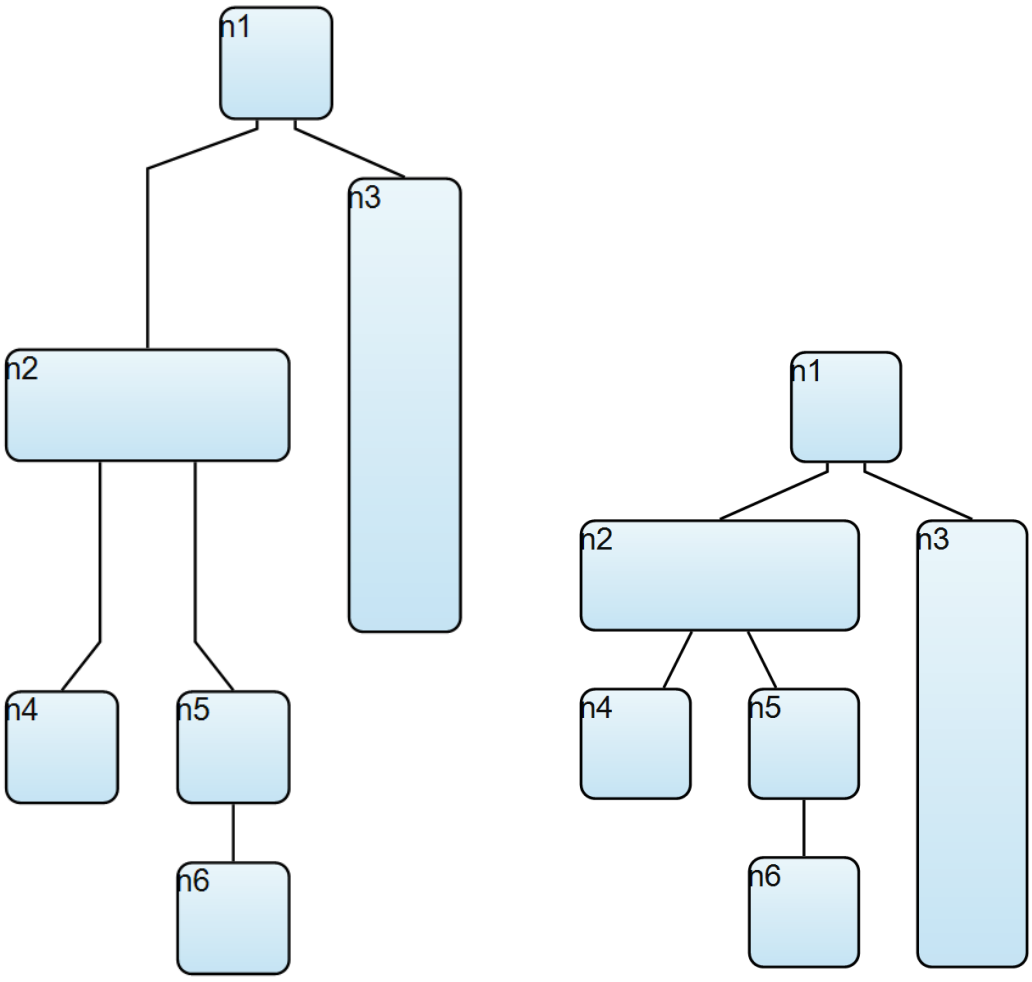
When splitting the entire dataset into the two groups with and without compaction enabled, the average answers to the first three questions do not change significantly. However, whether compaction was enabled or not did significantly correlate with the answers of the fourth question. A possible explanation for this is that most users that were tested with compaction enabled were from the university faculty. They may have voted more for the AvoidOverlap edge router because they may have seen more instances, in which such overlap made it significantly harder to read the graph or because they know which one of the algorithms is new.

All in all the feedback on the first three questions was generally positive, so the adaption of the concepts from Petzold [Pet19] and Schönberner [Sch19] worked on MrTree, while the feedback on the edge router was that it depends on whether the graphs drawing using the MiddleToMiddle router has overlaps or not. Finally the feedback to compaction during some of the post interview conversations were slightly positive depending on how much it actually shrank the graph down.

7.3 Compaction

The amount with which a graph drawing is shrunk during compaction differs a lot depending on the graph. Simple graphs such as the on in **Figure 7.2** do not change at all, since one dimensional compaction cannot move any nodes. However, if there is a large difference between the height of two nodes and the smaller node has small child nodes, then compaction can shrink the graph down depending on the size of the larger node and the size of the subtree under the smaller node, as seen in **Figure 7.3**.

To measure the size difference of the graph the height is measured, since only the height can



(a) layout without one dimensional compaction **(b)** layout with one dimensional compaction

Figure 7.3. An example graph layouted with and without one dimensional compaction

change from one dimensional compaction applied in the downward direction. The height used here is taken from the `Graph Bounds Processor` as seen in **Figure 6.1**, which computes the max and min x and y values of the graph based on the nodes positions and sizes in the graph. The height value is calculated by subtracting the `ymin` from the `ymax` and read by using Eclipse debug tools. Therefore, the height is measured in elk coordinates.

The resulting height of the graph shown in **Figure 7.3** without applying compaction is 340.0 and 220.0 once compaction was applied. That is a height reduction of 35%.

The amount of the height reduction in this example depends on the height of `n3` and the height of highest subtree under `n2`. As long as the total height of the subtree that has `n2` as its root is lower or equal to the height of `n3` level preservation will keep all nodes in the same level. Since the height of `n3` and the height of the subtrees under `n2` can be arbitrarily chosen, a higher height reduction is possible for this graph constellation.

Let hnX be the height of node X with $X \in \mathbb{N}$ and let nns be the node node spacing, which is here

set to the default value of 20, and let *hs* be the height of the highest subtree under `n2`. The height of **Figure 7.3 (a)** can be calculated using $hn1 + nns + hn3 + nns + hs$, and the height of **Figure 7.3 (b)** using $hn1 + nns + hn3$. Since *hs* can not be higher than $hn3 - hn2 - nns$ without triggering level preservation during compaction, it is set to $hn3 - hn2 - nns$ here. The height reduction can now be calculated by $1 - \frac{hn1+nns+hn3}{hn1+nns+hn3+nns+hn3-hn2-nns}$. Decreasing the size of `n3` results in smaller *hs*, which means the height reduction is smaller. Therefore, `n3` needs to be increased to find the maximal possible height reduction for this graph.

$$\lim_{hn3 \to \infty} 1 - \frac{hn1 + nns + hn3}{hn1 + nns + hn3 + nns + hn3 - hn2 - nns} = 0.5$$

This graph can be compacted by up to 50%. However, to estimate the average graph height reduction across graphs used in real world applications, real world examples are needed.

Chapter 8

Conclusion

This thesis is about augmenting the MrTree algorithm with new features. The interactive concept that Petzold [Pet19] and Schönberner [Sch19] developed is implemented on MrTrees *NodeOrderer*, adding a new edge routing option and adding compaction to MrTree.

8.1 Summary

A ElkGraph representation defines almost all important relations that nodes in a graph posses with other nodes. However, sometimes it may be beneficial to sort the child nodes of a node in a certain way to improve the readability of the graph. For this purpose I have reworked `NodeOrdering` and added the constraint option, which allows us to order the nodes using a position constraint option on each node. A UI was implemented into KEITH that allows to interactively set the position constraints mostly as it is defined by Petzold [Pet19] and Schönberner [Sch19], with the addition that a dashed rectangle is drawn around the siblings of the dragged node to visualize the range in which the user may order the nodes instead of around the entire level because of the limitations here.

Furthermore, I have implemented the layout direction option into MrTree to allow it to be as flexible as the layered algorithm in that regard.

I implemented a new edge routing solution, which for performance reasons does not check for individual collision, but rather uses general edge routing techniques to prevent overlaps from happening.

Finally the one dimensional compaction using the scanline algorithm [Rüe18] was added to make sure the graph is not larger and therefore less readable that it needs to be. Furthermore, the compaction should not interfere with the edge routing so the original level structure of the tree had to be preserved without impairing the compaction of the graph.

8.2 Future Work

For truly representative results from the user study more participants are needed. It would additionally be beneficial to add more questions to the survey, as Masoodian *et al.* [MLB+15] did it, in regard to the two edge routing modi and to let them be answered using a Likert scale instead of a binary option.

To evaluate the effectiveness of compaction, a large dataset of used tree graphs could be compiled and the height decrease measured.

A concept for edge cases in the interaction between the one dimensional compaction and the AvoidOverlap edge router should be developed too, as the compactor may spread sibling nodes over multiple levels if there is a lot of space on one side and none at the other. Such a scenario makes it difficult for an edge router such as the AvoidOverlap edge router to function properly.

Either the compactor or the edge router could be revised. The one dimensional compaction's upside is its fast runtime. However, it results in less desirable node constellations as described earlier.

Two dimensional compactors have a much higher runtime and asymptotic behavior [Len90] but it may be possible to consistently find alignments for the child nodes of a node that make sure those child nodes all end up at the same level after one dimensional compaction.

Alternatively, the AvoidOverlap edge router could be augmented in a way that allows it to route edges between and through nodes of the same level in a more consistent way. If the levels that may contain multiple nodes of different hierarchical levels as MrTree keeps track of them could be split into those hierarchical levels, the edge router could work normally on them. However, that would mean that larger nodes would span multiple levels, so the edge router needs to be stopped from routing through those larger nodes.

Alternatively, edges could be routed using an edge router that is based on the Cohen–Sutherland algorithm. This would allow the router to keep the edges as straight as possible, which participants of the user evaluation preferred in the MiddleToMiddle router, and to avoid overlaps at the same time. However, the runtime could prove to be an issue with this solution.

Chapter 9

Appendix

9.1 List of Abbreviations

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KEITH KIEL Environment Integrated in Theia

LSP Language Server Protocol

IDE Integrated Development Environment

ELK Eclipse Layout Kernel

GUI Graphical User Interface

UI User Interface

JSON JavaScript Object Notation

JSON-RPC JavaScript Object Notation - Remote Procedure Call

npm Node Package Manager

SVG Scalable Vector Graphics

UX User Experience

MDE Model Driven Development

KLighD Kieler Light Weight Diagrams

SCCharts Sequential Constructive Statecharts

melk ELK Metadata File

LMB left mouse button

AABB Axis Aligned Bounding Box

9.2 User Evaluation Files

```
algorithm: mrtree
interactiveLayout: true
debugMode: false
weighting: CONSTRAINT
elk.direction: DOWN
edgeRoutingMode: AvoidOverlap
compaction: true

node n1
node n2
node n3

edge n1 -> n2
edge n1 -> n3
```

Listing 9.1. The contents of the `a_tutorial.elkt` file.

```
algorithm: mrtree
interactiveLayout: true
debugMode: false
weighting: CONSTRAINT
elk.direction: DOWN
edgeRoutingMode: AvoidOverlap
compaction: false

node n1 {
   layout [ size: 120, 30 ]
}
node n2 {
   layout [ size: 70, 30 ]
positionConstraint: 3
}
node n3 {
   layout [ size: 40, 40 ]
positionConstraint: 2
}
node n4 {
   layout [ size: 30, 40 ]
positionConstraint: 1
}
node n5 {
   layout [ size: 90, 42 ]
positionConstraint: 0
}
node n6 {
   layout [ size: 90, 40 ]
positionConstraint: 1
}
node n7 {
   layout [ size: 30, 50 ]
positionConstraint: 0
}
node n8 {
   layout [ size: 20, 40 ]
positionConstraint: 2
}
node n9 {
   layout [ size: 50, 140 ]
positionConstraint: 1
}
node n10 {
   layout [ size: 90, 50 ]
positionConstraint: 0
}
node n11 {
   layout [ size: 15, 50 ]
positionConstraint: 2
}
node n12 {
   layout [ size: 25, 70 ]
positionConstraint: 1
}
node n13 {
   layout [ size: 20, 40 ]
positionConstraint: 0
}
node n14 {
   layout [ size: 60, 50 ]
}
node n15 {
   layout [ size: 30, 60 ]
positionConstraint: 1
}
node n16 {
   layout [ size: 90, 40 ]
positionConstraint: 0
}

edge n1 -> n2
edge n1 -> n3
edge n1 -> n4
edge n1 -> n5
edge n1 -> n14

edge n2 -> n6
edge n2 -> n7
edge n2 -> n8

edge n6 -> n10
edge n6 -> n9

edge n10 -> n13
edge n10 -> n11
edge n10 -> n12

edge n8 -> n14

edge n4 -> n15
edge n4 -> n16

edge n5 -> n1
```

Listing 9.2. The contents of the `b_tasks.elkt` file.

```
algorithm: mrtree
interactiveLayout: true
debugMode: false
weighting: CONSTRAINT
elk.direction: DOWN
edgeRoutingMode: MiddleToMiddle
compaction: false

node nRoot{
   layout [ size: 120, 30 ]
}
node nA {
   layout [ size: 70, 30 ]
positionConstraint: 0
}
node nB {
   layout [ size: 40, 40 ]
positionConstraint: 2
}
node nC {
   layout [ size: 30, 40 ]
positionConstraint: 1
}
node nD {
   layout [ size: 90, 42 ]
positionConstraint: 3
}
node nAl {
   layout [ size: 90, 40 ]
positionConstraint: 0
}
node nBl {
   layout [ size: 30, 50 ]
positionConstraint: 2
}
node nCl {
   layout [ size: 20, 40 ]
positionConstraint: 1
}
node nAI {
   layout [ size: 50, 140 ]
positionConstraint: 1
}
node nBI {
   layout [ size: 90, 50 ]
positionConstraint: 0
}
node nX {
   layout [ size: 15, 50 ]
positionConstraint: 2
}
node nY {
   layout [ size: 25, 70 ]
positionConstraint: 1
}
node nZ {
   layout [ size: 20, 40 ]
positionConstraint: 0
}
node nCI {
   layout [ size: 60, 50 ]
}
node nDl {
   layout [ size: 30, 60 ]
positionConstraint: 1
}
node nZl {
   layout [ size: 90, 40 ]
positionConstraint: 0
}

edge nRoot-> nA
edge nRoot-> nB
edge nRoot-> nC
edge nRoot-> nD
//edge nRoot-> nCI

edge nA -> nAl
edge nA -> nBl
edge nA -> nCl

edge nAl -> nBI
edge nAl -> nAI

edge nBI -> nZ
edge nBI -> nX
edge nBI -> nY

edge nCl -> nCI

edge nC -> nDl
edge nC -> nZl

edge nD -> nRoot
```

Listing 9.3. The contents of the `c_tasks.elkt` file.

Bibliography

[BP90] Karl-Friedrich Böhringer and Frances Newbery Paulisch. "Using constraints to achieve stability in automatic graph layout algorithms". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '90. Seattle, Washington, USA: Association for Computing Machinery, 1990, pp. 43–51. ISBN: 0201509326. DOI: 10.1145/97243.97250. URL: https://doi.org/10.1145/97243.97250.

[Cha79] Alfred Shannon Charles Wetherell. "Tidy drawings of trees". In: *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*. Vol. SW-5. 5. 1979. DOI: https://doi.org/10.1109/TSE.1979.234212. URL: https://ieeexplore.ieee.org/document/1702661.

[Dom18] Sören Domrös. "Moving model-driven engineering from eclipse to web technologies". Master Thesis, Department of Computer Science, Kiel University. Nov. 2018. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf.

[Edw81] John S. Tilford Edward M. Reingold. "Tidier drawings of trees". In: *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*. Vol. SE-7. 2. 1981.

[Fol96] James D. Foley. *Computer graphics: principles and practice*. Addison-Wesley Professional, 1996, p. 113.

[Fru91] E. Reingold Fruchterman. "Graph drawing by force-directed placement". In: *Software – Practice and Experience*. Vol. 21. 11. 1991, pp. 1129–1164.

[HM98] Weiqing He and Kim Marriott. "Constrained graph layout". In: *Constraints* 3.4 (Oct. 1998), pp. 289–314. ISSN: 1572-9354. DOI: 10.1023/A:1009771921595. URL: https://doi.org/10.1023/A:1009771921595.

[II90] John Q.Walker II. "A node-positioning algorithm for general trees". In: *Software: Practice and Experience*. Vol. 20. 7. 1990, pp. 685–705. URL: http://www.cs.unc.edu/techreports/89-034.pdf.

[Knu71] D. E. Knuth. "Optimum binary search trees". In: *Acta Informatica*. Vol. 1. 1971, pp. 14–25.

[KPE16] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. "The ide portability problem and its solution in monto". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 152–162. ISBN: 9781450344470. DOI: 10.1145/2997364.2997368. URL: https://doi.org/10.1145/2997364.2997368.

[Len90] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. USA: John Wiley & Sons, Inc., 1990. ISBN: 0471928380.

[LLY06] Yi-Yi Lee, Chun-Cheng Lin, and Hsu-Chun Yen. "Mental map preserving graph drawing using simulated annealing". In: *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation - Volume 60*. APVis '06. Tokyo, Japan: Australian Computer Society, Inc., 2006, pp. 179–188. ISBN: 1920682414.

[MLB+15] Masood Masoodian, Birgit Lugrin, René Bühling, and Elisabeth André. "Visualization support for comparing energy consumption data". In: *19th International Conference on Information Visualisation*. Piscataway Township, New Jersey, Vereinigte Staaten: IEEE, 2015, pp. 28–34. DOI: 10.1109/iV.2015.17.

[Pet19] Jette Petzold. “Intentional layout in sprotty diagrams: defining user interaction”. Bachelor Thesis, Department of Computer Science, Kiel University. Sept. 2019. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jet-bt.pdf.

[Pet95] Marian Petre. “Why looking isn’t always seeing: readership skills and graphical programming”. In: *Commun. ACM* 38.6 (June 1995), pp. 33–44. ISSN: 0001-0782. DOI: 10.1145/203241.203251. URL: https://doi.org/10.1145/203241.203251.

[Pur02] HELEN C. Purchase. “Metrics for graph drawing aesthetics”. In: *Journal of Visual Languages & Computing* 13.5 (2002), pp. 501–516. ISSN: 1045-926X. DOI: https://doi.org/10.1006/jvlc.2002.0232. URL: http://www.sciencedirect.com/science/article/pii/S1045926X02902326.

[Ren18] Niklas Rentz. “Moving transient views from eclipse to web technologies”. Master Thesis, Department of Computer Science, Kiel University. Nov. 2018. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf.

[Rüe18] Ulf Rüegg. *Sugiyama layouts for prescribed drawing areas*. Kiel Computer Science Series 2018/1. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, Kiel University, 2018.

[Sch19] Connor Schönberner. “Intentional layout in sprotty diagrams: reevaluation gesetzter constraints”. Bachelor Thesis, Department of Computer Science, Kiel University. Sept. 2019. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cos-bt.pdf.

[SSv13] C. Schneider, M. Spönemann, and R. von Hanxleden. “Just model! — putting automatic synthesis of node-link-diagrams into practice”. In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 2013, pp. 75–82.

[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 (1981).

[TDB88] R. Tamassia, G. Di Battista, and C. Batini. “Automatic graph drawing and readability of diagrams”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 18.1 (1988), pp. 61–79.