A Straightline Tree Layout with Vertical Position Constraints

Claas Nitschke

Bachelor's Thesis
September 2023

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

Advised by
Maximilian Kasperowski

Selbstständigkeitserklärung

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

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel, 29.09.2023

Claas Nitschke

Abstract

Automatic layouts are an important process for visualizing data structures and software components, especially in large scale software development. Drawing those graphs by hand is not an option, since even small changes can require a redrawing of the whole diagram, making it extremely time-consuming. A common layout approach for trees are layered drawings where nodes of the same level are vertically aligned, like the Reingold–Tilford algorithm or MrTree.

This thesis proposes a new algorithm called *Yconstree* for generating visually pleasing layouts for non-layered trees. The algorithm supports vertical position constraints and can prioritize between straightline drawing and model order. Inspired by a work of v. d. Ploeg, support for straightline drawing is achieved by introducing outlines to simplify the outer shape of subtrees. We discuss the numerous challenges of such a layout and present different approaches of solving them.

As part of the KIELER project, ELK contains a number of different layout algorithms, including drawings for layered trees and other types of graphs. However, when trees include nodes with predetermined vertical positions, possibly outside of those layers, layouts are only possible with non-layered tree drawings. Yconstree was initially developed in python and then implemented into ELK for use in the KIELER project.

Acknowledgements

First I would like to thank Prof. Dr. Reinhard von Hanxleden for giving me the opportunity to write this bachelor's thesis and providing me with constructive feedback.

A special thanks goes to my advisor Maximilian Kasperowski, for giving me the topic of this thesis, introducing me to ELK and providing me with feedback and support for my work. I also want to thank the whole Real-Time and Embedded Systems Group for their feedback and tips.

Finally i want to thank my family and friends for being such a great support, especially while writing this thesis. This would not have been possible without you.

Contents

List of Figures

Acronyms

ELK	Eclipse Layout Kernel
LOL	Left Outer Line
ROL	Right Outer Line
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
RGB	red-green-blue

Chapter 1

Introduction

Trees are a common structure in mathematics and computer science, but also used in a wide variety of other fields. Consisting of *nodes* and *edges* to connect these nodes, the structure mostly referred to as a tree in computer science is an *ordered tree*. In this type of tree with directed edges, the relationship between two nodes is that of a *parent node* and a *child*. Parent nodes are placed above their children, giving the tree an order from top to bottom. Nodes without children are referred to as *leaves*, the top node without a parent is called the *root* of the tree. From decision trees, data visualization, inheritance in many object-oriented programming languages, to optimized data storage for fast searching algorithms, there are many applications of ordered trees. Even expressions in programming languages and logic statements can be visualized as trees consisting of single evaluations.
Ordered trees can be described textually or visually. A textual description can be easier interpreted, edited and maintained by machines. However, since textual representations are not as easy to read for humans, drawing *layouts* of given trees is a typical task when working with those data-structures. This can be done by hand, which works for small trees, but becomes time-consuming and difficult for more complex and bigger structures. Therefore, automatic drawing algorithms for different kinds of trees have been developed for a long time.
Many algorithms use an approach where nodes are only placed on fixed vertical *layeres*, so every child of a node is on the same vertical position. For a recent work on *top-down layouts* in the Real-Time and Embedded Systems Group of the University of Kiel, a tree layout is needed that can not only draw non-layered trees by taking the nodes width and height into account, but where nodes can also have a predetermined fixed vertical position. My goal is to develop an algorithm capable of producing such layouts. For better aesthetics, edges of this layout should not be orthogonal, but mostly straight lines.

Related Work

Layered Trees

In 1979, inspired by a work of Knuth from 1971 [Knu71], Wetherell and Shannon published a paper on tree layout algorithms [WS79]. In their approach, all notes were in distinct layers with static distances between the layers. They were the first to introduce aesthetic criteria for tidy trees, a method, that has since been used and upgraded in many times. Reingold and Tilford iterated on that subject and added a new criterion to make mirror images of the trees produce an exactly mirrored layout [RT81]. The Reingold Tilford algorithm has since been cited in many works and is one of the fundamentals of tree layouts. For tree layouts, ELK currently uses an algorithm called MrTree, which is based on the work of John Q. Walker II [Wal90]. It's a successor to Tilford and Reingold and focused more on the drawing of general m-trees. Still, his algorithm used a layered approach.

Non-Layered Trees

Bloesch solved the problem of non-layered trees by discretizing the layout vertically and thereby transforming the problem back to a layered layout [Blo93], which is then solved using an algorithm by Vaucher [Vau80] or a slightly modified version of Reingold and Tilford. Stein and Benteler used a similar approach by creating bounding boxes for each node and splitting them horizontally and vertically into multiple, layered nodes. A layout can then be made by applying an algorithm for layered trees and afterwards transforming them back into the original non-layered nodes. This thesis is based on the work of van der Ploeg on non-layered trees in linear time. He used threads to shape the outline of subtrees to combine them into bigger trees. To simplify the problem, v. d. Ploeg used orthogonal edges, which lead to very compact trees, but are not always desired. In this thesis, I will modify his algorithm to work for trees with straight edges.

Chapter 2

Preliminaries

This chapter will give an overview about some of the terms and create a better understanding of the requirements for our algorithm.

2.1 Definitions

These definitions are mostly from university lectures or common literature, some however are modified or newly added to fit the given task. We begin with some definitions on graphs in general.

Graph A *graph* $G = (V, E)$ is a set of *vertices* V and a set of *edges* E. Each edge joins a pair of vertices. Vertices are also called *nodes*.

Path A *path* p is a tuple $(v_1 ... v_n) \in V^n$, $v_1 ... v_n \in V$, $n \in \mathbb{N}$ and $(v_i, v_{i+1}) \in E$ for all $1 \leqslant i < n$. If $\forall v_a, v_b \in V$ there is a path $p = (v_a ... v_b)$, the graph is called *connected*.

Cycle A *cycle* is a path $(v_a ... v_b)$ with $v_a = v_b$. A graph is called *cyclic*, if it has at least one cycle and and *acyclic*, if it has no cycles.

Now we get to some definitions more specific to trees.

Tree A *tree* is an acyclic connected graph.

Directed In a *directed* tree, edges have a distinct direction. They start at one node and end on an other, instead of just connecting two nodes. In this thesis the term *tree* will refer to a *directed tree*.

Parent/Child A node p is the *parent* of a node c and c is the *child* of p, if $(p, c) \in E$. In A tree, nodes have only one parent, but can have multiple children.

Siblings Two nodes are *siblings*, if they have the same parent.

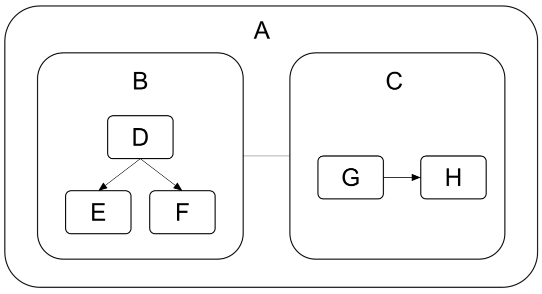
Leaf A $leaf$ is a node without children.

Root A *root* is a node without a parent. A tree has exactly one root.

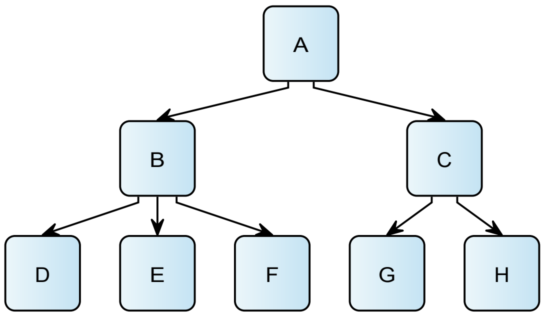
layered/non-layered A tree is *layered* if all nodes on the same level are vertically aligned. If a tree is not layered, it is a *non-layered* tree.

Subtree A *subtree* is subdevision of a tree. In a tree, every node can be viewed as the root of its own subtree. It has every property of a tree, except its root is a child in another tree. The smallest subtree possible is a single leaf.

Margin A *margin* defines an area around a node that is also occupied by this node.



(a)

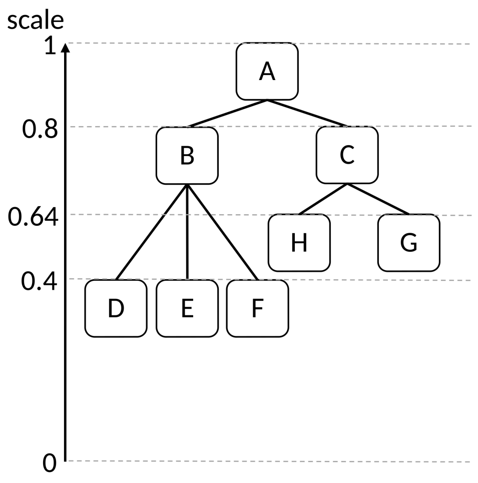


(b)

Figure 2.1. An example of a simple compound graph and its corresponding hierarchy tree, made using MrTree.

A	0.8	0.8	1
B	0.5	0.4	0.8
C	0.8	0.64	0.8
D	1	0.4	0.4
E	1	0.4	0.4
F	1	0.4	0.4
G	1	0.64	0.64
H	1	0.64	0.64

(a) Example scale factors.



(b) Visualization of render scales.

Figure 2.2. A hierachy tree with scale factors as vertical position contraints.

2.2 Visualizing Hierarchy Trees

The initial usage of this algorithm is the visualization of hierarchy trees for compound graphs of top-down layouts. A compound graph is a graph consisting of smaller graphs. They are a very useful tool for visualizing large scale software by displaying multiple software components at once, each with its own individual layout. Figure 2.3 shows multiple examples of compound graphs. The hierarchy of such a compound graph can be displayed as a tree using a simple layered layout algorithm like MrTree [Wal90], shown in Figure 2.1. In this case, the root of the tree represents the graph and each child is one of its smaller graphs. Their components are their children in the tree and so on. Top-down layout is a technique for layouting compound graphs by staring with the top level of the graph. Since the available space is now finite, components and their own layouts must be scaled to fit in the new layout. A hierarchy tree displaying each layout's scale factor can therefore not be layouted with a layered

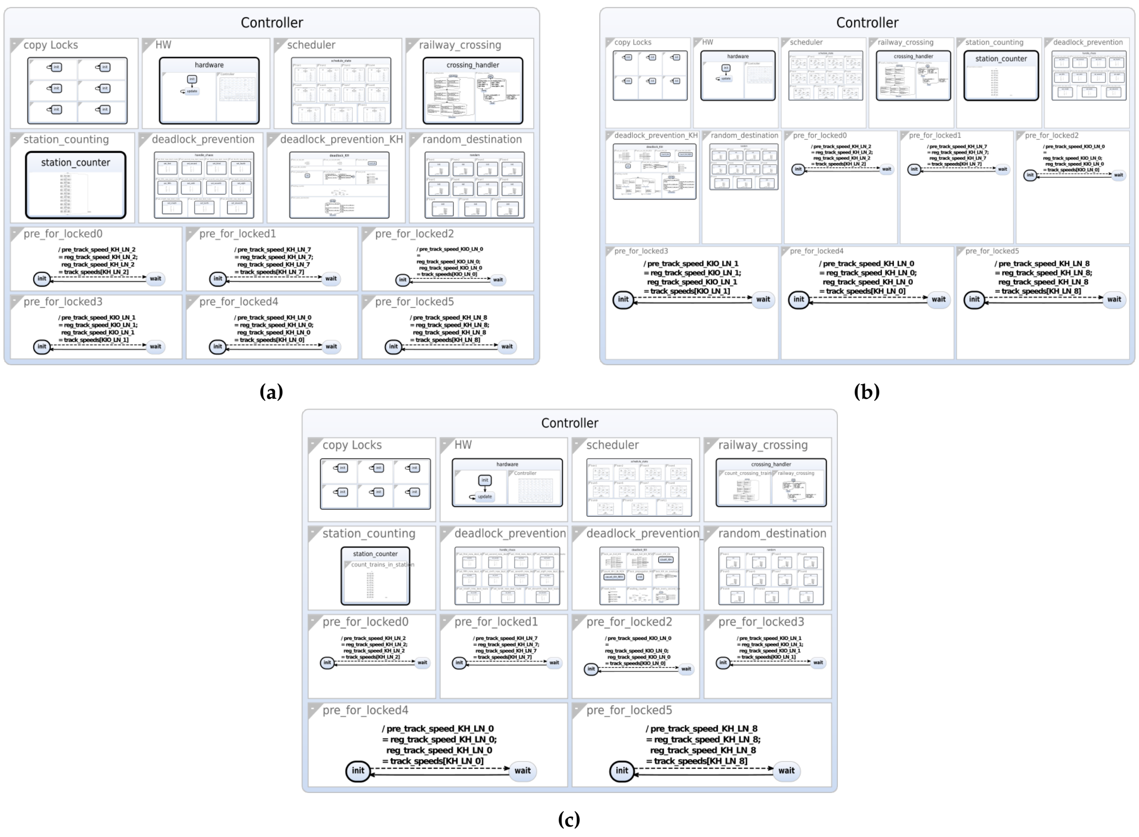


Figure 2.3. Examples of compound graphs using top-down layout,

tree. Figure 2.2 shows the scale factors of a compound graph and its associated hierachy tree with scale factor used for the vertical positions of nodes. The layout for this tree was made by hand, giving us an example of what this layout algorithm should be capable of. We can now define our aesthetic requirements accordingly.

2.3 Aesthetic Requirements

Aesthetic requirements, also called aesthetic properties, aesthetic criteria or aesthetics, are a well known concept of designing layout algorithms. They are rules for the layouts, produced by the algorithms, to make them visually pleasing. Wetherell and Shannon introduced the use of aesthetic requirements for tree layouts [WS79]. For their work, they defined three criteria. Some, like Reingold and Tilford [RT81], extend this approach with the addition of new requirements. Others created a completely new list of those rules [UW89] [HRN03]. While there are common goals for most algorithms, e.g., that nodes should not overlap, many criteria can change depending on the goal of the specific algorithm. The algorithm proposed in this thesis has the following requirements.

Self Similarity Every subtree of this tree should fulfill all requirements itself. Meaning the layout of a subtree should not depend on its position in the tree and isomorphic subtrees are drawn identically.

No Overlaps Edges and nodes should be placed in such a way that there are no overlaps. This includes overlaps of two nodes, nodes and edges or edge-crossing.

Vertical Constraints Nodes can have a fixed vertical position, that should not be changed by the algorithm.

Compactness Nodes should be placed as tightly as possible without violating other requirements to reduce whitespace.

Straightline Drawing All edges should be straight lines without bends.

Chapter 3

Concept

My goal is to create an algorithm, capable of making a layout for a given tree. Nodes of the tree can have vertical constraints, meaning they have fixed vertical positions that cannot be changed. Not all nodes have to have these constraints. In this chapter, I present the basic ideas and functionality of the proposed algorithm, splitting the task into solvable parts and discuss various approaches to solve them.

3.1 Layout Phases

To get a better structure and in preparation for an implementation in ELK the algorithm is split into phases, each with its own task. Algorithm 1 gives an overview of all the tasks. First, we have to do some preparations. We have to check whether our given tree is a valid input. Our algorithm only works on trees, so we have to check for cycles in the given graph using the *checkTree(t)* function. Second, all nodes are placed at their vertical position with the *setYCoordinates(t)* function. If a node has a vertical constraint, this is its position. Nodes without a vertical constraint are placed under their parent. We can call this function recursively with the benefit that when we place a node, its parent has already been placed.

Algorithm 1: Yconstree

```
  Input  : A tree $t$, some nodes may already have fixed vertical positions
  Output : A tree layout
  // check for illegal input
1 checkTree($t$);
  // set missing y-coodinates for every node without vertical constraints
2 setYCoodinates($t$);
  // set relative x-coodinates (recursive)
3 setRelativeX($t$);
  // calculate absolute x-coordinates
4 setAbsoluteX($t$);
  // draw the edges
5 edgeRouting($t$);
```

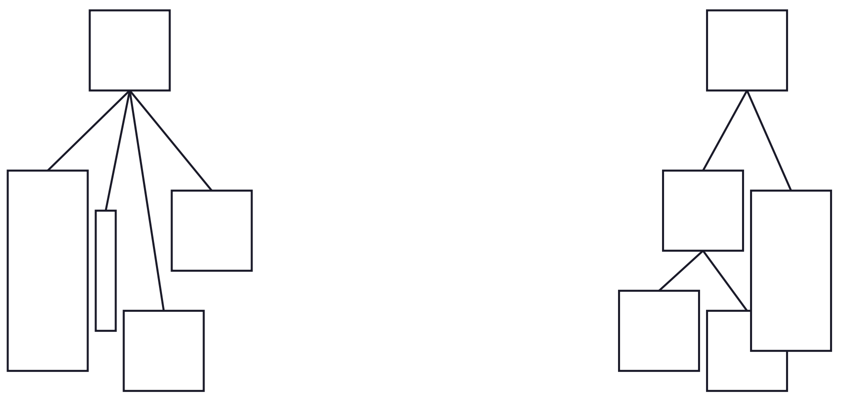

(a) A tree with different sized nodes. **(b)** Layout with an overlap.

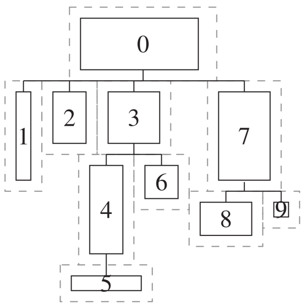
Figure 3.1. Problems of non-layered tree drawing without outlines or contours.

3.2 Outlines

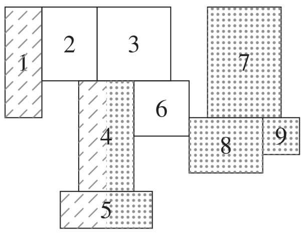
While calculating a node's x-coordinate, we have to prevent overlaps. Multiple nodes might share the same vertical positions, especially since our tree is non-layered and nodes have no restrictions regarding their height, as shown in Figure 3.1a. Furthermore, every child might have children of its own, so simply separating children with shared x-coordinates using their respective width does not solve this problem (shown in Figure 3.1b). In [Der14], v. d. Ploeg proposed a modification to the Reingold–Tilford algorithm, which uses the *contours* of subtrees to calculate distances between them and thereby upgrading it to work with non-layered trees. A *contour* is a list of rectangular shapes, describing the horizotal position of the rightmost nodes for every y-coordinate. Figure 3.2b shows a tree from v. d. Pleog [Der14] with the contour-nodes marked. This simplification enables a very fast method of distance calculation without checking every node of a subtree. His method however only works for trees with orthogonal edges since in that case, edges can be ignored most of the time (see Figure 3.2a). My Straightline-Drawing-Requirement explicitly asked for straight lines, so a new type of contour is needed. An *outline* is a linked list of relative coordinates. Each outline-node contains an x- and a y-coordinate and a link to the next outline-node. While the y-coordinate is absolute, the x-coordinate of each outline-node is relative to its predecessor, so you don't need to update them when the position of the tree changes. Each node of the tree has two outlines attached to it, the Left Outer Line (LOL) and the Right Outer Line (ROL). The first node of an outline contains the position, relative to the tree, which is dependent of the node's size and maybe some margins. So far, this could also be done with contours or even with just the size and margin of the node. The benefit of outlines is that they can follow the shape of the whole tree, including the edges, not just the nodes, shown in Figure 3.2c. This ensures the Compactness-Requirement without violating the No-Overlaps-Requirement by placing a node in the same space as an edge. Figure 3.3a shows a tree layouted by my new algorithm. Figure 3.3b shows the same tree, but this time, the outlines of its subtrees can be seen.

3.3 Relative Placement using Outlines

This phase of the algorithm calculates the horizontal position of every node relative to its parent using the *setRelativeX(t)* function. To fulfill the Self-Similarity requirement, this phase takes the form of a recursive function, which is called on the root node of the tree. This function then calls itself on every



(a) A treedrawing by v. d. Ploeg.



(b) Right and left contour of a tree, marked in a different shading.

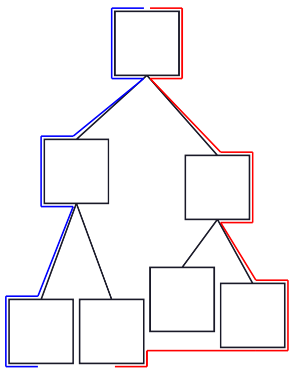
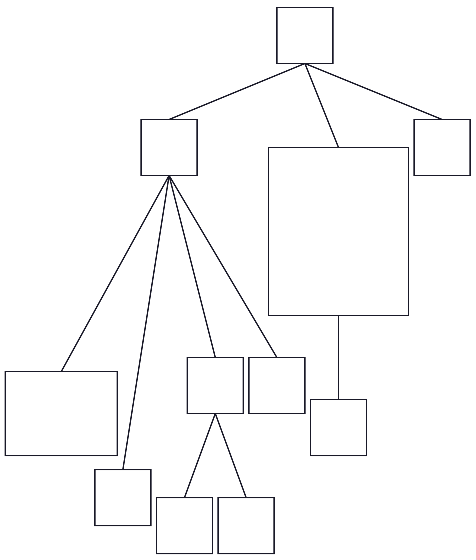
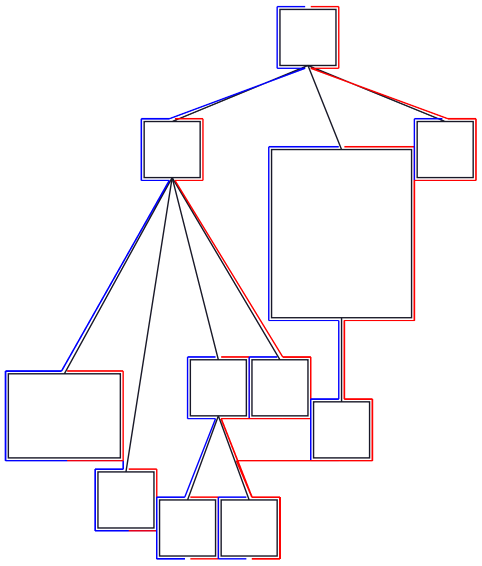

(c) A tree with outlines.

Figure 3.2. A comparison between outlines and contours.


(a) A more complex tree.


(b) The same tree with outlines and outlines of subtrees.

children of this node and layouts the entire tree in this way. The formerly discussed outlines are used in this phase to achieve compactness without overlaps. This gives our algorithm a number of tasks, it must be able to perform. I will first give a general overview of all tasks before we go into more detail. Our first task is to draw a simple outline around a single node. This can be as simple as a straight vertical line on the left and right side, consisting of two outline nodes each, or more complex. Our second task is to bundle all children of a node together without violating self similarity or create overlapping. Since every child represents its own subtree, which can have a more complicated outline, we split this task into two parts. First, we must calculate the minimum distance needed between subtrees, second if one of the subtrees has an outline reaching a deeper vertical position, the other one must enhance its outline. Our third and final task is to connect the parent and it's outlines to its children, thereby finishing the layout and forming outlines around the whole tree or subtree.

Algorithm 2: setRelativeX(t)

```
Input  : A tree or subtree $t$ all notes have fixed vertical positions
Output: The tree has now a layout.
makeSimpleOutlines($t$);
if $t.hasChildren()$ then
    $children \leftarrow t$.getChildren();
    foreach $child$ in $children$ do
        setRelativeX($child$);
    // sorting the children is only needet in one method
    children.sortToVshape();
    for $i \leftarrow 0$ to $sizeof(children) - 1$ do
        // this function always needs children[0] to update it's outline
        bundleChildren($children[0], children[i], children[i+1]$);
    $parentPosition$ = calculateParentPosition($t$) ;
    foreach $child$ in $children$ do
        $child.x \leftarrow child.x - parentPosition$;
    connectOutlines($t.leftoutline, children[0].leftoutline$);
    connectOutlines($t.rightoutline, children[sizeof(children) - 1].rightoutline$);
```

3.3.1 Setting Outlines around a Single Node

When the algorithm starts working on a node, the first step is to place outline-nodes directly around the node, using only its height, width, and margins. This is the default case and shown in Figure 3.4a. In my first and simplest approach, this is achieved by placing four outline-nodes, two for each outline, one for each corner of the rectangular node. Layouts produced using this method tend to have a lot of unnessesary whitespace. To improve this, I added some more outline nodes, to form a tighter boundary around the edges. Figure 3.4 shows a comparison between a graph with more whitespace and the one with tighter outlines. In theory, even more complicated shapes like circles or stars around the nodes are possible, but not desired in this thesis. If the given node is a leaf, the *setRelativeX(t)* function for this (sub)tree is finished at this point. If the given node is an inner node and has children, things get more complicated. We have to combine all children to a bundle and connect the outlines of this bundle to the outlines of the parent. This is described in the following sections.

3.3.2 Linking Subtrees

When an inner node only has one child, we can skip this part and go straight to connecting the outlines. But if the node has two or more children, we need to pack those as close as possible together before connecting them with their parent, enabling compactness without overlaps. We start by simply placing the leftmost child. Now, one by one, new children are added to the right of those that are already placed. This is where the outlines become useful. We can follow the last added children's ROL and the new child's LOL to calculate the distance needed for them to prevent overlaps. Once we have the distance we can place the new child in its correct place. Since the x-coordinates of trees and outlines are both relative to their predecessor, this moves not only the new child, but the whole subtree that is attached to it. However, in the end our package of children needs an outline that spans the whole package of children. If there is a difference in vertical depth between the newly added child and the children added so far, the ROL of the newly added child or the LOL of the first child must be extended

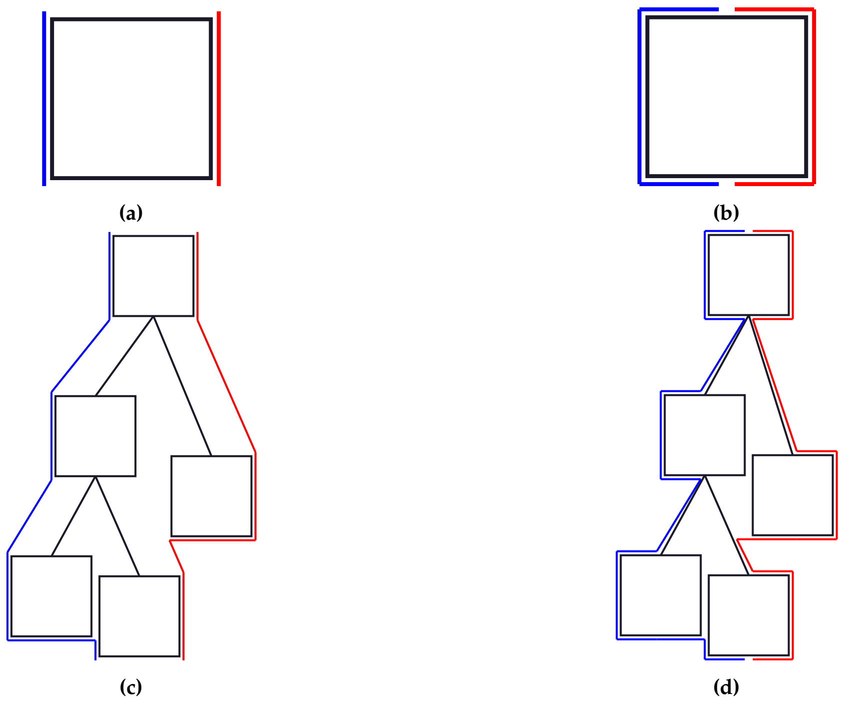


Figure 3.4. A comparison between simpler and tidier outlines.

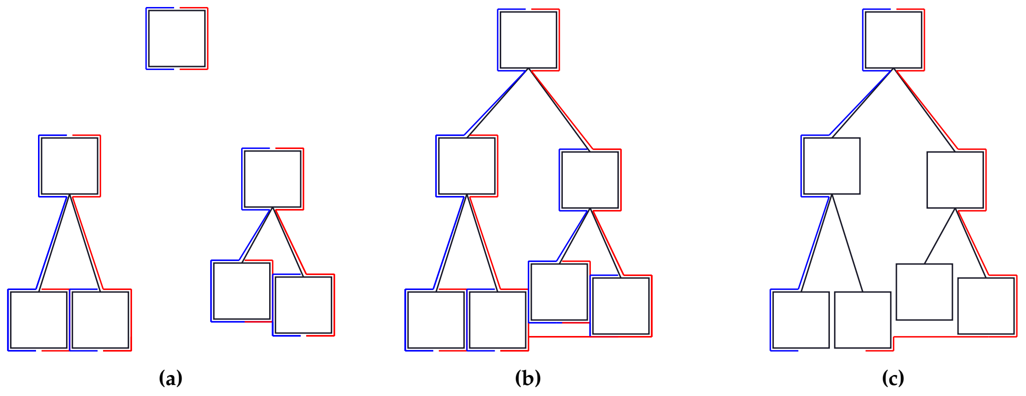


Figure 3.5. Linking the children of a node

to adapt to the new depth. Therefore, the function linking the subtrees always needs three inputs: The leftmost child, the last added child and the new child to add. In the first call of this method, the leftmost child and the last added child are the same. Figure 3.5a and Figure 3.5b demonstrate how more complex subtrees can be connected using outlines. The new outline of the entire tree is shown in Figure 3.5c, including an update to the right outline to comply with the left subtrees bigger depth.

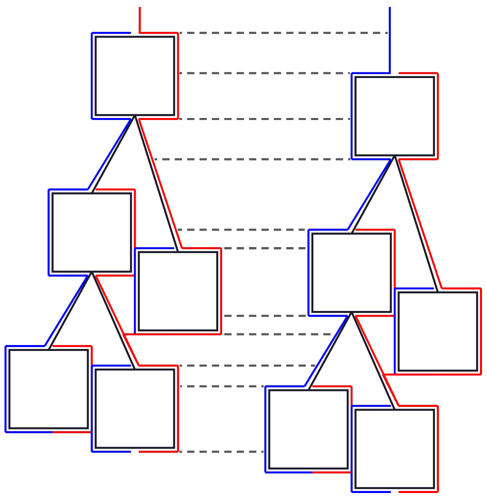


Figure 3.6. All distance calculations between two trees.

3.3.3 Calculating Minimum Distances between Subtrees

In his work, v. d. Ploeg used a very direct method to calculate minimum distances between subtrees [Der14]. For every pair of contour-parts, his approach only needed to subtract their x-coordinates from each other and got the solution. Outlines, however, have a more complex shape and while the basic idea stays the same, the comparisons become a little bit more detailed. First, we have to make one comparison per outline-node, not just one on any pair of contours. Second, instead of comparing pairs of contour-parts, for each outline node we need to calculate the other outline's x-coordinate at the y-coordinate given by the outline node. As an example, we calculate the x-coordinate of a ROL at a y-position given by an outline-node on the other subtrees LOL. In this case, relative coordinates are a disadvantage, since we cannot compare x-coordinates of two subtrees when they only contain the position relative to its own predesessor. That is why, while moving down the outlines, we accumulate x-coordinates of the outline nodes to get absolute x-coordinates. We can then start by calculating the gradient of the x-coordinate on this part of the ROL. By multiplying this with the difference in y-coordinates between the outline-node on the LOL and this part of the ROL, we get the correct x-coordinate, relative to this part of the outline. We now only need to add our accumulated x-coordinate of the ROL and compare this to the accumulated x-coordinate of the LOL node to obtain a distance. We calculate this distance for every outline-node on the LOL and similarly for every outline-node on the ROL. The maximum of all those distances is the minimal distance needed between those two subtrees. To account for cases in which one subtree is completely below the other, the start of an outline is extended with a line that goes straight up, shown in Figure 3.6.

3.3.4 Extending Outlines

When we link subtrees together, there might be a difference in vertical depth between them. After all subtrees are connected to ther parent, we want an outline surrounding the parent and all children and their subtrees, so we can use this newly created structure as a subtree, if our parent is itself a child of another node. So while linking the subtrees, we have to update the outlines of those subtrees. In every node, we save the maximal vertical position of the outlines. This makes the comparison between depths very simple. If there is a difference, one of the subtrees has to extend an outline to match

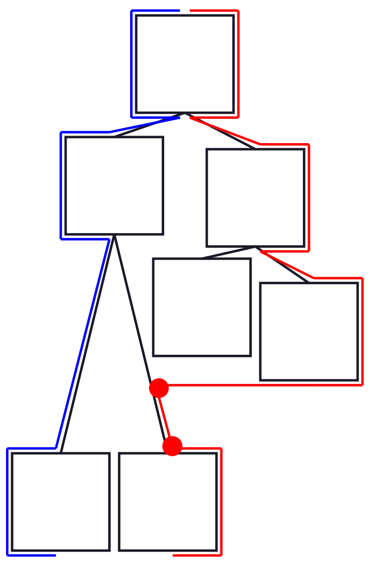


Figure 3.7. A tree with an extended outline. The newly added outline-nodes are marked.

the bigger depth. If our newly added child is the bigger one, the LOL of the leftmost child must be extended to match the LOL of our newly added subtree, and if our newly added child has the smaller subtree, its ROL must be extended to match the ROL of our last added subtree. We extend an outline by connecting us to the longer outline. To find a connecting point, we move along the longer outline until we are at the same height, where Along the way, we again accumulate our relative X-coordinates so we can calculate the distance between the end of our shorter outline and the longer one. When we have the distance, we can now extend our outline by making use of the outlines datastructure. Since an outline is a linked list, we add a new node to the end of our shorter outline to close the gap between the outlines and link this to the rest of the longer outline, using a second new node. Figure 3.7 shows which nodes must be newly added to extend the outline.

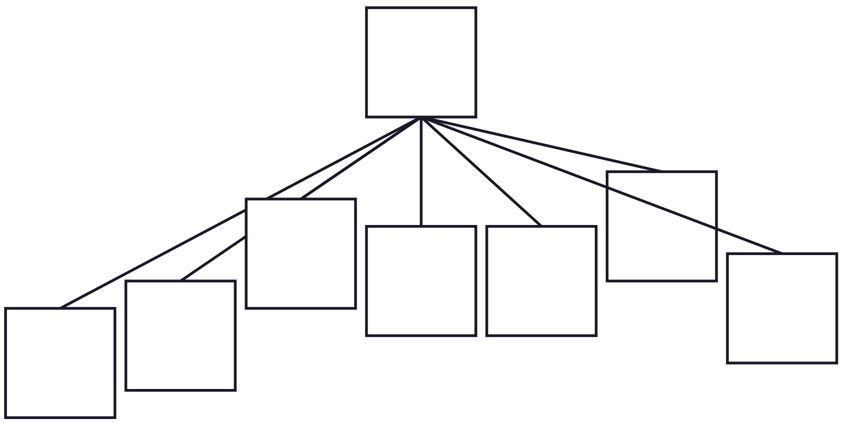


Figure 3.8. A tree with linked subtrees and straight edges. Overlaps occur.

3.3.5 The Overlapping Problem

The children of a leaf can initially be positioned at different heights. When drawing the edges to their parent node using straight lines, those lines may overlap other children. Figure 3.8 shows a tree with linked subtrees and a parent placed in the middle of them. As you can see, a straightline edgerouting would lead to overlaps which must be prevented. This problem can be solved in multiple ways, including adding more whitespace between subtrees or dropping the straight line requirement to use bends. In the following, we will discuss three of those approches.

Solution 1: Adding Whitespace

The probably simplest solution to the problem would be to add more whitespace between the subtrees, until we have no overlaps anymore. However, this method escalates to an extremly wide tree, if only one child is in a bad spot, as shown in Figure 3.9. Therefore this method will not be further covered in this thesis.

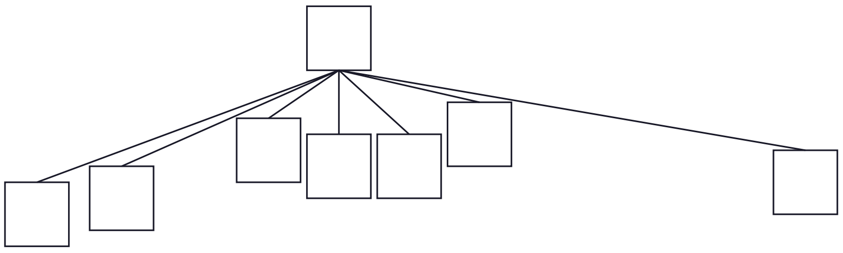


Figure 3.9. A tree layout using solution 1.

Solution 2: Ordering Children

One method proposed in this thesis solves the overlapping problem by ordering children by their starting height. This has to be done before connecting them. After ordering, the children form a V-shape to guarantee straight edges to the parent without overlaps. To visualize this, we can imagine a parent node with a single child. We place the children directly below the parent. Every new child is now added on the right side. As long as every new child is at least on the same height or higher

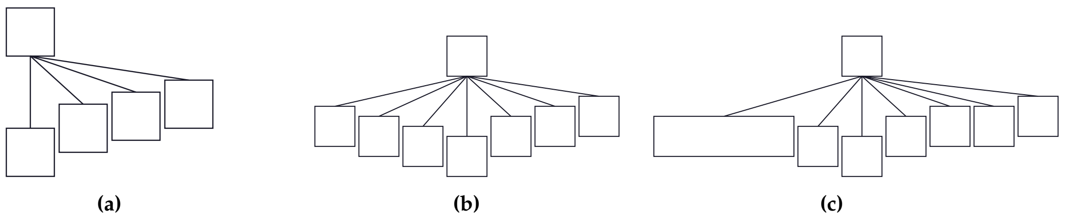


Figure 3.10. Ordering children by their height.

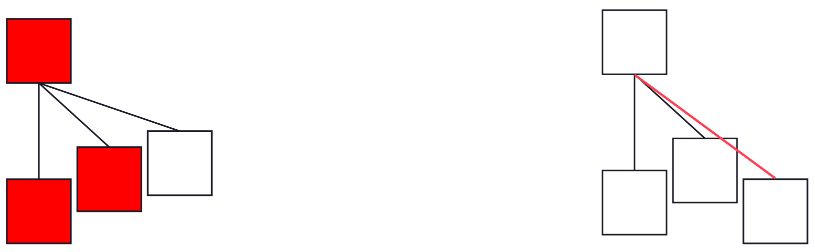
than its predecessor, there will never be an overlap (see Figure 3.10a). We now mirror this behavior on the left side and end up with the parent centered above the child with the lowest vertical position and all edges connecting with it without overlaps, shown in Figure 3.10b. So we need two lists of children ordered by their vertical height. For better aesthetics, the lists should have roughly the same accumulated width and should both include children from the whole spectrum of starting heights. This problem is quite similar to a typical scheduling problem in computer science, the equal distribution of tasks to two machines. In this case, the children's width equals the time required for a task and the two lists represent the two machines. Like most scheduling problems, this cannot be solved perfectly in linear time. We can, however achieve a good approximation of this by first ordering all children by starting heights in a singular list. Now we make two new lists and keep track of their accumulated width. By transferring every child one by one to the list with the currently lower accumulated width, we get a relatively balanced distribution, shown in Figure 3.10c. The figure shows the same tree as in Figure 3.10b but with one of the children being extremely wide. The first child gets added to one of the lists without increasing the accumulated width due to it being the middle point.

Proof of no Overlaps

To give a proof that this method guarantees no overlaps we have to consider all cases overlaps can occur.

- ▻ Overlaps between two nodes.
- ▻ Crossings between edges.
- ▻ Overlaps between edges and nodes.

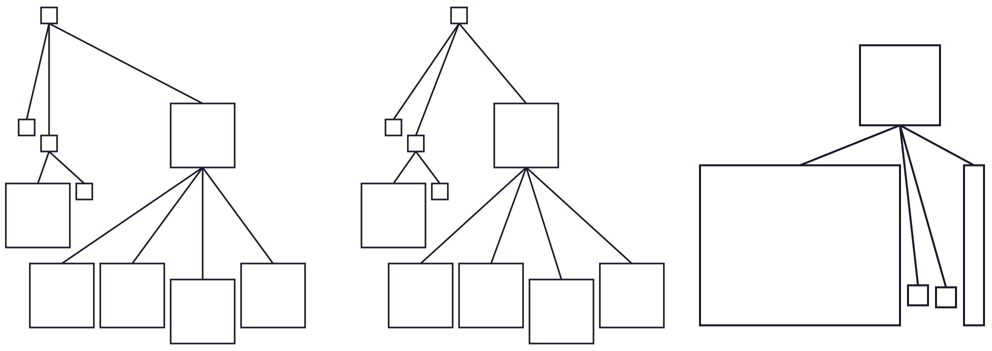
The first child can always be placed directly underneath the parent. For every additional child, we can then check for overlaps. In this proof, we only place children to the right of our already added children, but by mirroring the behavior to the left side, we obtain a proof for the whole V-shape. The first possible case of overlaps are overlaps between two nodes. We mark every place our newly added node cannot be in Figure 3.11a. Since it is one of our prerequisites that children have a lower vertical position than their parents, an overlap with the parent is impossible. Using our distance calculation from Section 3.3.3, we can place our new child next to the already placed ones, preventing overlaps between them. This also prevents overlaps with edges or nodes inside the children's subtrees, since those are all inside the subtree's outline and thereby part of the node's outer shape. All edges between parent and children start in a joint point on the parents bottom and are straight lines, so crossings between those lines are impossible as well. Our last possible case for overlaps are overlaps between edges and nodes. All new nodes are added to the right, their edges go to the left or in case of the first child, straight up. Therefore our outline extension in Section 3.3.3 can also prevent overlaps between our newly added node and other nodes edges. For overlaps between our new edge and the other


(a) Illegal places for a new node. **(b)** Illegal placed edge.

nodes the other node must be between the ending point of the edge and the parent. To be in this position, it must have a higher vertical position than the newly added node, shown in Figure 3.11b, which is not possible for our list of children, sorted in ascending order. Therefore, an overlap cannot occur in this scenario.

Placing the Parent

We now have to find a position for the parent. The simplest solution would be to find the child with the lowest vertical position and place the parent directly above it. In some cases, this position is not very fitting, as shown in Figure 3.12a. The parent is very far to the left since there is a big gap between the second and the third child. We can have a small fix for multiple children with the same height, by placing the parent in the middle of those, but this does not solve the overall problem. If we could place the parent in the middle of all its children, it would give the tree a better overall look as shown in Figure 3.12b. However, while we have a proof of no overlaps for our first position, this is not guaranteed for our better position. A solution of this problem is to calculate both positions, put the parent on the position where no overlaps are guaranteed and move it as far as possible to the better position. The parents better position in Figure 3.12c would be a litte bit more to the left, but it didn't move that far, because that would create an overlap.


(a) A tree with the parent directly above the lowest child. **(b)** The same tree with an improved layout. **(c)** The parents position is restrained to prevent an overlap.

Solution 3: Redefining the Straightline Requirement

The previous solution, while fitting our aesthetic criteria, had one major flaw: It destroyed the model order, the order in which the children of a node are listed. This was not an aesthetic criteria, but is a desired goal and the ignoring it can make working with this algorithm more frustrating, especially if

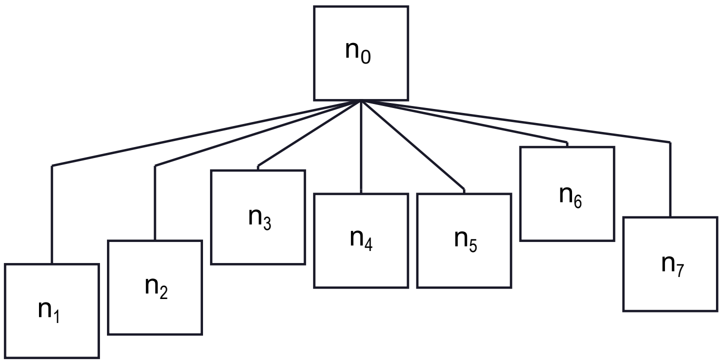


Figure 3.13. A tree layout using bendpoints, thereby preserving the model order.

you want a specific order. Therefore, I propose one more method to solve this problem. We add a new order-preserving requirement and redefine our straightline requirement, to make an exception, if an insistence on straight lines would lead to overlap. These are our modified requirements.

Order Preserving Children of a node should be drawn in the same order, in which they are given to the algorithm.

Straightline Drawing Wherever this would not lead to overlaps, edges should be straight lines without bends.

With our new criteria, we can now place our parent in the middle between its children and set bendpoints to positions above them, where no overlaps are guaranteed again. Figure 3.13 shows a tree layouted using this new method. Notice that the algorithm sets bendpoints whenever there are nodes with a higher vertical position between them and the parent node. Since we don't require reordering children, this method can be executed after linking the subtrees.

3.3.6 Connecting Children to their Parent

In the *setRelativeX(t)* phase, all children are placed relative to their parent. Therefore when we talked about placing the parent in the last sections, we actually only calculated this position and moved the children, so our parent stays at $x = 0$. Now we need to connect the outlines of the parent to the children. We connect the LOL of the parent to the LOL of the leftmost child and the ROL of the parent to the parent to the ROL of the rightmost child. As in Section 3.3.4, we can again take advantage of the linked list structure of outlines. Per outline, we create one new node to bridge the gap between the outlines. If we use our *Redefining the Straightline Criteria* solution from **??** to our overlapping problem, we may need to add an additional outline node to take possible bendpoints of the edges into account.

3.4 Finishing the Layout

When all nodes are succesfully placed relatively, we can now finish the layout. First, we move the root of our tree. Since the root is currently at $x = 0$, some of its children may have negative coordinates. We find the minimal x-coordinate of the whole tree and move our root by the same amount, thereby moving the whole tree into positive coordinates. Then, we convert all relative x-coordinates into absolute ones. We can do that again using recursion and calculate an absolute x-coordinate on the way. In our last phase, we now have to do some basic edgerouting. For most cases, edges are straight lines between the bottom of parents and the top of their children, for our Section 3.3.5 method, we may need a bendpoint.

Chapter 4

Implementation

4.1 Developing in Python

In ELK, algorithms are implemented in a more complex, class driven structure, consisting of phases and processors (see Section 4.2). For developing and testing the algorithm on its own, a simpler and more direct form seemed advantageous. I decided for a first implementation in Python. This way, I can also include the algorithm as a simple text document in the appendix of this thesis. To be able to test it, a visual plotter is needed. I made a simple tree plotter using the matplotlib.pyplot library[1]. Matplotlib is usually used for plotting functions and diagrams, but you can also use it to simply display a picture, consisting of an array of red-green-blue (RGB) values. So before we can actually display our trees, we first need some functions, to draw the layouted tree onto our canvas. Here, another advantage of the custom-made plotter comes into place. Since outlines are a new structure introduced by this algorithm, KIELER is not able to render those outlines. Most debugging was related to wrongly placed outlines, so a method of displaying those outlines was essential. Using my custom-made tree plotter, a visualization during development is possible. The rendering of outlines in my own tool was easy to implement, since it uses the same functions and librarys as needed for the edge routing. Implementing such a feature into KIELER is possible, but would require a deeper look into the rendering software of KIELER, which is so far not part of this thesis.

4.1.1 Rendering Trees using Pyplot

Pyplot can render pictures given to it as a three dimensional *Numpy array*[2]. Numpy is a library for fixed sized arrays and more limited data types, that are not standard in Python. Therefore, it is required for many libraries, including pyplot. The first and second dimension of our array represent width and height of the picture, the third dimension has a fixed size of three for red, green and blue values of the discribed pixel. All values are stored as eight bit unsigned integers. We can now write directly onto this array and Pyplot allows us to see the result without needing to export them as a picture. Tree nodes are drawn as rectangles using their width and height, while edges and outlines are drawn using the *Bresemheim algorithm*, a typical algorithm for rendering lines[Bre98]. In Python, this algorithm can be imported as a library. In my depiction, left outlines are drawn blue, while right outlines are drawn red. The rendering to pixels allows me to display parts, where left and right outlines connect as violett. It does this by checking whether a pixel is already colored and, if so, only changing its red or blue color channel.

[1]https://matplotlib.org/
[2]https://numpy.org/

4.1.2 Rendering Trees to SVGs

The pyplot renderer is very useful for developing and debugging, however the pixelated display is not beneficial for presenting results. Scaling up the resolution can help, but it leads to thinner and thinner lines. A vector image can achieve better results than this by not describing pictures as an array of pixels, but instead as a collection of geometrical forms. Therefore, it has no fixed resolution. In Python, vector-images can be created using the *svgwrite library*[3]. The images are exported as *SVG-files*. Most of the pictures shown in this thesis are made using this SVG-exporter.

4.2 Eclipse Layout Kernel

ELK is a framework for layout algorithms. It holds a collection of algorithms and an infrastructure that bridges the gap between layout algorithms and diagram viewers and editors [4]. Actively developed by the Real-Time and Embedded Systems group at Kiel University, it is hosted under the Eclipse Foundation. ELK is not only used for trees, but for all kinds of graph structures. Graphs in ELK consist of emphElkNodes as its nodes/vertices and emphElkEdges as its edges. Some of ELK's special features are ports, which are not important for this implementation, and hierarchical nodes, which can be used with this algorithm. Hierarchical nodes are nodes or even whole graphs inside a node [Fou23]. This affects width and height of that corresponding node. Besides basic attributes like the nodes x- and y-coordinate or its width and height, nodes in ELK can have a number of properties attached to them, which can be given default values or set manually. The set of available options includes for example emphmargins, small borders around a node where other nodes shouldn't be placed. Algorithms can support those properties by including them into their calculations. Other features, like hierarchical nodes, are indirectly supported by including the node's width and height and calculating the size of the finished graph. Algorithms can also introduce their own properties. In our case, I introduced *vertical Constraint*, a new property to be set by the user. It defines the vertical position, the node must have in the final layout.

4.2.1 KIELER

KIELER is a research project by the Real-Time and Embedded Systems group at Kiel University. The name was an acronym for Kiel Integrated Environment for Layout Eclipse Rich Client, aimed at the Eclipse platform. Nowadays, the project also includes a wide variety of Java and JavaScript libraries, standalone tools and VS Code extensions. ELK and its associated tools, The KLighD framework[SSH13], the modeling language SCCharts and tools for process safety analysis and software project visualizations are all part of the KIELER project. For my work, I used the original KIELER tool, based on Eclipse. Figure 4.2 shows a tree layouted by my algorithm *Yconstree* in the KIELER tool. On the left of the picture, all nodes, edges and other options are defined. The program can then use ELK and my algorithm to create a layout of the given tree and draw it on the right side of the screen.

4.2.2 Implementation in ELK

Algorithms in ELK are built using a pipeline of phases and processors. Phases are the main component of an algorithm and describe the main steps of calculating the layout. Between those phases, processors can be added to the pipeline to calculate smaller, more modular calculations. Figure 4.1 shows an

[3]https://pypi.org/project/svgwrite/
[4]https://eclipse.dev/elk/

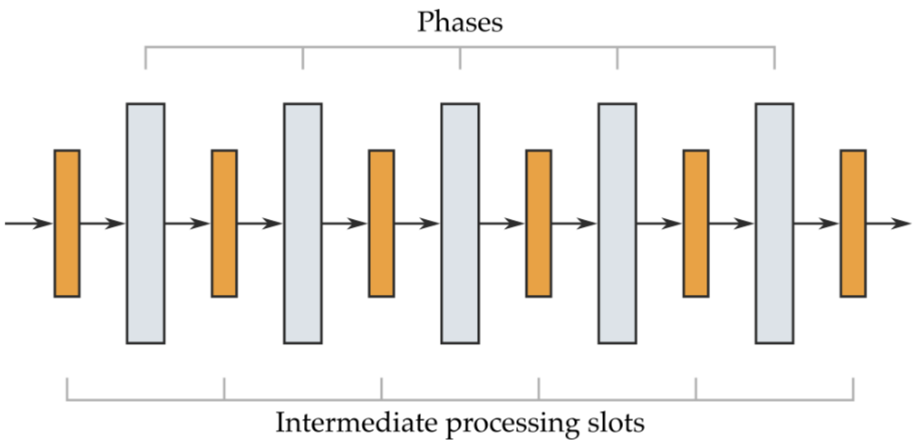
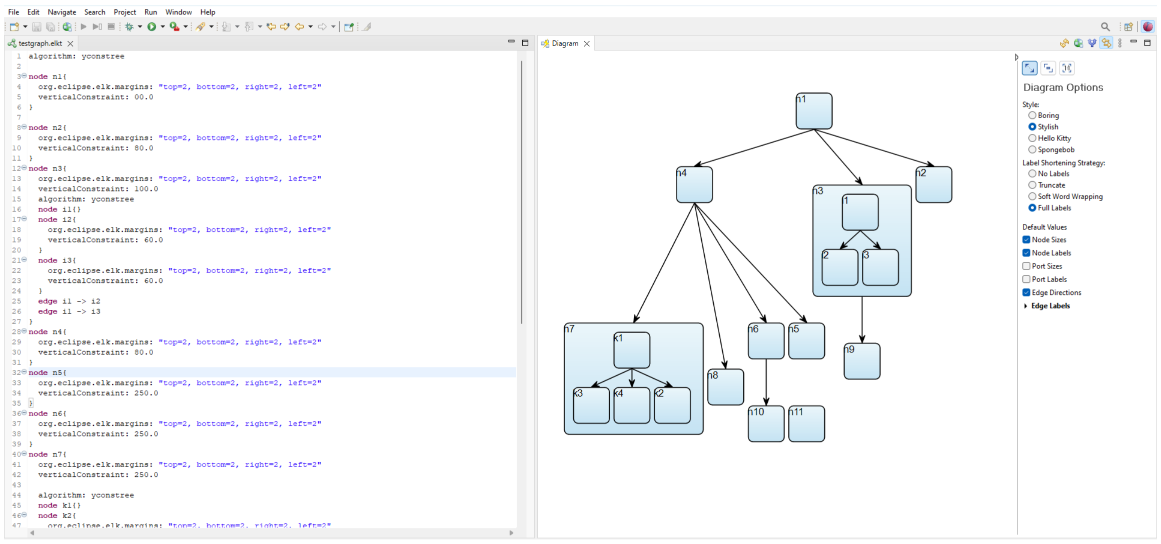


Figure 4.1. A visualization of the algorithms layout.

overview of this pipeline. For my approach there was no need for the use of processors, so I structured it into four phases. I originally planned to have the check for an illegal graph in a completely separate phase, but this turned out to be better when done inside the algorithm's main phases. I ended up with a phase for the placement of all y-coordinates, one for the relative placement of x-coordinates, one phase that transforms those coordinates into absolute coordinates and a last one for edge routing.

Graphs in ELK work a little different, compared to the trees I used in my algorithm development. One example of this is that while my Python version simply takes the root as its input, layout algorithms in ELK get a *graph*-object as an input, which is an ElkNode containing all nodes and edges of the graph as its hierarchical children. In ELK, the term children does not refer to the parent-child relationship of nodes in trees, but to its hierarchical children inside the node. Figure 4.3 shows a node with its five children. ELK is not specifically designed for trees, but for all kinds of graphs, which might not have a root. Therefore, graphs don't have a specially marked root node. All nodes of a graph are simply the children of this graph. Finding the children of a node in the context of trees can be done by finding the destination of all *outgoing edges* of that node. A benefit of the more general design of ELK is that we can also find a nodes parent by moving to the startpoint of an incomming edge. This way, we can find the root of our tree and check for illegal inputs by looking at the number of incomming edges in all nodes of the graph. One node, the root, should not have an incomming edge. If a node has more than one incomming edge, it is not a tree, therefore an illegal input.



```
algorithm: yconstree

node n1{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 00.0
}

node n2{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 80.0
}
node n3{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 100.0
  algorithm: yconstree
  node i1{}
  node i2{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  node i3{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  edge i1 -> i2
  edge i1 -> i3
}
node n4{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 80.0
}
node n5{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 250.0
}
node n6{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 250.0
}
node n7{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 250.0

  algorithm: yconstree
  node k1{}
  node k2{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
```

Diagram Options

Style:
Boring
[✓ Stylish]
Hello Kitty
Spongebob

Label Shortening Strategy:
No Labels
Truncate
Soft Word Wrapping
[✓ Full Labels]

Default Values
[✓ Node Sizes]
[✓ Node Labels]
Port Sizes
Port Labels
[✓ Edge Directions]
Edge Labels

Figure 4.2. A Tree in the KIELER tool.

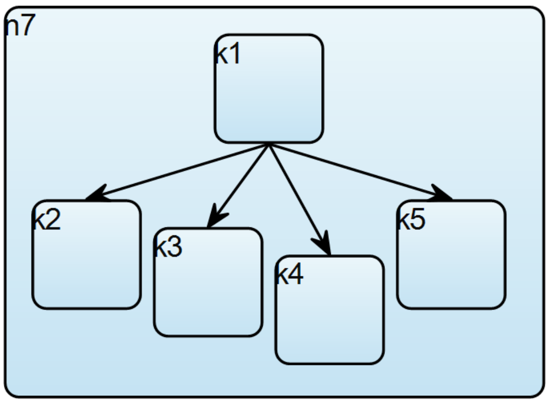


Figure 4.3. In ELK, *children* are the nodes inside a node.

Chapter 5

Conclusion

This chapter first summarises the results of this thesis in Section 5.1. Then, we discuss possible future work in Section 5.2, which could enhance our presented solution.

5.1 Summary

This thesis proposed a new algorithm for non-layered tree layouts. Designed for creating hierarchy trees in top-down layouts, nodes of this tree can have vertical position constraints and edges are mostly drawn in straight lines. We defined a number of aesthetic requirements for such a layout and identified key challenges for such a design. To enable straightline drawing, outlines are introduced as a more detailed variant of contours used in a work from v. d. Ploeg, which inspired this approach. A problem regarding overlaps while simultaneously reducing whitespace and keeping straight lines and model order proofed to be a greater challenge. We gave two solutions to this problem, one by abandoning model order and ordering nodes in a helpful way and one by making a modification to our *straightline requirement* allowing a bendpoint and thereby avoiding all possible overlaps. The finished algorithm was then implemented into ELK for later use in the KIELER project. The algorithm is called *Yconstree*. Most structures and functions could be directly transferred from python into ELK. Some changes had to be made, to account for the classdriven structure of ELK. The implementation is able to produce layouts as required by the task and can now be used for further research.

5.2 Future Work

This section presents some possible improvements and future applications on this topic.

5.2.1 Further Development in ELK

ELK has a lot of options and features for layout algorithms, which can be supported by the algorithms. Currently, Yconstree only supports a handfull of these options. Not every option and feature is useful in every algorithm, but more possibilities for customisation of the produced layouts can improve the usefulness of the resulting layouts. We can also think about the introduction of further new options, special to this algorithm. A good example of this is the inclusion of multiple shapes for outlines around a node. This would enable more detailed outlines and could be done by using preprocessors. We can also use this option to restore the first, extremely simplified version of outlines around the node from Section 3.3.1 as an available option.

5.2.2 A Pipeline of Automatic Hierarchy Trees

The current work on hierarchy trees for top-down graph layouts can generate automatic discriptions of those trees. These discriptions however cannot be directly imported into elk, since they use a different discription format. A simple conversion tool might be helpfull to close this gab an enabeling a completly automatic pipeline for generating and visualizing hierarchy trees.

5.2.3 Runtime Optimization

The algorithm [Der14] of v. d. Ploeg is able to produce a layout in linear time, wich is the optimal solution for an algorithm to place every node. While the version with a strict staight line requirement is far from linear time and will probably never reach that goal, due to its reliance on sorting algorithms, the method with a modified straightline requirement is quite close to run in linear time. With further optimization, these layouts might be possible in linear time.

Appendix A

Python Code

This is the Algorithm, made in Python 3.11. You find examples of trees at the bottom part. The code is split into fife parts, separated by horizontal lines. The first part introduces the two new classes, second and third part contain functions for drawing the tree using Pyplot or as an SVG. The fourth part is the algorithm and the last part are examples.

```
import numpy as np
import matplotlib.pyplot as plt
from bresenham import bresenham
import svgwrite

class Tree:
    '''
    y is an absolute coord, during the alg, x is realtive to it’s parent
    '''
    def __init__(self, w, h, y, c):
        self.w = w
        self.h = h
        self.x = 0
        self.y = y
        self.c = c
        self.margin_t = 2
        self.margin_b = 2
        self.margin_l = 2
        self.margin_r = 2
        if c == None:
            self.cs = 0
        else:
            self.cs = len(c)
        # die outlines
        self.lol = None
        self.rol = None
        # minimal und maximal values of the area of the tree
        self.minX = 0
        self.maxX = 0
        self.minY = 0
        self.maxY = 0
        # maximale depth of an outline
        self.outline_maxY = 0
        self.edgeCornerHeight = 0
```

```
class Outline:
    '''
    A point in the outline. All x-coordinates are relative to their predesessor
    '''
    def __init__(self, x, y, next):
        self.x = x
        self.y = y
        self.next = next


# --------------------------------- Drawing ------------------------

def drawline(pic, x1, y1, x2, y2, color):
    points = list(bresenham(x1, y1, x2, y2))
    for i in points:
        (x, y) = i
        if y >= 0:
            if pic[y,x][1] == 0:
                pic[y,x] = pic[y,x] + color
            else:
                pic[y,x] = color

def drawoutline(pic, outline, color):
    # x must be added, y can be kept, cause it is absolute
    x = outline.x
    o = outline

    while o.next != None:
        drawline(pic, x, o.y, x + o.next.x, o.next.y, color)
        x += o.next.x
        o = o.next

# show function for outlines
def showOutlines(outlines):
    hight = 300
    width = 400
    picture = np.zeros( (hight,width,3), dtype=np.uint8 )
    for outline in outlines:
        drawoutline(picture, outline, [200,200,200])
    plt.imshow(picture)
    plt.show()



def drawtree(pic, t, color):
    def drawpoint(pic, t, color):
        for a in range(t.w + 1):
            pic[t.y, t.x + a] = color
            pic[t.y + t.h, t.x + a] = color
        for b in range(t.h):
            pic[t.y + b, t.x] = color
            pic[t.y + b, t.x + t.w] = color
    drawpoint(pic, t, color)
    if (t.cs > 0):
        for i in t.c:
            drawtree(pic, i, color)
            drawline(pic, (t.x + t.w//2), t.y + t.h, i.x + i.w//2, i.edgeCornerHeight, color)
            drawline(pic, i.x + i.w//2, i.edgeCornerHeight, i.x + i.w//2, i.y, color)

def showTree(t, outlines):
    def drawoutlines(t, allOutlines):
        t.lol.x += t.x
        t.rol.x += t.x
        drawoutline(picture, t.lol, [0,0,255])
        drawoutline(picture, t.rol, [255,0,0])
        if allOutlines and t.cs > 0:
            for a in t.c:
                drawoutlines(a, allOutlines)



    width = t.maxX - t.minX + 1
    hight = t.maxY + 1

    picture = np.zeros( (hight,width,3), dtype=np.uint8 )
    for y in range(hight):
        for x in range(width):
            picture[y,x] = [255,255,255]


    drawtree(picture, t, [100,100,100])
    if outlines == 'y' or outlines == 'a':
        drawoutlines(t, outlines == 'a')

    plt.imshow(picture)
    plt.show()


# ----------------------------------  svg       -----------------------
def svg_drawoutline(dwg, outline, offset, color):
    x = outline.x
    o = outline
    while o.next != None:
        startpoint = (x + offset, o.y + offset)
        endpoint = (x+o.next.x+offset, o.next.y+offset)
        dwg.add(dwg.line(startpoint, endpoint, stroke=color))
        x += o.next.x
        o = o.next

def svg_drawtree(dwg, t, offset, color):
    def svg_drawpoint(dwg, t, color):
        dwg.add(dwg.rect((offset + t.x, offset + t.y), (t.w, t.h),
            stroke=color,
            fill='white')
        )

    svg_drawpoint(dwg, t, color)

    if (t.cs > 0):
        for i in t.c:
            svg_drawtree(dwg, i, offset, color)
            startpoint = (offset + t.x + t.w//2, offset + t.y + t.h)
            endpoint = (offset + i.x + i.w//2, offset + i.edgeCornerHeight)
            dwg.add(dwg.line(startpoint, endpoint, stroke=color))

            startpoint = (offset + i.x + i.w//2, offset + i.edgeCornerHeight)
            endpoint = (offset + i.x + i.w//2, offset + i.y)
            dwg.add(dwg.line(startpoint, endpoint, stroke=color))


def makeSVG(t, savedir, name, outlines):
    offset = 10
    def svg_drawoutlines(t, allOutlines):
        t.lol.x += t.x
        t.rol.x += t.x
        svg_drawoutline(dwg, t.lol, offset, 'blue')
        svg_drawoutline(dwg, t.rol, offset, 'red')
        if allOutlines and t.cs > 0:
            for a in t.c:
                svg_drawoutlines(a, allOutlines)

    if name == "":
        name = input("Name of the file? ")

    dwg = svgwrite.Drawing(savedir + name + '.svg', profile='tiny')


    svg_drawtree(dwg, t, offset, svgwrite.rgb(10, 10, 16, '%'))
    if outlines == 'y' or outlines == 'a':
        svg_drawoutlines(t, outlines == ’a’)
        print("drawing completed")
    dwg.save()



# ---------------------------------  algs      ---------------------------

# calculate minimal needed distance between two outlines to prevent overlaps
def outlineDistance(outline1, outline2):
    # a constant for a very small value, outlines can start at this value
    minimalY = -100

    # build new outlines with the minimalY value, this ensures a result.
    changed_outline1 = Outline(outline1.x, minimalY, Outline(0, outline1.y, outline1.next))
    changed_outline2 = Outline(outline2.x, minimalY, Outline(0, outline2.y, outline2.next))

    # the return value
    dist = changed_outline1.x - changed_outline2.x

    # first run (compare points of o1 with o2)
    o1 = changed_outline1
    o2 = changed_outline2
    x1 = o1.x
    x2 = o2.x
    while o1 != None and o2.next != None:
        if o2.next.y > o1.y:
            deltaX = o2.next.x
            deltaY = o2.next.y - o2.y
            newdist = x1 - x2 - ((o1.y - o2.y) * deltaX) // deltaY

            dist = max([dist, newdist])

            o1 = o1.next
            if o1 != None:
                x1 += o1.x
        else:
            o2 = o2.next
            x2 += o2.x

    # second run (compare points of o2 with o1)
    o1 = changed_outline1
    o2 = changed_outline2
    x1 = o1.x
    x2 = o2.x
    while o2 != None and o1.next != None:
        if o1.next.y > o2.y:
            deltaX = o1.next.x
            deltaY = o1.next.y - o1.y
            newdist = x1 - x2 + ((o2.y - o1.y) * deltaX) // deltaY

            dist = max([dist, newdist])

            o2 = o2.next
            if o2 != None:
                x2 += o2.x
        else:
            o1 = o1.next
            x1 += o1.x


    return dist

def makeSimpelOutlines(t):
    endpart = Outline(0, t.y + t.h + t.margin_b,Outline(t.w//2,t.y+t.h+t.margin_b,None))
    t.lol = Outline(-t.margin_l+t.w//2,t.y-t.margin_t,Outline(-t.w//2,t.y-t.margin_t,endpart))
    endpart = Outline(0,t.y+t.h+t.margin_b,Outline(-(t.w//2),t.y+t.h+t.margin_b,None))
    t.rol = Outline(t.w//2+t.margin_r,t.y-t.margin_t,Outline(t.w//2,t.y-t.margin_t,endpart))
    t.minX = t.x - t.margin_l
    t.maxX = t.x + t.margin_r + t.w
    t.minY = t.y - t.margin_t
    t.maxY = t.y + t.margin_b + t.h
    t.outline_maxY = t.lol.next.next.y

def bundleChildren(leftSubtree : Tree, a : Tree, b : Tree):
    # a constant for a very small value, outlines can start at this value
    minimalY = -100

    dist = outlineDistance(a.rol, b.lol)
    b.x = a.x + dist

    # extend left outline
    if leftSubtree.outline_maxY < b.outline_maxY:
        lastL = leftSubtree.lol
        LabsX = lastL.x + leftSubtree.x # absolute x coordinate
        # jump to the end of leftSubtree
        while lastL.next != None:
            lastL = lastL.next
            LabsX += lastL.x
        # now find a fitting spot in lol of b
        bItterator = Outline(b.lol.x, minimalY, Outline(0, b.lol.y, b.lol.next))
        BabsX = bItterator.x + b.x
        while bItterator.next.y <= lastL.y:
            bItterator = bItterator.next
            BabsX += bItterator.x
        # We are now at the right spot, we can now make the link.
        deltaX = bItterator.next.x
        deltaY = bItterator.next.y - bItterator.y
        change = ((lastL.y - bItterator.y) * deltaX) // deltaY
        # The new point.
        newX =  -LabsX + BabsX + change
        newNext = Outline(bItterator.next.x - change, bItterator.next.y, bItterator.next.next)
        lastL.next = Outline(newX, lastL.y, newNext)
        # update outline_maxY
        leftSubtree.outline_maxY = b.outline_maxY

    # extend right outline
    if b.outline_maxY < a.outline_maxY:
        lastB = b.rol
        BabsX = lastB.x + b.x
        # jump to the end of b
        while lastB.next != None:
            lastB = lastB.next
            BabsX += lastB.x
        # now find a fitting spot in rol of a
        aItterator = Outline(a.rol.x, minimalY, Outline(0, a.rol.y, a.rol.next))
        AabsX = aItterator.x + a.x
        while aItterator.next.y <= lastB.y:
            aItterator = aItterator.next
            AabsX += aItterator.x
        # We are now at the right spot, we can now make the link.
        deltaX = aItterator.next.x
        deltaY = aItterator.next.y - aItterator.y
        change = ((lastB.y - aItterator.y) * deltaX) // deltaY
        # The new point.
        newX = AabsX - BabsX + change
        newNext = Outline(aItterator.next.x - change, aItterator.next.y, aItterator.next.next)
        lastB.next = Outline(newX, lastB.y, newNext)
        # update outline_maxY
        b.outline_maxY = a.outline_maxY

def YconstreeStep(t : Tree):
    makeSimpelOutlines(t)
    if t.cs > 0:
        # layout for all children
        for i in t.c:
            YconstreeStep(i)
        # sort children
        sortSubTrees(t)
        # linking children
        for i in range(t.cs - 1):
```

```
        bundleChildren(t.c[0], t.c[i], t.c[i+1])

    # find a good spot for the parent
    pos = 0
    maxDepth = 0
    maxDepthStartPos = 0
    while pos < t.cs and t.c[pos].y >= maxDepth:
        if t.c[pos].y > maxDepth:
            maxDepthStartPos = pos
            maxDepth = t.c[pos].y
        pos += 1
    shift = (t.c[maxDepthStartPos].x + t.c[pos-1].x + t.c[pos-1].w - t.w) // 2 - t.x

    better_shift = (-t.c[0].x+t.c[0].w//2+t.c[t.cs-1].x+t.c[t.cs-1].w//2-t.w) // 2 - t.x

    # if better_shift is left from shift
    if better_shift < shift:
        for i in range(0,maxDepthStartPos):
            for j in range(i+1, maxDepthStartPos + 1):
                Rol = t.c[i].rol
                RolX = t.c[i].x + Rol.x
                posX = t.c[j].x + t.c[j].w // 2
                while Rol != None and Rol.y < maxDepth:
                    deltaY = ((t.y + t.h) - maxDepth)
                    new_shift = posX-t.w//2+(posX-RolX)*deltaY // (maxDepth-Rol.y)
                    better_shift = max([better_shift, new_shift])

                    # update Rol und RolX
                    Rol = Rol.next
                    if Rol != None:
                        RolX += Rol.x
        shift = better_shift

    # if better_shift is right from shift
    if better_shift > shift:
        for i in range(pos,t.cs):
            for j in range(pos-1, i):
                Lol = t.c[i].lol
                LolX = t.c[i].x + Lol.x
                posX = t.c[j].x + t.c[j].w // 2
                while Lol != None and Lol.y < maxDepth:
                    deltaY = ((t.y+t.h)-maxDepth)
                    new_shift = posX-t.w//2+(posX-LolX)*deltaY // (maxDepth-Lol.y)
                    better_shift = min([better_shift, new_shift])

                    # update lol und LolX
                    Lol = Lol.next
                if Lol != None:
                    LolX += Lol.x
            shift = better_shift

        # move the children, so the parent is in the right spot
        for i in t.c:
            i.x -= shift



        # lol update:
        newX = t.c[0].x + t.c[0].lol.x - t.lol.x
        t.lol.next.next.next.next = Outline(newX, t.c[0].lol.y, t.c[0].lol.next)
        # rol update:
        newX = t.c[t.cs-1].x + t.c[t.cs-1].rol.x - t.rol.x
        t.rol.next.next.next.next = Outline(newX, t.c[t.cs-1].rol.y, t.c[t.cs-1].rol.next)

        # update outline_maxY
        # update min und max for x und y
        for i in t.c:
            t.outline_maxY = max([t.outline_maxY, i.outline_maxY])
            t.minX = min([t.minX, i.x + i.minX])
            t.maxX = max([t.maxX, i.x + i.maxX])
        t.maxY = t.outline_maxY

        # set edgecornerheight. only needet for visuals an a dummy in this configuration.
        for child in t.c:
            child.edgeCornerHeight = child.y

def alternativeYconstreeStep(t):
    makeSimpelOutlines(t)
    if t.cs > 0:
        # layout for all children
        for i in t.c:
            alternativeYconstreeStep(i)

        # linking children
        for i in range(t.cs - 1):
            bundleChildren(t.c[0], t.c[i], t.c[i+1])


        shift = (-t.c[0].x+t.c[0].w//2+t.c[t.cs-1].x+t.c[t.cs-1].w//2-t.w )//2 - t.x

        for i in t.c:
            i.x -= shift

        for child in t.c:
            child.edgeCornerHeight = child.lol.y
        # set edgecornerHeights for childen right of the parent
        i = 0
        while i < t.cs-1 and t.c[i].x + t.c[i].w + t.c[i].margin_r - t.w//2 <= 0:
            i += 1

        globalEdgeCornerHeight = t.c[i].edgeCornerHeight
        for a in range(i, t.cs):
            if globalEdgeCornerHeight < t.c[a].edgeCornerHeight:
                t.c[a].edgeCornerHeight = globalEdgeCornerHeight
            else:
                globalEdgeCornerHeight = t.c[a].edgeCornerHeight

        # set edgecornerHeights for childen left of the parent
        i = t.cs - 1
        while i > 0 and t.c[i].x - t.c[i].margin_l - t.w//2 >= 0:
            i -= 1

        if i < t.cs:
            globalEdgeCornerHeight = t.c[i].edgeCornerHeight
            for a in range(i, -1, -1):

                if globalEdgeCornerHeight < t.c[a].edgeCornerHeight:
                    t.c[a].edgeCornerHeight = globalEdgeCornerHeight
                else:
                    globalEdgeCornerHeight = t.c[a].edgeCornerHeight

        # lol update:
        newX = t.c[0].x + t.c[0].lol.x - t.lol.x
        newOutlinepart = Outline(0, t.c[0].lol.y, t.c[0].lol.next)
        t.lol.next.next.next.next = Outline(newX, t.c[0].edgeCornerHeight, newOutlinepart)
        # rol update:
        newX = t.c[t.cs-1].x + t.c[t.cs-1].rol.x - t.rol.x
        newOutlinepart = Outline(0, t.c[t.cs-1].rol.y, t.c[t.cs-1].rol.next)
        t.rol.next.next.next.next = Outline(newX, t.c[t.cs-1].edgeCornerHeight, newOutlinepart)

        # update outline_maxY
        # update min und max for x und y
        for i in t.c:
            t.outline_maxY = max([t.outline_maxY, i.outline_maxY])
            t.minX = min([t.minX, i.x + i.minX])
            t.maxX = max([t.maxX, i.x + i.maxX])
        t.maxY = t.outline_maxY



def moveTreeRoot(t):
    def findMinimalX(t):
        if t.cs == 0:
            return t.x - t.margin_l
        else:
            minSubtreeX = 0
            for i in t.c:
                minSubtreeX = min([minSubtreeX,findMinimalX(i)])
            return minSubtreeX + t.x
    t.x -= findMinimalX(t)

'''
converts all relative coordinates to real ones
'''
def absolutTreeCoords(t):
    if t.cs > 0:
        for i in t.c:
            i.x += t.x
            absolutTreeCoords(i)

def layout(t, strategy):
    match strategy:
        case 'straight':
            YconstreeStep(t)
        case 'bend':
            alternativeYconstreeStep(t)
        case _:
            print("wrong strategy")
    moveTreeRoot(t)
    absolutTreeCoords(t)

'''
Sort the subtrees to form a half-cirle, uses mergesort
'''
def sortSubTrees(t):
    def mergeTrees(l, a, b, r):
        la = l[a:b]
        lb = l[b:r]
        ap = 0
        bp = 0
        for i in range(a, r):
            if (bp < (r-b)) and ((ap >= b-a) or (la[ap].y > lb[bp].y)):
                l[i] = lb[bp]
                bp += 1
            else:
                l[i] = la[ap]
                ap +=1
    def mergesortTrees(l, a, b):
        if a + 1 < b:
            n = (a + b)//2
            mergesortTrees(l, a, n)
            mergesortTrees(l, n, b)
            mergeTrees(l, a, n, b)
    mergesortTrees(t.c, 0, t.cs)


    a = [t.c[t.cs-1]]
    b = []
    a_w = 0
    b_w = 0
    for i in range(1,t.cs):
        if a_w <= b_w:
            a.append(t.c[t.cs - 1 - i])
            a_w += t.c[t.cs - 1 - i].w
        else:
            b.append(t.c[t.cs - 1 - i])
            b_w += t.c[t.cs - 1 - i].w
    a.reverse()

    c = a + b

    for i in range(t.cs):
        t.c[i] = c[i]

# ---------------------------------- example trees ----------------------------
#
# The layout function has the options 'straigt' and 'bend' for the two
# possible modes.
#
# showTree and makeSVG both have an option for showing outlines.
# 'y' = outlines are shown
# 'a' = outlines and the outlines of all subtrees are shown
# anything else = no outlines
#

# first example
n12 = Tree(20, 20, 220, None)
n11 = Tree(10, 10, 220, None)
n10 = Tree(20, 20, 190, [n11,n12])
n9 = Tree(40, 20, 160, None)
n8 = Tree(40, 20, 160, None)
n7 = Tree(40, 20, 160, None)
n6 = Tree(10, 10, 110, None)
n5 = Tree(40, 40, 110, None)
n4 = Tree(10, 10, 70, None)
```

```
n3 = Tree(40, 40, 60, [n7,n8,n9,n10])
n2 = Tree(10, 10, 80, [n5,n6])
n1 = Tree(10, 10, 0, [n2,n3,n4]) #[n2,n3,n4]
layout(n1,'straight')
showTree(n1, 'y')
# you need to set the savedir
#makeSVG(n1, "savedir/", "tree layout", 'a')

# second example
c8 = Tree(40, 40, 110, None)
c7 = Tree(40, 40, 100, None)
c6 = Tree(40, 40, 90, None)
c5 = Tree(40, 40, 90, None)
c4 = Tree(40, 40, 80, None)
c3 = Tree(40, 40, 120, None)
c2 = Tree(40, 40, 70, None)
c1 = Tree(40, 40, 10, [c3, c8, c4, c5, c6, c2,c7])
layout(c1, 'bend')
showTree(c1, 'n')
```

Appendix B

Examples for trees in KIELER

These is an example of a tree for the KIELER tool.

```
algorithm: yconstree

node n1{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 00.0
}
node n2{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 80.0
}
node n3{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 100.0
  algorithm: yconstree
  node i1{}
  node i2{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  node i3{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  edge i1 -> i2
  edge i1 -> i3
}
node n4{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 80.0
}
node n5{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 250.0
}
node n6{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 250.0
}
node n7{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 250.0
  algorithm: yconstree
  node k1{}
  node k2{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  node k3{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  node k4{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  node k5{
    org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
    verticalConstraint: 60.0
  }
  edge k1 -> k2
  edge k1 -> k3
  edge k1 -> k4
  edge k1 -> k5
}
node n8{
  org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"
  verticalConstraint: 400.0
}
node n9{org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"}
node n10{org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"}
node n11{org.eclipse.elk.margins: "top=2, bottom=2, right=2, left=2"}

edge n1 -> n2
edge n1 -> n3
edge n1 -> n4
edge n4 -> n5
edge n4 -> n6
edge n4 -> n7
edge n4 -> n8
edge n3 -> n9
edge n6 -> n10
edge n6 -> n11
```

Bibliography

[Blo93] Anthony Bloesch. “Aesthetic layout of generalized trees”. In: *Software: Practice and Experience* 23.8 (1993), pp. 817–827.

[Bre98] Jack E Bresenham. “Algorithm for computer control of a digital plotter”. In: *Seminal graphics: pioneering efforts that shaped the field*. 1998, pp. 1–6.

[Der14] Atze van Der Ploeg. “Drawing non-layered tidy trees in linear time”. In: *Software: Practice and Experience* 44.12 (2014), pp. 1467–1484.

[Fou23] Eclipse Foundation. *Eclipse layout kernel*. `https://eclipse.dev/elk/`. Accessed: 18.09. 2023.

[HMM+] Reinhard von Hanxleden, Michael Mendler, Christian Motika, Christoph Daniel Schulze, and Steven Smyth. “Sccharts, kieler and the eclipse layout kernel”. In: ().

[HRN03] Masud Hasan, Md Saidur Rahman, and Takao Nishizeki. “A linear algorithm for compact box-drawings of trees”. In: *Networks: An International Journal* 42.3 (2003), pp. 160–164.

[Knu71] Donald E. Knuth. “Optimum binary search trees”. In: *Acta informatica* 1 (1971), pp. 14–25.

[RT81] E.M. Reingold and J.S. Tilford. “Tidier drawings of trees”. In: *IEEE Transactions on Software Engineering* SE-7.2 (1981), pp. 223–228. DOI: `10.1109/TSE.1981.234519`.

[SB07] Benno Stein and Frank Benteler. “On the generalized box-drawing of trees: survey and new technology”. In: *International Conference on Knowledge Management*. Citeseer. 2007, pp. 416–423.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. “Just model!—putting automatic synthesis of node-link-diagrams into practice”. In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE. 2013, pp. 75–82.

[UW89] ABR UGGEMANN-KLEIN and D Wood. “Drawing trees nicely with tex”. In: *Electronic Publishing* 2.2 (1989), pp. 101–115.

[Vau80] Jean G Vaucher. “Pretty-printing of trees”. In: *Software: Practice and Experience* 10.7 (1980), pp. 553–561.

[Wal90] John Q Walker. “A node-positioning algorithm for general trees”. In: *Software: Practice and Experience* 20.7 (1990), pp. 685–705.

[WS79] Charles Wetherell and Alfred Shannon. “Tidy drawings of trees”. In: *IEEE Transactions on software Engineering* 5 (1979), pp. 514–520.