Improving Crossing Reduction with Greedy Switching

Alan Schelten

Bachelorarbeit
eingereicht im Jahr 2015

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme
Prof. Dr. Reinhard von Hanxleden

Betreut durch: Dipl.-Inf. Christoph Daniel Schulze

Abstract

Graphs can be used in the visualization of information in many different fields. Layer-based graph layout distributes vertices among a set of layers and tries to maximize the number of edges pointed in one direction. Minimizing the number of crossings is one of the central aesthetic criteria in automatic layout of such graphs.

In this context, the barycenter heuristic is a popular algorithm to reduce crossings. Characteristic and sometimes obvious crossings remain, however. To remove these, this thesis examines the use of the greedy switch heuristic as a post-processor. This simple algorithm traverses each layer and node and switches neighbouring nodes if this reduces the number of edge crossings. Run separately it runs slowly and returns solutions of bad quality. However when run as a post-processing step to the barycenter heuristic, the algorithm runs fast enough to be feasible for real-time applications. Furthermore the greedy switch algorithm is suitable to remove many of the characteristic crossings remaining after the execution of the barycenter heuristic.

The original implementation of the greedy switch heuristic uses a crossing matrix to store crossings of edges incident to all combinations of two nodes. Calculating this crossing matrix runs in quadratic time. By computing the entries of this matrix on demand the run-time of the heuristic is improved.

The KIELER Layout (KLay) Layered algorithm is a layer-based graph layout algorithm. It uses port-based graphs, where each vertex has ports and each edge connects ports in the graph. Furthermore it allows edges between vertices in a single layer. This results in different causes for crossings. Algorithms for counting crossings for all these cases are developed or improved. Since often only the number crossings of edges incident to two nodes is needed for the greedy switch heuristic, algorithms for counting only these are also described.

Using these, two principle variants of the greedy switch heuristic are compared: Changing node order while considering the crossings to only one or both neighbouring layers. An experimental evaluation shows that the latter runs fast enough to be used in real-time applications, while the former is shown to be more effective in reducing the crossing count but has a slower run-time. Both versions are shown to improve the number of crossings sufficiently to justify permanent integration into KLay Layered.

Eidesstattliche Erklärung

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

Kiel,

Contents

Chapter 1

Introduction

Graphs are an important possibility for visualizing many different types of data, in such varying fields as economy, social science, computer science or mathematics. A common approach to automatic graph layout is layer-based graph layout, which places the vertices on layers and attempts to let most edges go in a left-to-right direction. Figure 1.1 shows such a layered graph. A readable and clear presentation of such graphs is an essential aspect of understanding the data being presented. Readability and aesthetics of the graph are influenced by a number of aspects. One of the most important points is minimizing the number of crossings between edges of the graph, a problem which has been shown to be NP-complete [EW94]. There has been a large amount of research contributed to this topic, introducing many heuristics and some optimal algorithms. Most algorithms sweep through the graph and compares neighbouring layers, assuming the order of vertices to be fixed in one layer (the *fixed* layer) and attempting to minimize

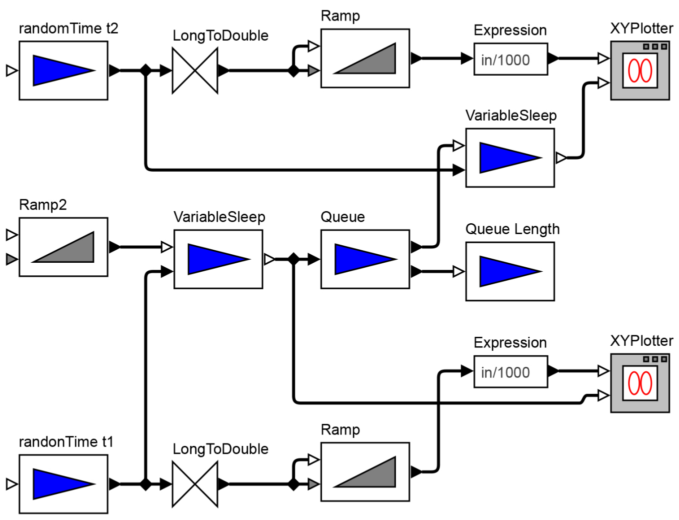


Figure 1.1. Example of a layered Ptolemy graph.[1]Note the data flow direction from left to right.

[1]Ptolemy is a project studying modelling, simulation and design of concurrent, real-time and embedded systems. Its presentation of models uses the KLay Layered Layout algorithms. See `Ptolemy.eecs.berkeley.edu`, accessed 03/28/2015

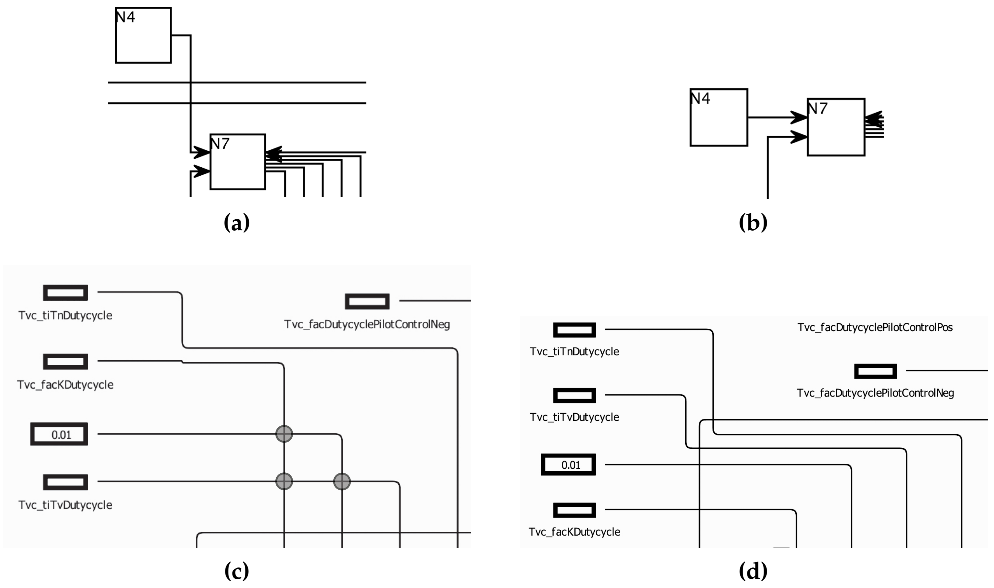


Figure 1.2. (a) and (c) show typical errors of the barycenter algorithm which can be corrected by the greedy switch heuristic, resulting in (b) and (d). The removed crossings in (c) are marked.

crossings caused by edges to the other layer (the *free* layer) by calculating a good order of the nodes. Even when reduced to this task of minimizing crossings by ordering only one part of a bipartite graph, the problem still remains NP-complete [EW94].

The most common algorithm used to minimize crossings is the barycenter heuristic, a method which produces good results and runs fast even for very large graphs. It does, however, return results with characteristic errors, which sometimes can seem painfully obvious to users. These crossings reduce the aesthetic quality of the graphs. Often enough, a simple switch in the order of nodes in one layer would result in a significantly clearer drawing. This thesis follows this intuition by examining the greedy switch heuristic, which switches neighbouring nodes in the free layer as soon as this reduces the amount of crossings. Used separately, this algorithm is slow and often constructs solutions with more crossings than other methods. Based on the intuition that obvious crossings can often be removed by simply switching neighbouring nodes, the algorithm seems like a feasible post-processing step. Before describing the exact contribution of this thesis and giving an overview of the way it is constructed, the following section will examine previous research on the greedy switch heuristic.

1.1 Related Work

The concept of the layer-based layout algorithm originally was introduced by Sugiyama, Tagawa and Toda [STT81]. This algorithm was designed to automatically layout what they called hierarchical systems. This concept has been adapted widely, although the focus in the naming of the method has shifted from the term hierarchy to the fact that the nodes are placed in layers. In this thesis, the term layer-based layout is used.

A major element of this thesis is integrating and evaluating the results of the research into the KIELER Layout (KLay) Layered algorithm. The project provides an open source solution for automatic graph layout and incorporates a wide range of features and possibilities including different types of graph layout algorithms. *KLay Layered* returns a layered layout of graphs and uses the barycenter heuristic. This automatic graph layout algorithm is based on the framework by Sugiyama et al. and is described in detail on their homepage[2]. A comprehensive list of publications dealing with the architecture and specific parts of the algorithm can be found on the same page (e. g., [SSvH14], [Sch11]).

The crossing minimization problem has been researched extensively, with a very large number of publications suggesting heuristics and optimal methods for two-layer crossing minimization (with or without one layer fixed) as well as for the general k-level crossing minimization problem. No current and extensive overview or comparison of effectiveness of the many approaches exists. Jünger et al. published a comparative study for heuristics and exact algorithms of two-layer problem in 1997 [JM97] and Martì et al. published a comparison of heuristics for the same problem in 2003 [ML03], however many new solutions have been suggested since then.

Only few authors seem to be interested in the greedy switch algorithm. The reason for this is obvious: Used as a separate algorithm, greedy switch is slow and gives bad results. This has been shown by theoretical analysis as well as experimental evaluations.

The greedy switch heuristic was originally introduced by Eades and Kelly [EK86] together with the greedy insert heuristic[3] and the split heuristic.[4] Eades and Kelly come to the conclusion that the worst case runtime of greedy switch is $O(|L||E|^2)$, where $|L|$ is the number of nodes in the layer being sorted and $|E|$ is the number of edges. They explain this with the total number of possible crossings being $|E^2|$, noting that *the time complexity is hard to compute*. Chapter 3 discusses this in more detail. In order to know when to switch nodes, they precompute the crossing matrix, which stores the number of crossings between edges incident

[2]`rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KLay+Layered` accessed 03/25/2015

[3]For each step, Greedy Insert chooses a node which minimizes the number of crossings to nodes already chosen and places it as the next node in the layer.

[4]The split heuristic imitates a quicksort approach to order nodes

node i and node j in a matrix entry $c_{i,j}$. The time complexity to compute this matrix is given as $O(|L||E|)$. Their experimental evaluation showed that greedy switching performs poorly on low density graphs and slightly better than the other two suggested algorithms on high density graphs. The paper also shows a pathological example for greedy switching, where no crossings are reduced in a graph which could have a much smaller number of crossings (see Chapter 3 for further details).

Gansner et al. [GNV88] use greedy switch as a post-processing step after applying their variant of the median heuristic.[5] The reason for using the post-processing is similar to the motivation for this thesis: They aim to reduce *obvious* crossings. Astoundingly, they report a 20-50% reduction in edge crossings after executing this step, however without giving any exact numbers. The paper proposes an interesting variation of the heuristic: Usually greedy switch only switches nodes when a switch would reduce the number of crossings. Here, however, on every second forward and backward traversal they also switch nodes when the crossing amount does not change. Supposedly this improves the result. This approach is avoided in this thesis for several different reasons: The implementation chosen in this thesis only terminates sweeping inside one layer when no switch has occured. This would not be possible if nodes are switched without change in crossing number, since this would lead to an endless loop in many cases. Therefore the number of sweeps would have to be predefined. Since the intuition here is that few crossings remain to be removed after execution of the barycenter algorithm, this would either increase the run-time or decrease solution quality. Furthermore, choosing this number is difficult, because in many cases the nodes would simply be switched back to their original positions each time. Adapting this approach could be part of future research however.

Jünger and Mutzel [JM97] did an experimental analysis of eight different crossing minimization algorithms including greedy switch. Note that greedy switch was not used as a post-processor but as a separate algorithm. Jünger and Mutzel also developed an optimal algorithm which allows them to precisely evaluate the success of the heuristics. Once again, greedy switch is shown to fare very badly on low density graphs. On high density graphs its results are close to the optimum. However, the algorithm is shown to significantly slow down in relation to the density of the graph. In some graphs, it is much slower than the optimal algorithm. If the graphs are sparse and the number of nodes are increased, the running time is better than some of the other algorithms. In comparison, the *barycenter heuristic* originally suggested Sugiyama et al. [STT81] and the one implemented in KIELER is very robust: It runs very fast and gives the best solutions compared to the other heuristics examined.

[5]The median heuristic calculates the median of the position of adjacent nodes in the fixed layer and sorts each node in the free layer by the median value. This is similar to the barycenter heuristic (see Section 2.3).

Martì et al. [ML03] did another comparison of heuristics in order to be able to judge the effectiveness of their own developments. Once again they show that greedy switch gives bad results. For example, on test graphs with 10 nodes on each layer, greedy switch has an average deviation from the optimum result of 132%. Their examination confirms the relative success of the algorithm on dense graphs and the very bad performance on sparse graphs. Strangely enough, when testing the experimental running time, greedy switch seems to fare very well even compared to the barycenter algorithm. The authors give no explanation and neither is the source code available to see if any optimizations were attempted. There seem to be two possible explanations: First of all the time is given in tenths of seconds in order to be able to compare the running time to the optimal algorithm and close to optimal heuristic suggested by the authors which, not surprisingly, run much slower than the other heuristics. For the test cases given however, this unit is not exact enough. The second possible explanation could be that many of the results shown are experiments run on sparse graphs, where the greedy switch algorithm has little to do (see Chapter 3).

Martì et al. also take into account combinations of algorithms, namely: Semimedian heuristic[6] and greedy switch, split and greedy switch, barycenter and split, first greedy switch then split and finally barycenter and greedy switch, which is the variant we will be discussing here. This last option results in a slight improvement in relation to the optimal number of crossings with no significant change in computation time. The combination of barycenter with greedy switch seems to be the best compromise between speed and precision, although the difference to the barycenter method is small.

As a side note, many further heuristics have been suggested which are not compared in this paper, many of which give better results than the barycenter algorithm. None of these, however, run faster (i. e., [YS99], [BF06], [Cat95], [LMV97]).

To the best of my knowledge, the greedy switch algorithm has not been examined further. This overview showed that only little research has been done on the greedy switch algorithm. However, the suggested approach of using greedy switch as a post-processing step has been investigated by some authors.

Let us now turn our attention to what contribution this thesis makes.

1.2 Problem Description

This thesis examines the consequences of implementing the greedy switch algorithm as a post-processing step to the barycenter heuristic.

[6]The semimedian heuristic is a hybrid of the barycenter and median heuristic introduced by Mäkinen [Mä90].

The intuition justifying this approach is supported by the observation that the current barycenter heuristic often results in obvious errors. Therefore the barycenter heuristic and some characteristic errors are examined.

The characteristics of the greedy switch algorithm are considered and variants developed which aim at optimizing speed or solution quality of the greedy switch heuristic. To be able to implement the heuristic, the exact number of crossings between neighbouring nodes and for complete layers need to be calculated. Existing crossings counting algorithms in KLay Layered are examined and when necessary, new ones are developed.

Using the results of an experimental evaluation, recommendations for a final integration into KLay Layered are given.

1.3 Outline

This thesis is structured in the following manner:

Chapter 2 defines terminology and gives an introduction into the context forming the basis for this thesis. This includes the general principle of layer-based graph layout and the barycenter heuristic. To set the stage for the following chapters, typical errors of the barycenter algorithm are discussed and presented.

Chapter 3 discusses the greedy switch heuristic from different perspectives. The general form of the algorithm is described and theoretical run-time and pathological cases are discussed. On an abstract level, different approaches to the algorithm are presented.

Stepping down in the level of abstraction, Chapter 4 discusses methods for counting edge crossings, which is the main algorithmic challenge in implementing the greedy switch heuristic. To do this, different types of edges are considered i. e., edges passing between layers, edges passing between nodes in the same layer and edge crossings resulting from the order of north/south ports on a node. For each of these the crossings in the complete layer and for neighbouring nodes are discussed separately. Finally, the problem of hyperedge crossings is addressed. For all of these cases, existing algorithms are presented and where necessary, new algorithms are developed.

Chapter 5 gives a short introduction into the architecture of KLay Layered and explains the architecture chosen for the implementation of the greedy switch heuristic as a post-processor.

The experimental evaluation in Chapter 6 shows the results of practical experiments which judge both quality and speed of the implementation. Recommendations on the configuration of the greedy switch heuristic are given.

Finally, Chapter 7 summarizes the results and suggests further venues of research related to the topic of this thesis.

Chapter 2

Preliminaries

2.1 Terminology

Definitions and terms used across this thesis are collected here, so should the meaning of a particular word or symbol be forgotten, this is the place to look.

Definition 1. Ports and Port-Based Graph
A *port-based graph* is a tuple $G_p = (V, E, P, vp)$. Let $D = \{n, e, s, w\}$ (D as in ***D**irection*) be the set of sides a port can be on. The letters n, e, s, w in the set D stand for *north, east, south, west*. P is the set of ports. Using these, the function $vp : P \rightarrow V \times D = \{n, e, s, w\}$ maps ports to nodes and the side of the node the port is on. In order to simplify the algorithms for counting crossings in a graph, we will be using graphs with ***undirected** edges* connecting ports $E = \{p_1, p_2\}, p_1, p_2, \in P$. Even though data flow graphs obviously use directed edges, the direction of the edges is not of importance for this thesis. We will use the term nodes instead of vertices to stay consistent with the KLay Layered source code.

To be able to easily describe all ports on a given side of a layer, $P_d, d \in D$ shall be a list containing all ports on side d sorted by their position in the layer (see Definition 2).

Definition 2. Position Functions
Each node and also each port has a *position*, i. e., in each layer ports and nodes are numbered from north to south when on the eastern or western side of a node and east to west when on the northern or southern side. Formally, functions $pos_p : P \rightarrow \mathbb{N}$ and $pos_v : V \rightarrow \mathbb{N}$ map ports or nodes to their indexes.

Definition 3. Connected Edges
Edges connected to a node on a given side d shall be written as $conn_d(v)$. Edges connected to a port shall be written as $conn(p)$.

Definition 4. Switch
The greedy switch heuristic works by *switching* the order of nodes, when this reduces the amount of crossings. Switching thereby changes the values of the position functions. Formally, let pos_v and pos_p be defined as above. Switching nodes n_1 and n_2 with $pos_v(n_1) < pos_v(n_2)$ results in changed function pos'_v and pos'_p with $pos'_v(v_1) = pos_v(v_2)$ and $pos'_v(v_2) = pos_v(v_1)$. For the port positions,

the value functions must also change to reflect the new position of the ports in the layer: $\forall p : vp(p) = (v_1, d), d \in D, pos'_p(p) = pos_p(p) + |conn_d(v_2)|$ and $\forall p : vp(p) = (v_2, d), d \in D, pos'_p(p) = pos_p(p) - |conn_d(v_1)|$.

Definition 5. Fixed Port Order
Port ordering is *fixed* when the input graph defines the order of ports for each node. Port ordering is *free* if the layout algorithm can switch the order of ports on the node. In our case a free ordering of ports means that while the order can be switched, the side of the node on which the ports are situated must remain the same. To show free port order, each port on a node $v \in V$ and a side $d \in D$ with free port order has the same position value i. e., for $P_{v,d} = \{p \mid p \in P \wedge vp(p) = (v, d)\}$ it must hold that $pos_p(p'_v) = pos_p(p''_v)$ for all pairs $p'_v, p''_v \in P_v$. This means that we do not count crossings caused by the ordering of ports and assume that a port ordering algorithm can remove all of these crossings. As shown in Section 4.2, this assumption is not always true.

Definition 6. Layer
In layer-based graph layouts, there are $n \in \mathbb{N}$ layers $L_i, i < n$. These partition the set of nodes, i. e., each node is contained in exactly one layer. The nodes in a layer are a tuple $L_i = (v_0, \dots, v_{k_i}), k_i \in \mathbb{N}$, where k_i is the number of nodes in L_i. The layered graph shall be denoted by the tuple $\mathbb{L} = (L_0, \dots, L_{n-1})$ Layers L_i, L_j are *neighbours* or *neighbouring* if $|i - j| = 1$.

In contrast to most publications on crossing minimization I will assume a left to right or east-west layout of the layers. This is because KLay Layered is mostly used to layout data flow diagrams, where the left-right direction is more common.

In most algorithms the problem of minimizing edge crossings is reduced to the subproblem of minimizing crossings between two neighbouring layers. The order of nodes in one of these two layers is held fast, while the order of nodes in the second layer is changed. This second layer shall be called the *free layer*, or L_f, as opposed to the *fixed layer* or L_x.

Definition 7. In-layer and Between-layer Edges
Edges are either *between-layer* or *in-layer*. They are between-layer when they pass from one layer to a neighbouring layer and in-layer when they connect nodes in the same layer. Formally, an edge is an in-layer edge, when for edge $e = (p_1, p_2), p_1, p_2 \in P$ and a side $d \in D$ it holds that $vp(p_1) = (v, d), vp(p_2) = (v', d), v, v' \in L$. Between-layer edges pass between neighbouring layers: An edge e is an in-layer edge iff $e = (p_1, p_2), p_1, p_2 \in P : vp(p_1) \in L_i \wedge vp(p_2) \in L_j \wedge L_i$ and L_j are neighbours

Definition 8. Long Edge Dummies, North/south-port Dummies
In order to simplify the problem of crossing minimization to two layers, edges traversing more than one layer are replaced by a chain of long-edge dummies, one for each layer.

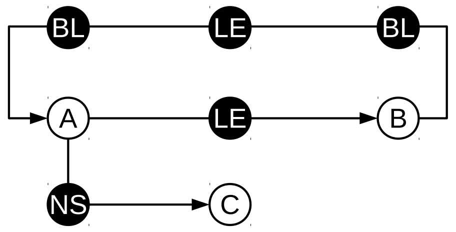


Figure 2.1. The feedback edge from A to B creates two in-layer edge dummies (IL). When traversing more than one layer an edge is split up into portions separated by long-edge dummies (LE). When a node has an edge connected to its north or south side, a north/south edge dummy is created (NS).

When an edge is incident to a port on the north or south side of a node, a north/south port dummy is added to show where the bend in the edge will be drawn. See Figure 2.1 for examples of in-layer edge dummies, long edge dummies and north/south port dummies.

Definition 9. Adjacency List
An *adjacency list* $A(v,d)$ to a node v on side $d \in D$ is a sorted list of ports connected by edges to ports on node v on side d. Formally, $A(v,d) = (p_0, \ldots, p_{n-1}), \forall i < n, \exists \{p_i, p'\} \in E, vp(p_i) = (v,d)$ and $\forall p_i, p_{i+1}, i < n-1, pos_p(p_i) < pos_p(p_{i+1})$.

Definition 10. Crossing Matrix
The *crossing matrix* for a free layer L_f and a fixed layer L_x is a matrix $(c_{i,j})_{i,j<|L_f|}$ where each entry $c_{i,j}$ saves the number of crossings of edges incident to two nodes v_i, v_j where v_i is above v_j, or: $pos_v(v_i) < pos_v(v_j)$. Equivalently the entry $c_{j,i}$ saves the number of crossings incident to those nodes when v_j is above v_i.

Definition 11. Hypergraph, Hyperedge
In accordance with Spönemann et al. [SSRvH14], a directed *hypergraph* is a pair $G = (V, H)$ where $H \subseteq P \times P$ is a set of *hyperedges*. Each $(S, T) \in H$ has a set of *sources* S and a set of *targets* T. KLay Layered supports the drawing of orthogonal hyperedges.

With these definitions all set, we will have a look at the layout method our algorithm will be a part of.

2.2 Layered Graph Layout

The following chapter will give a short introduction to some basic background information concerning the type of graph layout in question and the basic algorithm framework used for automatic layouting, called KLay Layered.

The goal of KLay Layered is to construct a graph layout which is suited for graphs which have an inherent edge direction, such as data flow diagrams. Data flow diagrams are used to demonstrate or model the flow of data through components of a given system. As an example consider the one given above in Figure 1.1.

As the name suggests, KLay Layered uses a *layered* approach to automatic graph drawing. This means that nodes are distributed among a set of layers. In order to make the graph as readable as possible, the algorithm tries to optimize different criteria, the most important being:

- ▻ Maximize the number of edges directed from left to right, following the reading direction.
- ▻ Minimize crossings between edges.
- ▻ Minimize edge bends.

KLay Layered is based on an algorithm developed by Sugiyama, Tagawa and Toda [STT81]. This algorithm was designed to automatically layout what they called hierarchical systems. The term *hierarchy* assumes that the first level of the drawn layout is the top of a hierarchy and the directed edges show the nodes further down in the hierarchy. For this reason, Sugiyama et al. draw their graphs in a top-down order. This principle has been followed in most of the following literature. As has been said, we will consider a left-to-right layout, since the goal of KLay Layered is to layout data flow diagrams.

The following section only gives a very short and very general overview of the Sugiyama framework. For a more detailed description, see the original paper [STT81].

The algorithm is divided into four steps:

1. Distribute nodes to layers and replace long edges with node dummies.
2. Permute order of nodes.
3. Position nodes horizontally.
4. Remove dummy edges, replace with edge bends.

A hierarchical ordering of graphs is considered to be a *proper* layout when edges always go in a left-right direction and always go from one layer to a neighbouring layer. To make this possible, graphs layouted this way cannot contain cycles. After distributing the nodes across the different layers, any edges going further than to the neighbouring layer are broken by dummy nodes. These are nodes which in step 4 will be removed and replaced with edge bends.

The second step reduces the number of edges crossings. This is the main concern of this thesis and is more carefully discussed in Section 2.3 and Chapter 3.

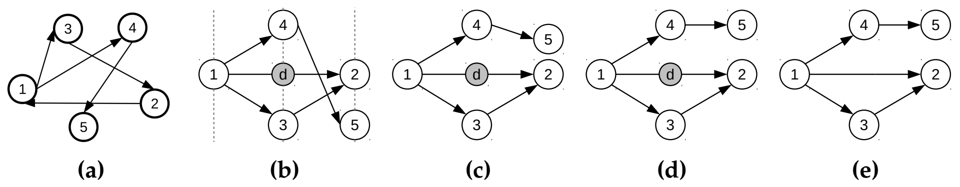


Figure 2.2. Steps in Sugiyama algorithm: (b) Adds dummy edges and assigns layers. (c) Reduces edge crossings. (d) Reduces edge bends. (e) Removes dummy nodes.

The third step positions the nodes horizontally in each layer in order to reduce the number of edge bends. Figure 2.2 shows the traversal of all the steps in the algorithm.

The following chapter will consider step two more closely: The *Barycenter Heuristic* is suggested by Sugiyama et al. [STT81] and is used in KLay Layered to reduce the number of crossings in a layered graph.

2.3 Barycenter Heuristic

As shown by Eades and Wormwald, minimizing the number of edge crossings by permuting nodes in a layered graph is NP-complete [EW94].

The common approach is to reduce the difficulty of problem by only considering a pair of layers at a time. As determined in Definition 6, one of these is the free layer L_f, whose node order is permuted, and the other is the fixed layer L_x, whose node order is held fast, a problem which should be easier but unfortunately is also NP-complete [EW94].

In order to reduce edge crossings in the whole graph, many heuristic algorithms sweep from left to right and backward across the layers until no further improvement of the number of crossings is achieved, as shown in the algorithm *layerSweep* (Algorithm 1).

Depending on the heuristic, at least the permutation of the first fixed layer needs to be set before one can sweep forward and backward across the layers. The simplest way to do this is to choose a random order. Obviously some choices for the order of the first layer may be better than others. To take this into account, one can simply run the algorithm with several randomized values and take the best result.

To choose a good permutation with feasible computation time, very many different heuristics have been suggested. Sugiyama et al. developed an algorithm which they called the *barycenter heuristic* [STT81]. This heuristic is fast in theory and in practice and gives good results, which is why KLay Layered uses this algorithm (for experimental evaluations see the research discussed in Section 1.1).

The name *barycenter* is a synonym for the centre of mass, or the point where

Algorithm 1: layerSweep

```
   Data: Layers $\mathbb{L}$
   Result: Reordered layers $\mathbb{L}_{bestOrder}$
1  lastCrossings = $\infty$;
2  currentCrossings = countCrossings($\mathbb{L}$);
3  forward = true;
4  while $lastCrossings > currentCrossings$ do
5      $\mathbb{L}_{bestOrder}$ = copy of $(L_0, \dots, L_n)$, storing the current order of each layer
6      if $forward$ then
7          for $i = 0$ to $n - 2$ do
8              $L_x = L_i$
9              $L_f = L_{i+1}$
10             $L_i$ = permute($L_x, L_f$)
11     else
12         for $i = n - 1$ to $1$ do
13             $L_x = L_i$
14             $L_f = L_{i-1}$
15             $L_i$ = permute($L_x, L_f$)
16     forward != forward;
17     lastCrossings = currentCrossings;
18     currentCrossings = countCrossings($\mathbb{L}$)
```

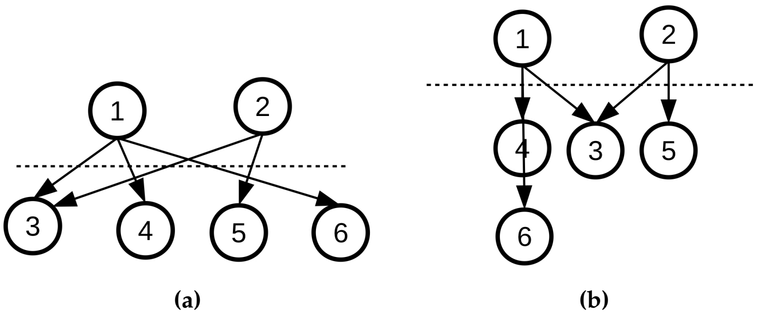


Figure 2.3. In b) the nodes the nodes are shown in the positions of their barycenters, or where the edges would be as short as possible. In the case of nodes 4 and 6, a random order is chosen, since both have the same barycenter.

the sum of gravitational pulls equals zero. This shows the general principle of the algorithm. For each node in the free layer, its neighbours in the fixed layer generate a *gravitational force* which pulls the free node to the barycenter of these forces, minimizing the length of the edges. The *barycenter* of a node is simply

Algorithm 2: barycenterPermute. Remember that $A(v, d)$ is an adjacency list, see Definition 9

```
Data: Fixed layer $L_x$, free layer $L_f$, Side $d \in D$ showing which side of the
      free layer the fixed layer is on
Result: Reordered free layer $L_{f_{reord}}$
1 $bary := V \rightarrow \Re$ mapping nodes to their barycenter value
2 for Node $v \in L_f$ do
3     if $N_{L_x}(v) \neq \emptyset$ then
4         $bary(v) := \sum_{v_n \in A(v,d)} \frac{pos_v(v_n)}{|A(v,d)|}$
5     else
6         $bary(v) :=$ random value between 0 and $|L_x|$
7 $L_{f_{reord}} = L_f$ sorted by $bary(v), v \in L_f$. Same values in random order.
```

calculated as the average of its neighbour's positions. The nodes are then sorted by their barycenters. Nodes with equal barycenters are sorted randomly. The algorithm barycenterPermute (see Algorithm 2) shows how simple this algorithm is. In a port-based graph this algorithm is simply changed to iterate through all ports on all nodes on the side of the free layer and use the pos_p function for ports (see [SFvHM10]).

The barycenter heuristic scales well. It calculates the barycenters in linear time and can then use efficient sorting algorithms, resulting in a running time of $O(|E| + |V_f|\, log|V_f|)$ ($|E|$ to calculate the barycenters and $|V_f|\, log|V_f|$ to sort the nodes).

The framework described by Sugiyama et al. only takes between-layer edges into account. For practical implementations however, and also in the case of KLay Layered, we must take in-layer edge and north-south port crossings into account as well.

Schulze [Sch11] suggested changes to the barycenter algorithm for in-layer edges. For each node v with an in-layer edge, the node v_{IL} connected to it by the in-layer edge influences the barycenter value of v. The position value used for calculation of the barycenter of v is the barycenter value of v_{IL}.

If the barycenter has not yet been calculated for v_{IL}, the barycenter calculation algorithm is recursively called for v_{IL}. To avoid an endless loop the in-layer edge to v obviously needs to be ignored.

When calculating the positions of north/south dummy nodes, the barycenter algorithm in KLay Layered does not currently take the positions of north and south ports on a node into account. Conceivably, the algorithm could be changed so that barycenter values of these dummy nodes are in some way influenced by the position of the connected north/south ports and the direction of their between-layer edges. This would go beyond the scope of this thesis, but might be

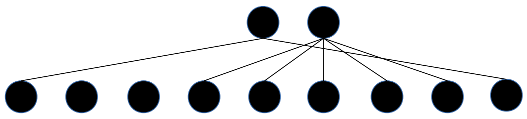


Figure 2.4. Simple pathological example for the barycenter heuristic. By switching the rightmost vertex in the bottom layer with the five vertices to its left, the graph would contain no crossings.

an interesting venue for further work in improving the results of the barycenter algorithm in KLay Layered.

All in all, the current method used in KLay Layered is fast and good, so why would one want to modify it? The following chapter will show some typical weaknesses of the barycenter algorithm.

2.3.1 Errors of the Barycenter algorithm

Let us now turn our attention to typical errors of the barycenter heuristic.

Assuming there are only two layers and one layer is fixed, and any configuration of nodes is possible in the fixed layer, we can easily construct pathological examples where the barycenter algorithm returns very bad results (see Figure 2.4).

In the context of the Sugiyama framework, such configurations are mostly prevented. The unconnected nodes can very well be put in a different layers, changing the values of the barycenters.

Let us instead consider an example of a characteristic situation in graphs layouted with KLay Layered, where obvious crossings remain.

Figure 2.5a shows a case where node `N4` could be set directly beside `N7` to reduce two crossings. Why does the barycenter algorithm not do this? After all, the *center of gravity* metaphor suggests that the edge connecting `N4` and `N7f` should be as short as possible. The reason for this is that the barycenter algorithm can in some cases increase the number of crossings. Consider a backward sweep of the barycenter algorithm across Figure 2.5a. Let the fixed layer and the free layer be as marked in Figure 2.5a. In this case, all dummy nodes for the long edges and nodes `N5` and `N3` are set to the same barycenter value, because their only neighbour in the fixed layer is `N9`. As can be seen in Algorithm 2, in this case the order of the nodes is determined randomly.[1] However, if for example the nodes which are connected to `N7` (`N5`, `N3` and two dummy nodes) are set further to the top of the layer, the node `N7` will follow and might cause a higher number of crossings than before, for example with the edge to node `N1`.

As a matter of fact, the result created by the algorithm would be significantly

[1]Note that the randomization seed used to generate random numbers by KLay Layered is kept constant for each graph to keep the layout consistent for the user.

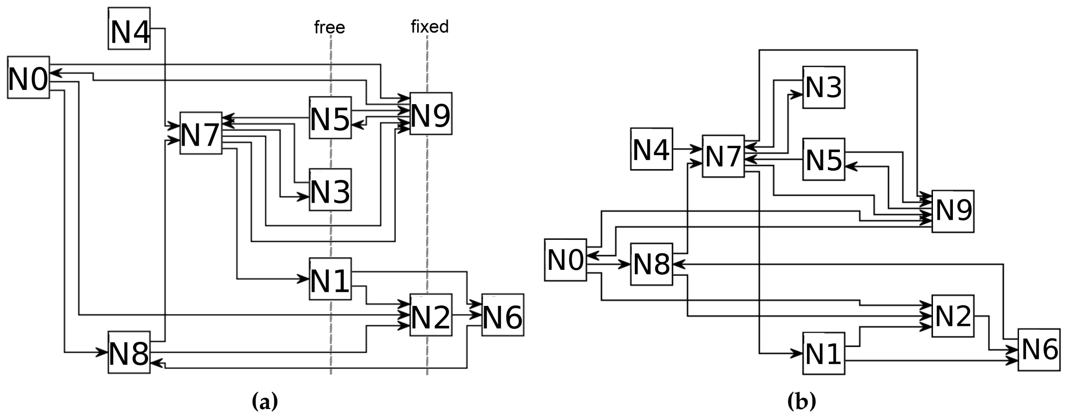


Figure 2.5. Part of a randomly generated graph showing a typical problem of the barycenter heuristic. A backward sweep of the barycenter algorithm would result in b).

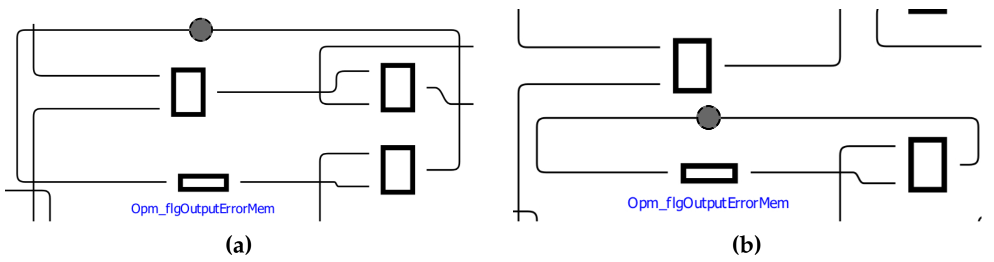


Figure 2.6. Part of a randomly generated graph showing a typical problem of the barycenter heuristic. A backward sweep of the barycenter algorithm would result in b).

worse, as can be seen in Figure 2.5b. In this case the ordering of the nodes neighbouring to N2 were also changed randomly and resulted in five crossings as opposed to the previous three. To prevent the heuristic worsening the number of crossings already achieved, after each forward or backward sweep the best order is saved, as can be seen in layerSweep (Algorithm 1). As soon as the crossing number is increased, the algorithm is aborted and the last saved layout taken. So since the N4 node in Figure 2.5a was never reached by a backward sweep, it stays where it was.

The position of dummy nodes which are part of in-layer edges is also determined by a barycenter value. This case is therefore prone to the exact same type of error as the between-layer edges. See Figure 2.6 for an exemplary error.

The implementation in KLay Layered does not sort north/south port dummies with respect to their connecting north/south port on the normal node. This results in many unnecessary crossings, as can be seen in the example given earlier in Figure 1.2c.

Errors due to this randomization seem to be quite common for the barycenter algorithm and can be reduced if the number of executions with different random values is increased. Obviously this increases running time by whichever number of repetition is chosen.

I believe that these types of edge crossings are specifically problematic, because humans can quickly and simply see how to improve the layout. In my opinion, the possibility of obvious corrections significantly reduces the aesthetic quality of the graph, a sentiment shared by users of KLay Layered who have provided feedback.

After this quick look at the current situation, the next section will describe which solutions the greedy switch algorithm as a post-processing step could offer.

Chapter 3

Greedy Switch Heuristic

The following chapters give a closer look at the principle of the greedy switch heuristic and variants to be considered when implementing it.

3.1 Algorithm

In the context of the layered approach to graph drawing, the heuristic is very quickly explained: For two neighbouring nodes, check to see if by exchanging their positions (*switching* them) the number of crossings is reduced. If it is, switch them, if it is not, don't. This principle is continued throughout the graph for all nodes in each layer.

The greedy switch heuristic was introduced by Eades and Kelly [EK86]. In their paper, they suggest three different algorithms for drawing what they call *two-layered networks*. This is a graph $G = (V, E)$ with two layers $\mathbb{L} = (L_1, L_2)$ with only *between layer edges*, i. e., edges passing from one layer to the other. Similar to the barycenter heuristic, the order of one layer is fixed (L_x) and the order of the other is free (L_f). The nodes in the free layer are permuted to reduce crossings. This is done using *greedyPermute* (Algorithm 3).

Naturally, two-layered networks are not a real application. For general layered networks the heuristic suggested by Eades and Kelly sweeps backwards and forwards across the graph, always considering two neighbouring layers. This is

Algorithm 3: greedyPermute

```
Data: Fixed layer $L_x$, Free layer $L_f$
Result: Reordered free layer $L_f$
1 continueSweeping := true
2 while continueSweeping do
3     continueSweeping := false
4     for neighbouring nodes $v_{upper}, v_{lower}$ in $L_f$ do
5         if switching $v_{upper}$ and $v_{lower}$ reduces number of crossings then
6             Switch values of $pos_v(v_{upper})$ and $pos_v(v_{lower})$
7             continueSweeping: = true
```

the exact same principle as was described earlier for the barycenter algorithm. In fact, there is no need to reprint the algorithm: *layerSweep* (Algorithm 1) can simply be executed just the same as with barycenter, using *greedyPermute* (Algorithm 3)) as the *permute* algorithm.

3.2 Quality and Speed

What is the running time of the greedy switch heuristic? Although it looks very similar to *Bubblesort*, there is no linear order of the nodes at which the algorithm will always terminate. It ends when has reached a stable ordering, where switching neighbouring nodes does not improve the crossing count. Since the heuristic only continues when a switch occurs the maximum number of possible traversions of the free layer is $|L_f|^2$, in the case where only one node is switched per traversion and each node is switched with every other node. This results in a worst case running time of $O(|L_f|)^3$.

However, actually constructing a pathological case where only one node is switched per sweep and all nodes are switched with each other is difficult or impossible and cannot be found in the literature. As Eades and Kelly [EK86] put it, the actual time complexity of greedy switch *is hard to compute*.

It is much simpler to construct pathological examples showing low solution quality. See for example Figure 3.1 where the resulting number of crossings is quadratic to the optimal number of crossings.

As has been shown in different experimental evaluations (see Section 1.1), the speed of the greedy switch heuristic deteriorates significantly when the density of the graph increases, while the difference in the number of crossings to an optimal ordering is decreased.

Since run-times of the algorithms counting crossings between nodes depend on the number of edges (see Chapter 4) the speed decrease is obvious. There is another reason for both speed decrease and quality increase: In sparse graphs, greedy switch can easily be prevented from correcting obvious errors. Consider the small example in Figure 3.2, where the nodes in the middle will not move, since switching them with neighbouring nodes would not remove any crossings.

As the literature overview shows (see Section 1.1), past experimental results have shown how badly the greedy switch heuristic fares compared to other algorithms when used as the only algorithm to reduce edge crossings. However, the motivation for the approach chosen here is to use the heuristic as a post-processing step after executing the barycenter algorithm. As shown in Section 2.3, this approach results in graphs with few crossings reasonably close to the optimal ordering, while keeping some (often *obvious*) crossings due to characteristic errors of the algorithm.

As a post-processing step, the greedy switch heuristic should therefore en-

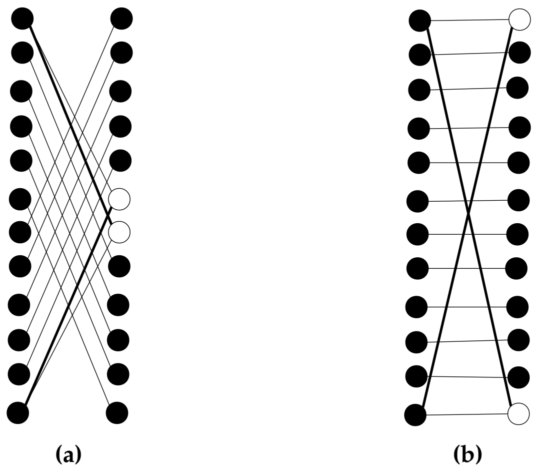


Figure 3.1. Pathological example for the greedy switch heuristic. The bold edges going to the two white nodes on the right layer (the free layer) prevent the greedy switch algorithm from doing anything in (a), because no switch of neighbouring nodes would reduce the crossing amount. However the optimal ordering shown in (b) has much fewer crossings. In numbers: (a) has $\left(\frac{|L_f|}{2}\right)^2 + 2 * \left(\frac{|L_f|}{2} + 1\right) = 50$ crossings as opposed to $2 * (|L_f| - 2) + 1 = 19$ crossings in (b).

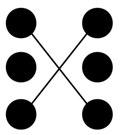


Figure 3.2. The nodes cannot be switched by the greedy switch heuristic because edges incident to neighbouring nodes have no crossings. Very simple crossings can remain.

counter a graph where only few crossings remain. Since it only continues to sweep through the free layer if at least one pair of nodes have been switched, the algorithm will run in linear time if no further crossings can be removed (assuming a precomputed crossing matrix, see Section 3.3.3).

Furthermore, in my opinion, the greedy switch heuristic is similar to the human approach of recognizing and removing unnecessary crossings in graphs. Consider a further example of an error of the barycenter algorithm (Figure 3.3). The unnecessary crossing caused by the position of the J node is strikingly obvious because one could simply pull it upwards next to VectorIntegrator2. This might also be the reason for the crossing being so obvious in the first place.

In summary, the greedy switch algorithm might specifically be well suited for resolving *obvious* errors caused by the barycenter heuristic and might have reasonable computation time when run as a post-processing.

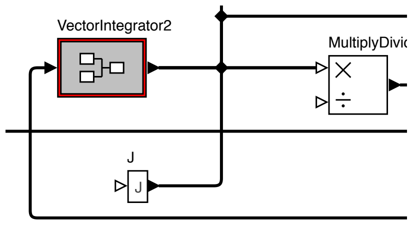


Figure 3.3. The crossing of the edge connected to the J node is so strikingly obvious to be especially bothersome.

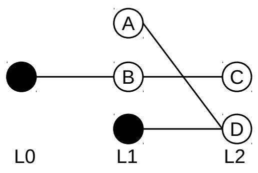


Figure 3.4. Example of an obvious crossing which would only be removed in a backward sweep of the one-sided variant.

The main algorithmic problem of greedy switch, namely deciding when to switch, still has not been addressed. The next section will take one level of abstraction further downward and compare six different general variants for this problem.

3.3 Deciding When to Switch

3.3.1 Fixing one Neighbour Layer or Both

As described above (Section 3.1), the originally proposed greedy switch heuristic uses the same layer sweep algorithm as the barycenter heuristic. This means that for each node in the free layer, the number of crossings to one of the free layer's neighbours is reduced.

Note that in some cases a forward sweep does not reduce any crossings, while a backward sweep would. Consider Figure 3.4: If L0 is the fixed layer and L1 the free layer there is no reason to switch nodes A and B. If L1 is the fixed layer and L2 the free Layer, nodes C and D are not switched because the number of crossings would remain the same. Only on a backward sweep when L2 is fixed and L1 free, would the obvious crossing be removed. Therefore in order to improve the results, the implementation chosen here always sweeps forwards *and* backwards before counting the number of crossings. Incidentally, the examples for bad results of the barycenter algorithms given so far (Figure 2.5a and Figure 3.3) would not be improved if the heuristic would stop after a forward sweep.

In the case of greedy switching (as well as with the barycenter heuristic),

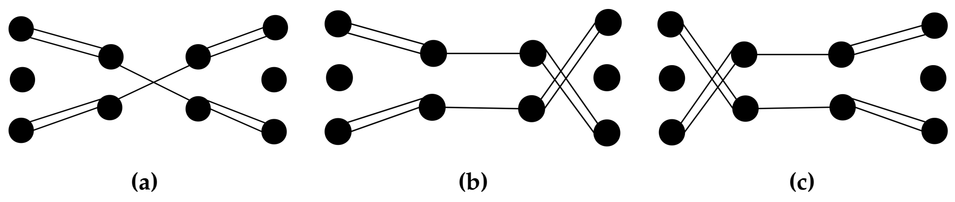


Figure 3.5. (a) shows the graph in its first layout with one crossing. (b) shows the graph after a forward sweep of the one-sided variant and (c) after the following backward sweep. If the algorithm aborts when no switch has taken place in a sweep like the two-sided variant does, this situation would end in an endless loop. If the algorithm aborts after a certain number sweeps, the amount of crossings would always be increased.

executing the heuristic can lead to an increase in the number of crossings. Consider the example in Figure 3.5. In order to prevent an increase in the number of crossings, the *layerSweep* algorithm (Algorithm 1) counts the number of crossings in the graph after each sweep and aborts if it has not improved.

Not surprisingly, counting the number of crossings in the whole graph is an expensive operation (this is discussed in detail in Chapter 4). There is however a simple variant of this heuristic which guarantees that nodes are only switched if they can not increase the number of crossings in the graph. To do this one can simply take into account the crossings to both neighbouring layers and only switch if the total number of crossings is reduced. I call this the *two-sided* approach as opposed to the usual variant, which equivalently will be named *one-sided*.

In this case we can abort after either a forward or backward sweep, just as in the normal *layerSweep* algorithm, also saving computation time. The examples of obvious barycenter errors named above would be successfully improved by a single forward sweep of the two-sided variant (Figure 2.5a, Figure 3.3 and Figure 3.4).

However, putting speed considerations aside, the two-sided approach in some cases leads to fewer reductions in edge crossings than the one-sided would.

Consider the simple example in Figure 3.6: With one-sided greedy switching, nodes A and B would be switched and the crossing would propagate to the next layer, where, after switching the right-most nodes, it would disappear. With two-sided greedy switching on the other hand, A and B would not be switched, because the number of crossings in the layers neighbouring the free layer would remain the same. The situation in Figure 3.6 happens quite often in the Sugiyama Framework, because edges spanning more than one layer (called *long edges* in KLay Layered) are broken by dummy nodes in each layer. Therefore, the two sided variants could not remove crossings by switching long edges with each other.

A major difference in the abilities of the two variants occurs when considering

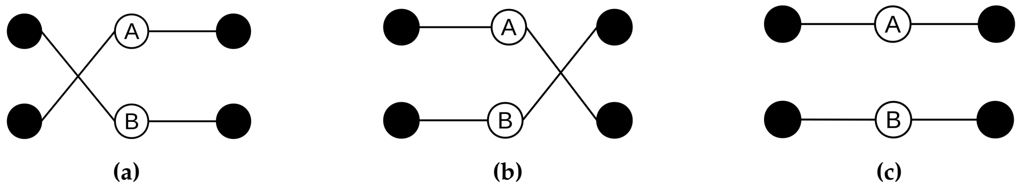


Figure 3.6. Assuming that the order of the nodes in first layer cannot be changed, the one-sided variant results in the layout seen in (c), while the two-sided variant would not do the step from (a) to (b), because the number of crossings is the same in both situations.

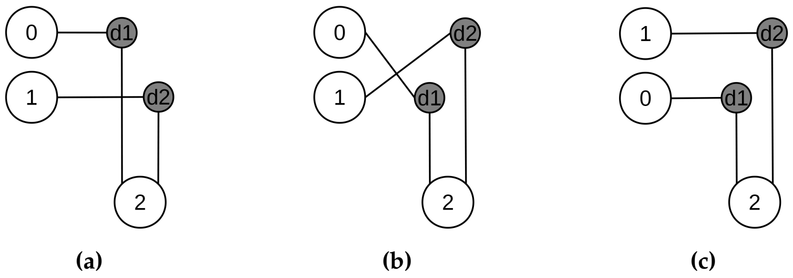


Figure 3.7. The one sided variant would execute the steps from (a) to (c) on a backward sweep, while the two-sided variant would not be able to change the graph because in the step from (a) to (b), the number of crossings is not reduced. If for some reason nodes `0` and `1` can not switch their positions, the situation in (b) would remain.

the crossings which can be caused by the ordering of north-south ports. Switching the order of north-south port dummies always causes crossings of between-layer edges to the neighbouring layer. Therefore, even if a north-south port crossing is reduced by switching north-south dummies, the two-sided variant will never change their order, because the number of crossings would remain the same. Only the one-sided variant will therefore switch north-south port dummies. In the case where the position of node connected to the north-south dummy in the next layer cannot be changed without increasing the number of crossings further, this could result in a less aesthetically pleasing drawing: The edge at the north/south port dummy would either not be in a 90° angle or, if the edges are drawn orthogonally, an extra edge bend would be added without actually reducing edge crossings.

One can also construct cases in which a better order of nodes in a single layer is rejected by the one-sided approach while the two-sided corrects the error, see Figure 3.8. These examples however, are more of a hypothetical nature.

In summary, we now have two versions of greedy switch, the two-sided and one-sided variant, which switch different nodes and where the two-sided version is certainly faster. The success in reducing crossing numbers and the actual speed of these two are examined in Chapter 6. The question which still remains is

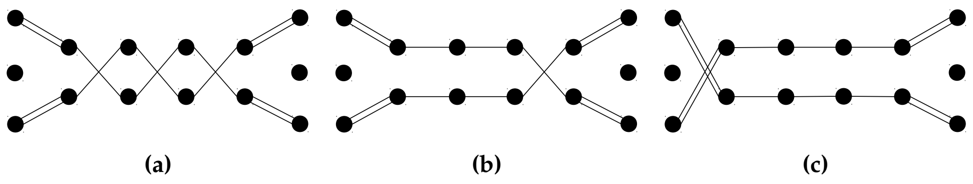


Figure 3.8. The two-sided variant results in the layout seen on the in (b), while the one-sided variant would end in (c). Since the four crossings here are more than the three in the original graph (a), the one-sided variant would not change the original graph.

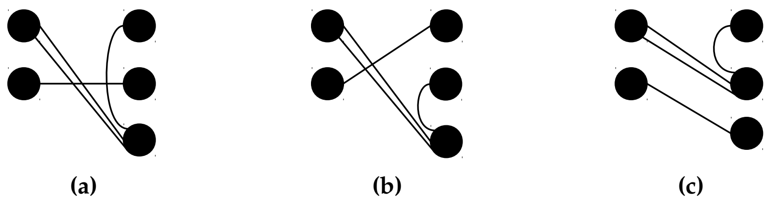


Figure 3.9. (a) shows the original layout. Assuming the order of the left layer is fixed, (b) shows the result when sweeping through the right layer from top to bottom, reducing one crossing but keeping two. (c) shows the result when sweeping from the bottom to the top of the layer, removing all crossings.

how and when to count which crossings to be able to choose if a switch would help or not. The next section will describe the *when* and *which*, while the *how* is sufficiently complex to be put into its own chapter (Chapter 4).

3.3.2 Comparing Sweep Directions in a Single Layer

In the usual form of the algorithm as described in Section 3.1, for each layer we start stepping through neighbouring pairs of nodes starting with the topmost node and sweeping downward. As can be seen in Figure 3.9, there are cases where sweeping from the top to the bottom of a layer results in a worse solution, than if the sweep direction were the other way around.

For this reason, the algorithm could simply be changed to be run twice, once sweeping from top to bottom in a layer, and once the other way around. After counting the number crossings for both possibilities, the better version is taken.

Obviously this increases run-time. Since the algorithm is run twice, it should simply double the time needed to compute a reordering. Furthermore, we need to count crossings for the two-sided variant as well, which we did not need to do before. This would remove some of the performance benefit of the two-sided variant.

Incidentally, cases where certain combinations of top-down and bottom-up sweeps return better solutions can also be constructed. Considering these as well would increase the run-time proportionally to the amount of variants tried. For the implementation described more closely below, only the two most obvious variants of only sweeping upwards or downwards in a layer are compared.

3.3.3 When to count which crossings

The algorithm suggested by Eades and Kelly uses a *crossing matrix* to decide when to switch. A quick reminder of definitions: The free layer L_f is a tuple of nodes $(v_0, \dots, v_{|L_f|-1})$ and the crossing matrix is a $|L_f| \times |L_f|$ matrix $C = (c_{i,j})_{i,j<|L_f|}$ where each entry $c_{i,j}$ stores the number of crossings of between-layer edges incident to nodes v_i, v_j with $pos_v(v_i) < pos_v(v_j)$, i. e., v_i is placed above v_j. Therefore, switching nodes v_i and v_j would reduce crossings as soon as $c_{i,j} < c_{j,i}$. As noted by Eades and Kelly, and as we will see in more detail later, calculation of this matrix can be done in time $O(|E||V|)$ [EK86].

Any previous research using the greedy switch algorithm computes this crossing matrix. However, in many cases only few entries of this crossing matrix will be needed. This is especially true when the barycenter algorithm has been run beforehand, because only few crossings should remain. If the greedy switch algorithm cannot optimize any crossings only the entries adjacent to the diagonal of the matrix are of any interest.

The naïve way to decide when a switch of neighbouring nodes could help, is to simply count all crossings in the layer for the original situation, switch the nodes, and then recount. At first glance, in the case where no crossings can be reduced by greedy switch, this method might seem to be more efficient, since it does not calculate unnecessary options. However, this intuition can quickly be refuted when considering the run-time of the algorithm. The fastest known algorithm for counting crossings between two layers runs in $O(|E|log|E|)$ [BJM02]. In the case where greedy switch cannot reduce crossings, it will visit each node and evaluate the number of crossings for the original order and for the switched order per visited pair of nodes. Not taking the special cases of in-layer edges and north-south port edges into account, this results in a minimal running time of $\omega(|V||E|log|E|)$. In the same case, including calculation of the crossing matrix, the usual version of the algorithm has a minimal running time of $\omega(|V| + |E||V|)$.

The intuition to avoid calculating unnecessary entries in the crossing matrix is a good one however. The entries can simply be calculated on demand and saved in the crossing matrix for possible later use. Calculating between-layer crossings of edges incident to neighbouring nodes $v_1, v_2 \in V$ for both the original and the switched order runs in time $O(|E_{v_1}| + |E_{v_2}|)$, where $|E_{v_i}|$ is the number of edges incident to $v_i \in V$. Therefore, in the case of no possible switch, the minimal running time of this variant is $\omega(|E|)$. Since the barycenter algorithm will be run before the greedy switch heuristic, the assumption is that this minimal running time will be the actual running time in many cases.

The algorithms for counting edge crossings must still be discussed more closely, and the data flow graphs which KLay Layered must work with do not only have between-layer edges. The following chapter steps the abstraction down a further level and discusses the exact algorithms and running times for calculating the number of crossings for all different types of edges drawn in KLay Layered.

Chapter 4

Counting Edge Crossings

The greedy switch heuristic is a simple algorithm which is quickly explained. However, the major challenge is developing algorithms which count the number of crossings in order to know when a switch of neighbouring nodes improves the graph. In the case of KLay Layered, the situation is significantly more complex than in most of the literature about automated layout of layered graphs, because we must take into account several different types of edges, namely

▷ *Between-layer edges*: Edges which connect nodes in the neighbouring layers.

▷ *In-layer edges*: Edges which connect nodes in the same layer.

▷ *North/south edges*: By ordering dummy nodes with edges to ports on the north or south side of a node, crossings can be reduced.

▷ *Hyperedges*: Edges going to the same node or port can be joined to a hyperedge, reducing the number of edges and crossings in an effort to increase readability.

In some cases the algorithms used to count these edges will be different depending on whether we aim to only count the crossings of edges incident to two (in some cases neighbouring) nodes or if we want to calculate all crossings in the layer.

The following sections will therefore discuss all of these different problems separately. In order to simplify the algorithms and shorten the discussion, the following sections will always use *undirected* edges.

4.1 Crossings of Between-layer Edges

The first problem to be discussed here will be the issue of counting crossings between edges passing from a layer to a neighbouring layer. The literature discussing layer-based layout and edge crossing reduction mostly only considers this situation.

4.1.1 Whole Layer

As mentioned earlier, an efficient algorithm for counting all crossings of edges passing between neighbouring layers was suggested by Barth et al. [BJM02]. This

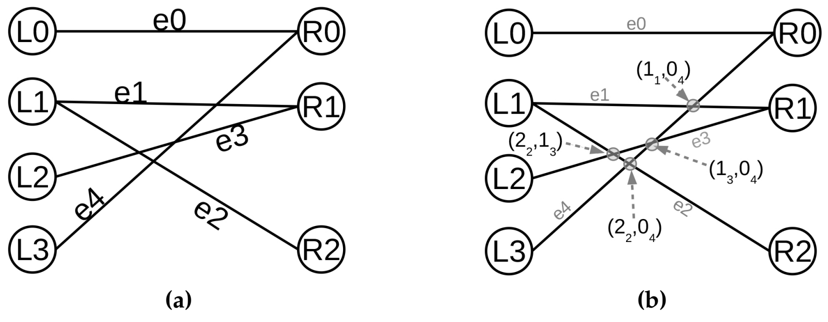


Figure 4.1. (a) shows the lexicographical sort order of the edges. (b) shows the correspondence of edge crossings to the inversions.

algorithm is implemented in KLay Layered to count the number of crossings remaining after every sweep of the barycenter heuristic. The following section summarizes the explanation from the paper of Barth et al.

To differentiate between the graph and the tree data structure that will be described below, the following explanation will use the term *vertex* instead of *node* for the graph and *node* for the tree data structure.

One definition in advance: In a sequence $\pi = (a_0, \ldots, a_{n-1}), n \in \mathbb{N}, a_i \in \mathbb{N}, i < n$, a pair (a_i, a_j) is called an *inversion* if $i < j \wedge a_i > a_j$. The *inversion number* $INV(\pi)$ is the number of inversions in a sequence. For example in $\pi = (0_0, 1_1, 2_2, 1_3, 0_4)$ with entries of the form $value_{position}$ there are 4 inversions: $(1_1, 0_4), (2_2, 1_3), (2_2, 0_4)$ and $(1_3, 0_4)$, hence $INV(\pi) = 4$.

In a two-layer graph (L_l, L_r) the between-layer edges can be sorted lexicographically in such a way so that in $\pi_E = (e_0, \ldots, e_{|E|})$ for each pair of edges $e_i, e_j \in \pi_E$, it holds that $e_i = \{l_i, r_i\} < \{l_j, r_j\} = e_j$ iff $pos_v(l_i) < pos_v(l_j)$ or $pos_v(l_i) = pos_v(l_j)$ and $pos_v(r_i) < pos_v(r_j)$. Consider for example the two-layer graph in Figure 4.1a in which edges are named in by their position in π_E.

We now take a look at the sequence of the position values of the nodes in the right layer sorted by their occurrence in π_E. Using the example of Figure 4.1 we have $\pi = (0_0, 1_1, 2_2, 1_3, 0_4)$ using the form $rightNodePosition_{edgeNumber}$. In π each inversion corresponds to an edge crossing in the graph. The inversions and their corresponding crossings are marked in Figure 4.1b.

Therefore, to count the crossings in a two-layered graph, it is sufficient to calculate $INV(\pi)$. To do this, Barth et al. use a so-called *accumulation tree*. This is a perfectly balanced binary tree with l leaves $(lf_0, \ldots, lf_{l-1})$, where l is the next power of two larger than the number of nodes in the right layer, $|L_r|$, formally: $2^{c-1} < |L_r| \leqslant 2^c = l, c \in \mathbb{N}$. Each node and leaf carries a value initialized to 0. When a leaf's value is incremented, all its ancestors' values are incremented as well. We number all leaves ascendingly corresponding to the positions of the vertices.

The algorithm of Barth et al. iterates through the vertices in the order of π incrementing the leaf with the corresponding position. Each time a left sibling's value is incremented, the value on the right sibling's node or leaf is added to the number of crossings.

For each visited vertex in L_R we must traverse the tree in $log|V|$ steps, resulting in a runtime of $|E|log|V|$. If we can choose the smaller layer of the two-layered graph as the one whose nodes are being represented in the accumulation tree, it runs in $|E|log|V_{small}|$. Whenever counting the crossings in a complete layered graph, this consideration is only relevant when the length of the layer is accessible in constant time. Even then, it might complicate the implementation without a significant performance benefit.

To implement the one-sided variant of the greedy switch algorithm, this counting method will be needed. More frequently however, we need to count the number of crossings for only two nodes, which is the focus of the following section.

4.1.2 Two Nodes

When deciding when to switch neighbouring nodes or when calculating the crossing matrix, we only need to calculate the crossings caused by between-layer edges incident to two neighbouring nodes or those corresponding to the current entry in the crossing matrix. Using the algorithm described in Section 4.1.1, and simply reducing the graph to two nodes, we already have an algorithm which calculates the number of crossings in $O(|E|log\ |V_{small}|) = O(|E_{v_{upper}} \cup E_{v_{lower}}|log\ 2) = O(|E_{v_{upper}} \cup E_{v_{lower}}|)$ where v_{upper}, v_{lower} are the two nodes currently being considered.

However, using an adaption of the suggestion in the original paper by Eades and Kelly [EK86], we can calculate both the values for the original order of nodes and the switched order in $O(|E_{v_{upper}} \cup E_{v_{lower}}|)$, giving a slightly more efficient algorithm.

To do this, we merge the adjacency lists $A_{v_{upper}}$ and $A_{v_{lower}}$ of the two nodes. Each time the adjacency with the highest position in the neighbouring layer is from $A_{v_{lower}}$, we can add to the number of crossings for the original order $c_{upperlower}$ the remaining length of $A_{v_{upper}}$ and remove the adjacency from $A_{v_{lower}}$. This is done analogously when the next adjacency comes from $A_{v_{upper}}$ for the switched order crossings $c_{lowerupper}$. If the next adjacencies in both lists have the same position value p, we add to $c_{upperlower}$ the number of remaining adjacencies in $A_{v_{upper}}$ without the number of instances where $pos_p(p') = p, p' \in A_{v_{upper}}$. We do the same for $c_{lowerupper}$, adding the number of remaining adjacencies in $A_{v_{lower}}$ without the number of instances where $pos_p(p') = p, p' \in A_{v_{lower}}$.

Consider the algorithm *mergeAdjacencies* (Algorithm 4) for an exact description and Figure 4.2 for a detailed explanation of each case the algorithm considers.

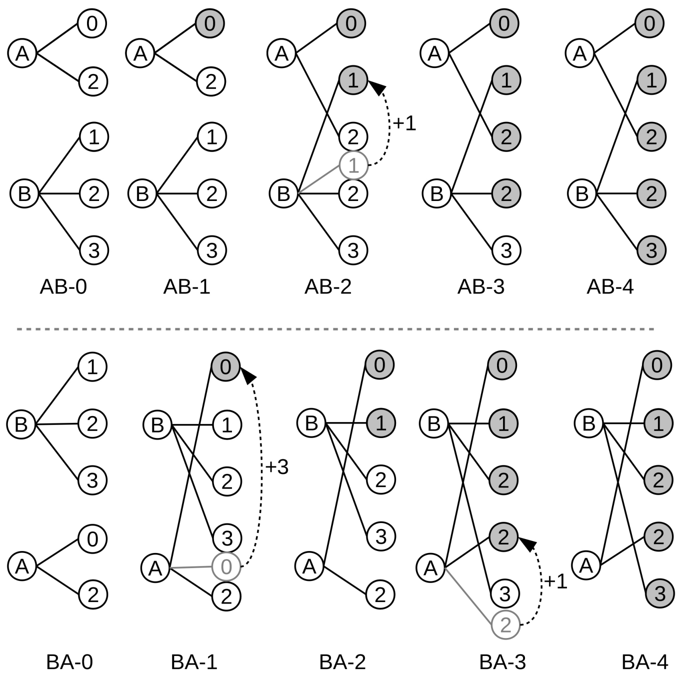


Figure 4.2. Between-layer edge crossings between two nodes A and B are calculated by merging sorted adjacency lists. The nodes filled with grey are deleted from the adjacency list by the algorithm. Both the crossings for the order A-B and for the order B-A can be calculated in the same step. There are three cases:

- ▻ BA-1: The position of the next adjacency to A is higher than the next adjacency to B. We add to the B-A crossings the number of remaining adjacencies to B.
- ▻ AB-2: The position of the next adjacency to B is higher than the next adjacency to A We add to the A-B crossings the number of remaining adjacencies to A.
- ▻ AB-3 and BA-3: The position of the next adjacency to B, in our case position 2, is the same as than the next adjacency to a We add to the B-A crossings the number of remaining adjacencies to B after the position of the next adjacency (position 2) and to the A-B crossings the number of remaining adjacencies to A after the position of the next adjacency (position 2).

Remember that edges connected to a node on a given side d are be written as $conn_d(v)$.

In the worst case, this algorithm must visit all edges of both nodes. This means that it runs in time $O(|E_{v_{upper}}| + |E_{v_{lower}}|)$. However, in KLay Layered the adjacencies of each node are not kept sorted by their target in the neighbouring layer. This means that before running the algorithm, we must sort the adjacencies, adding $|E_{v_{upper}}|\ log|E_{v_{upper}}| + |E_{v_{lower}}|\ log|E_{v_{lower}}|$. When sweeping through a layer, the sorted adjacencies can be calculated beforehand and reused. Eades and Kelly [EK86] assume that the adjacency list data structure is used for representing the

Algorithm 4: mergeAdjacencies

```
Data: Adjacency Lists $A_{v_{upper}}$ and $A_{v_{lower}}$ with $pos_v(v_{upper}) < pos_v(v_{lower})$,
      Side $d \in D$
Result: $c_{upperlower}, c_{lowerupper}$: number of crossings of edges incident to
        $v_{upper}, v_{lower}$
1  while $|A_{v_{upper}}| > 0$ and $|A_{v_{lower}}| > 0$ do
2      $p_{upper}$ := first element of $A_{v_{upper}}$
3      $p_{lower}$ := first element of $A_{v_{lower}}$
4      if $pos_p(p_{upper}) > pos_p(p_{lower})$ then
5          $c_{upperlower}$ += $|A_{v_{upper}}|$
6          remove first entry in $A_{v_{lower}}$
7      else if $pos_p(p_{lower}) > pos_p(p_{upper})$ then
8          $c_{lowerupper}$ += $|A_{v_{lower}}|$
9          remove first entry in $A_{v_{upper}}$
10     else
11         $v_{f\_upper}$ := the node to which the first entry of $A_{v_{upper}}$ is connected.
12         $c_{upperlower}$ += $|A_{v_{upper}}| - conn_d(v_{upper}, v_{f\_upper})$
13         $v_{f\_lower}$ := the node to which the first entry of $A_{v_{lower}}$ is connected.
14         $c_{lowerupper}$ += $|A_{v_{lower}}| - conn_d(v_{lower}, v_{f\_lower})$
15         remove first entry in $A_{v_{upper}}$
16         remove first entry in $A_{v_{lower}}$
```

complete graph and do not take port based graphs into account. Incidentally, the same must be done in KLay Layered for the algorithm of Barth et al.

In the case of computing the crossing matrix the runtime can be calculated as follows: When finding crossings of the first node v_0 to all other nodes, the incident between-layer edges of v_0 are visited $|V| - 1$ times. The edges of the next node v_1 then have been visited once and v_1 already has an entry for crossings with v_0. Therefore its edges must then only be visited $|V| - 2$ further times resulting in a total of $|V| - 1$ visits. This is continued equivalently for the rest of the nodes and results in the running time of $O(|E||V|)$ noted above. Since for KLay Layered we must sort the adjacency lists beforehand, we must add a factor of $\sum_{v \in L} |v| log |v|$.

4.2 Crossings of In-layer Edges

In-layer edges appear frequently in data flow graphs. Because inputs are usually drawn on the left of a node, as soon as data from an output function or group of functions is fed back recursively, the edge must wrap around the node. In this case dummy nodes are added in the same layer with an in-layer edge connecting

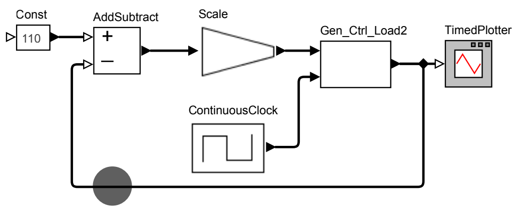


Figure 4.3. The edge going backward from Gen_Ctrl_Load2 to AddSubtract is a feedback edge, because it goes against the usual left to right data direction. The grey circle shows the position of the corresponding dummy node.

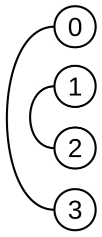


Figure 4.4. The current in-layer crossing count algorithm simply takes the difference of the port positions of each edge as the number of in-layer crossings. In this case the algorithm would count two crossings, while there are actually none.

the two. See Figure 4.3 for an example.

In-Layer edges can also cause crossings, both with between-layer edges and with other in-layer edges. In order to know when to switch neighbouring nodes and to prevent the one-sided greedy switch variant of adding extra edge crossings, the number of in-layer crossings need to be counted.

Since the barycenter algorithm also needs to know the current number of crossings after each sweep, KLay Layered uses an algorithm for counting all in-layer crossings in a layer. This algorithm is very simple and runs in linear time, however it only gives an upper bound for the in-layer crossings.

In order to calculate the upper bound it iterates through all ports, marking all those which are unvisited. For each visited port with an in-layer edge it simply adds the amount of port positions between both ends of the edge to the number of crossings. This however, will often lead to a number of crossings which is too high. As soon as there are in-layer edges which start and end between the ends of another in-layer edge, the algorithm will count all their ports as crossings, even though no crossings exist. See Figure 4.4 for an example.

For the greedy switch algorithm, the exact number of crossings must be known. The following section describes a simple algorithm for counting all in-layer crossings in a layer. The same algorithm is reused for counting crossings between neighbouring nodes.

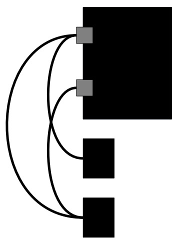


Figure 4.5. If the port order of the highest node is not set as fixed, the in-layer edge counting algorithm will set the position functions of all ports on the top node to the same value. Therefore the crossing will not be counted, since the assumption is that the port sorting algorithm can resolve the crossing. In this case, the assumption is wrong.

4.2.1 Whole Layer

Counting in-layer edge crossings is complicated by the fact that we do not want to count crossings of in-layer edges incident nodes whose port order is not fixed. We assume that all these crossings can be removed by sorting the port order, and leave the problem to a port sorting algorithm running after the greedy switch heuristic. This means that we cannot simply use the order of the edge end ports to count the edge crossings. Note that the assumption that the port sorting algorithm can remove all of the crossings between edges incident to a node with free port order is only an approximation. Figure 4.5 shows a hypothetical example where this is not the case. When using this assumption, the greedy switch heuristic will not return the best possible solution for these cases.

In order to count the number of in-layer crossings we proceed through all edges in the graph, saving the start and end position of each edge in a sorted list. If the edge is a between-layer edge, we simply add the position of the port on this layer to the list. Each time we meet an edge which has been visited, we count all port position in between the end and start position of this edge and delete it from the list. As defined in Definition 5, ports on nodes with free port order have equal position values, so in this way, crossings on the same node with free port order are ignored. The algorithm is run twice, once for the edges on the eastern side of the layer in question and once for those on the western side. See Figure 4.6 for a graphical explanation of the algorithm.

Since we will reuse the actual counting algorithm with different ports for counting the in-layer edge crossing on only two nodes, the algorithm is split into two parts, *allInLayer* (Algorithm 5)) and *countInLayerCrossings* (Algorithm 6)). As a reminder, $P_d, d \in D$ is a list of ports on side d (in this case east or west) ordered according to their position in the layer and $conn(p)$ is the set of edges incident to port p.

How fast is this method? The implementation uses a binary tree datastructure with the ability to specify a cardinality for its leaves. It is used to store the port

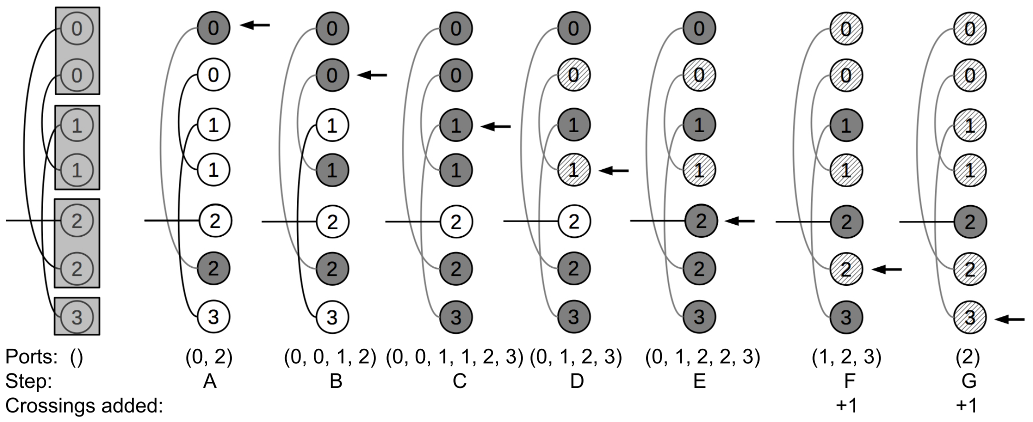


Figure 4.6. Counting in-layer edge crossings: The left-most situation shows the nodes on which the ports are situated: The nodes are drawn as a rectangle, while the circles are the ports with their position values. In this case the port order on all nodes is not fixed, resulting in only two counted crossings, because the port list sorting algorithm can remove these crossings by reordering the ports with same position values. The port positions which have been entered into the sorted list of port positions are marked in grey. The port positions where both ends of an edge have been visited are shown hatched. The arrow shows the port currently being visited. Note that in step `D` no crossing is counted, because the ports in the third and fourth step have the same position as they are on the same node without fixed port order. Similarly in step `F`, the crossing between the edge which is deleted from the list and the between-layer edge above it is not counted, because the position value is the same.

Algorithm 5: *allInLayer*. Remember that $P_d, d \in D$ is a list of ports on side s

```
   Data: Layer $L = (v_0, \dots, v_n - 1), n \in \mathbb{N}$, Side $d \in D$
   Result: number of crossings $c$ caused by in-layer edges
1  $E_{visited}$ := empty set of edges
2  $T_{P-visited}$ := binary tree structure, with port positions at its leaves
3  for $p$ in $P_d$ do
4      $c$ += countInLayerCrossings($p$, $E_{visited}$, $T_{P-visited}$)
```

positions of each edge. With $|E_{IL}|$ as the number of in-layer edges and $|E_{BL}|$ as the number of between-layer edges, this means that finding the first leaf $i \in T_{P-visited}$ with $i > p_{curr}$ runs in $O(log(2 \cdot |E_{IL}| + |E_{BL}|))$. Calculating the length of the subset $|\{\text{leaves } i \in T_{P-visited} \mid i > pos_p(p_{curr})) \wedge i < pos_p(p'_{curr}))\}|$ then needs linear time, in the worst case $O(2 \cdot |E_{IL}| + |E_{BL}|)$. Therefore *allInLayer* runs in $O(|E|(2 \cdot |E_{IL}| + |E_{BL})log(2 \cdot |E_{IL}| + |E_{BL}|))$.

Since the algorithm is run as a post-processor step to the barycenter algorithm, the actual distance between the ends of an in-layer edge should never be very large however, which should keep the linear factor $2 \cdot |E_{IL}| + |E_{BL}|$ very small, reducing the quadratic nature of the algorithm. Remember that due to the adaption of the

Algorithm 6: *countInLayerCrossings*

```
Data: Port $p$, Set of visited edges $E_{visited}$, Binary tree structure with port
      positions at its leaves $T_{P-visited}$
Result: Returns number of crossings $c$ caused by in-layer edges incident to
        $p$ with edges in $E_{visited}$ and updates set $E_{visited}$ and binary tree
        $T_{P-visited}$
1  for $e = \{p_{curr}, p'_{curr}\}$ in $conn(p)$ do
2      if $pv(p_{curr}) = (v, d) : v \in L$ then
3          if $e = \{p_1, p_2\} \notin E_{visited}$ then
4              $E_{visited} := E_{visited} \cup e$
5              $T_{P-visited}$ := add to tree $pos_p(p_1)$ and $pos_p(p_2)$
6          else
7              $E_{visited} := E_{visited} \setminus e$
8              $c$ += $|\{\text{leaves } i \in T_{P-visited} \mid i > pos_p(p_{curr})) \wedge i < pos_p(p'_{curr}))\}|$
9      else
10         $c$ += $|E_{visited}| - |\{\text{leaves } i \in T_{P-visited} \mid i = pos_p(p_{curr}\}|$
```

barycenter algorithm suggested by Schulze [Sch11] and implemented in KLay Layered, during the calculation of the barycenter of a node n with in-layer edges, the barycenter value of the connected in-layer node is also used as part of the calculation (see Section 2.3). This is done with the explicit goal of avoiding the end ports of in-layer edges being positioned too far apart.

In order to be used for greedy switch, only the crossing number for two neighbouring nodes is needed, which is discussed in the next section.

4.2.2 Between two Nodes

Now that we have an exact algorithm for counting in-layer crossings for all edges in the layer, the simplest way of implementing an algorithm which counts crossings for edges incident to two nodes is to reuse the algorithm given above. We simply pick only those ports P_{rel} connected to the two nodes and feed them to *countInLayercrossings*. See the algorithm *twoInLayer* (Algorithm 7).

This method counts all in-layer crossings, including those of in-layer edge crossings of edges incident to only one of the two nodes which exist due to fixed port-ordering. These crossings cannot be changed by exchanging the nodes and it is therefore unnecessary to count them. The correctness of the decision whether to switch nodes or not is not influenced however, since the number of crossings caused by fixed port order will be the same in both the original and the switched order.

To count the number of crossings for the switched order, the nodes must

```
Algorithm 7: twoInLayer
    Data: Nodes $v_1, v_2$
    Result: Crossing number $c$ caused by in-layer edges connected to $v_1$ and $v_2$
1   $P_{rel} := (p_0, \ldots, p_{k-1}), \{p_i, p'\} \in E, vp(p_i) = v_1 \vee vp(p_i) = v_2, pos_p(p_i) < pos_p(p_{i+1}), i < k-1$
2   $E_{visited}$ := empty set of edges
3   $T_{P-visited}$ := binary tree structure, with port positions at its leaves
4   for $p$ in $P_{rel}$ do
5       $c$ += countInLayercrossings($p$, $E_{visited}$, $T_{P-visited}$)
```

actually be *switched* and the crossings recounted, updating the values of the pos_p function as defined in Definition 4.

The running time of this algorithm is the same as *allInLayer*, however the number of considered edges is obviously reduced.

The original algorithm in KLay Layered counted the number of crossings between north/south edges and long edge dummies together with the in-layer edge crossings. Since this is now not the case for the algorithm described in this section those crossings will be counted together with the problem for the following section: counting crossings of north/south edges.

4.3 Crossings of North/South Edges

4.3.1 Whole layer

As described in Section 3.3, only the one-sided variant can switch the order of north/south dummy nodes. The two-sided variant could however be able to switch normal nodes and long-edge dummies where the north/south edges create unnecessary crossings.

The algorithm for counting north/south port crossings currently implemented in KLay Layered is very simple and runs in quadratic time. It traverses the layer and for each north/south port dummy iterates through all following nodes until it finds the normal node to which it is connected. On the way, it adds to the crossing number all north/south port edges it crosses, using a similar table as described for the algorithm for counting north/south edge crossing between two nodes described in Section 4.3.2.

The following section will describe a simple algorithm for counting all north-south port crossings which runs in linear time. Consider the image in Figure 4.7. Only the crossings of the horizontal edges are counted so as not to count the crossings twice. Let us first consider the node at position $0, 2$. To simplify, we can

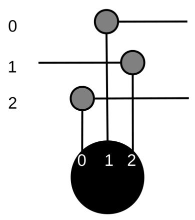


Figure 4.7. The row of numbers in the node show the position of the ports. The numbers on the side show the *nearness* of the north/south dummies to their origin port, i. e., the closer the node, the higher the nearness.

draw the situation as a matrix:

$$\left[\begin{array}{ccc} \cline{2-3} - & \multicolumn{1}{|c}{x} & \multicolumn{1}{c|}{-} \\ - & \multicolumn{1}{|c}{-} & \multicolumn{1}{c|}{x} \\ \cline{2-3} \mathbf{X} & - & - \end{array}\right]$$

The box shows the matrix to the top right of the node in the bottom left position written in bold font, which corresponds to the node at position $0, 2$. Comparing Figure 4.7 and the matrix, we can easily see that the number of crossings of a north/south dummy with an eastern edge is equal to the number of nodes in the marked area of the matrix.

For western edges on dummy nodes, consider the box in the following matrix for the node in position $2, 1$:

$$\left[\begin{array}{ccc} - & x & - \\ - & - & \mathbf{X} \\ \cline{1-2} \multicolumn{1}{|c}{x} & \multicolumn{1}{c|}{-} & - \\ \cline{1-2} \end{array}\right]$$

In this case the number of crossings is equal to the number of nodes in the remaining matrix to the bottom left of the node in question.

As defined in Definition 3, $|conn_d(v)|$ is the number of edges incident to node v on side $d \in D$. Let $V_{NS} = \bigcup_{v \in V, d \in \{s,n\}} conn_d(v)$ be the set of north/south port dummies. Furthermore, let $originNode : V_{NS} \rightarrow V$ be a function mapping a north/south port dummy to its connected normal node in the same layer and $originPort : V_{NS} \rightarrow V$ be the port the north/south port dummy is connected on for that node. In order to be able to find the needed *coordinates* for north/south port dummies, the *nearness* of a north/south dummy to its origin is defined as follows:

$$nearness := (V_{NS}, D) \rightarrow \mathbb{N}, nearness(v_{NS}, d) = |conn_d(originNode(v_{NS}))| - \\ |pos_v(v_{NS}) - pos_v(originNode(v_{NS}))|$$

For a north/south dummy $v_{NS} \in V_{NS}$ with $pos_p(originPort(v_{NS}))$ we now have the same coordinates as in Figure 4.7. Since in each row and column of the matrix, there can only be one node, the number of crossings for a north/south dummy

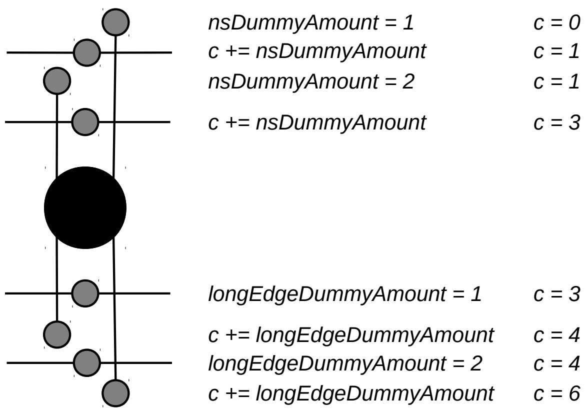

nsDummyAmount = 1 c = 0
c += nsDummyAmount c = 1
nsDummyAmount = 2 c = 1
c += nsDummyAmount c = 3
longEdgeDummyAmount = 1 c = 3
c += longEdgeDummyAmount c = 4
longEdgeDummyAmount = 2 c = 4
c += longEdgeDummyAmount c = 6

Figure 4.8. Step by step example for counting all crossings between long edge dummies and north/south nodes.

v_{NS} with an western edge can be calculated as:

$$min(pos_p(originPort(v)), nearness(v))$$

the number of crossings for a north/south dummy v_{NS} with an eastern edge on the northern side can be calculated as:

$$min(card_s(v_{NS}) - pos_p(originPort(v_{NS})) - 1, nearness(v_{NS}))$$

For north/south dummies on the southern side of a node, we can use the exact same algorithm, if the pos_p function also numbers the ports from east to west in ascending order.

We can now choose to iterate either over the north/south ports of each normal node, or over all north/south dummies of a node and add the resulting crossings. This algorithm obviously runs in linear time.

In KLay Layered, long-edge dummies can be drawn in between north/south dummies and their origin node. Obviously this causes crossings with all north-south edges passing through the long edge dummy. Originally, these crossings were counted at the same time as the in-layer edges. Since this algorithm has been replaced, they need to be counted. Counting these crossings is simple and can be done in a single pass across the layer. Each time we are on the north side of the origin node of north/south dummies, we collect the current number of north/south dummies which already have been visited and each time we meet a long edge dummy we add to the crossing count the current number of north/south dummies. On the southern side, we count the number of long edge dummies we meet and each time we meet a north/south dummy, we add to the crossing count the current number of long edge dummies. Consider the example Figure 4.8.

Once again we need a method for counting north/south edges for only two

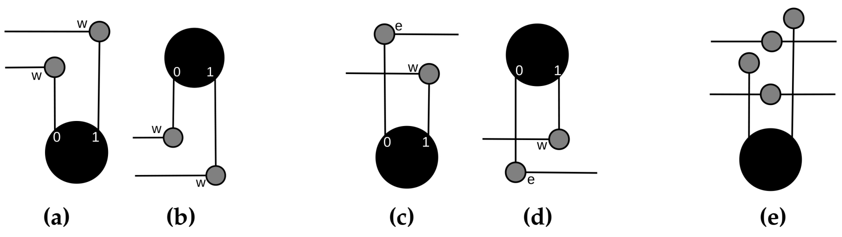


Figure 4.9. Two of the possible cases for north/south port crossings. The numbers on the large black origin nodes show the port positions of the north/south edges and the letters at the edges of the north/south dummies show the direction of the edges (e for east and w for west). The image in (b) shows the mirrored situation for southern ports as for the northern ports in (a). The same holds for (c) and (d). (e) shows crossings between north/south dummies and long-edge dummies.

Edge direction $v_{further}$	Edge direction v_{closer}	Port positions	Crossings
east	east	$pos_{further} < pos_{closer}$	c += 0
east	east	$pos_{further} > pos_{closer}$	c += 1
east	west	$pos_{further} < pos_{closer}$	c += 1
east	west	$pos_{further} > pos_{closer}$	c += 0
west	west	$pos_{further} < pos_{closer}$	c += 1
west	west	$pos_{further} > pos_{closer}$	c += 0
west	east	$pos_{further} > pos_{closer}$	c += 1
west	east	$pos_{further} < pos_{closer}$	c += 0

Table 4.1. All possibilities for north/south port crossings. All upper and lower nodes are called $v_{further}$ and v_{closer} depending on which north/south dummy is closer to the origin node. Furthermore, for this case let $pos_{further} := pos_p(originPort(v_{upper}))$ and $pos_{closer} := pos_p(originPort(v_{lower}))$ and c be the number of crossings.

neighbouring nodes, a simple problem which is addressed in the next section.

4.3.2 Two nodes

Counting north/south crossings between two neighbouring nodes can be done by considering all possible cases for two nodes. Except for when considering long-edge dummies and normal nodes with north/south ports, there is only one crossing possible between different north/south port dummies or north/south port dummies and long edge dummies. See Figure 4.9 for an overview of some of the cases. Table 4.1 shows all possible different cases which must be taken into account when considering two north/south dummies on the northern side of their origin node.

If the port positions of the southern ports are numbered ascendingly from

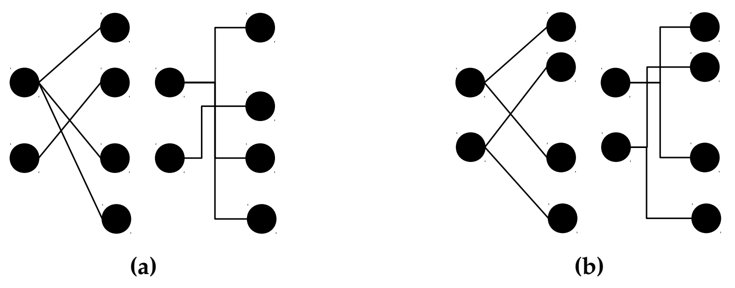


Figure 4.10. The example in (a) shows a reduction of crossings by drawing hyperedges. In (c) the number of crossings is increased. Note that by changing the node position in the eastern layer of (b), one edge crossing could be avoided. Example taken and adapted from [SSRvH14].

west to east in the same way as the northern ports, we can use the same table to look up the number of crossings. In this case the node closer to the origin node will be the upper node and the node further from the origin node will be the lower node.

As can be seen in Figure 4.9e, a long-edge dummy only causes a crossing with a north/south dummy if it is further away from the origin node of the north/south dummy then the north/south dummy itself.

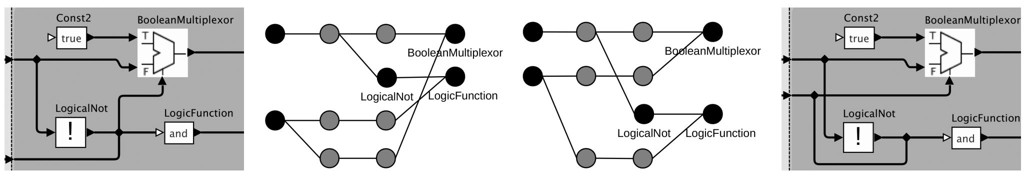
Lastly, a long-edge dummy causes crossings with all outgoing northern edges of a node if it is on its northern side and with all southern edges of a node if it is on its southern side. Since in KLay Layered the number of edges for a certain side of a node is not saved in the data structure, it must be recalculated each time, leading to slight increase in computation time.

Up to this point it has been assumed, that the number of crossings only depends on the ordering of the nodes in the layer. This is not the case with orthogonal hyperedges however, a problem which is discussed in the following section.

4.4 Crossings of Hyperedges

KLay Layered supports the drawing of orthogonal hyperedges. An example can be seen in Figure 4.10.

Figure 4.10 also shows the basic problem for the greedy switch algorithm. The number of crossings caused by hyperedges can be less or greater than the number of crossings would be if the same graph had been drawn with straight lines. Contrary to normal edges, in the case of hyperedges the exact number of crossings in a graph does not only depend on the order of the nodes in the layer, but also on the way the edges are routed and on the placement of the nodes. An example can be seen in Figure 4.10b. In the crossing reduction phase of the



(a) **(b)** **(c)** **(d)**

Figure 4.11. (a) shows an original drawing. (b) shows the actual data available at the node ordering phase. Long edge dummy nodes are drawn in grey. Note that there are actually two edges and that the two crossings which would actually be present if all edges were drawn straight, disappear when drawn as hyperedges. The greedy switch heuristic does not know of any hyperedges and therefore switches to the order seen in (c), reducing the straight-line crossings by one. When this order is drawn with hyperedges however, as seen in (d) it results in one extra crossing when compared with (a).

framework by Sugiyama et al., the routing of the edges is unknown.

The obvious problem in our case is that although a switch might reduce the number of straight line crossings, the actual number of crossings might be increased. A real-world example of this is shown in Figure 4.11.

Spönemann et al. suggest algorithms for approximating the number of hyperedge crossings during the crossing reduction phase. These algorithms can then be used by the barycenter algorithm to decide when the number of between-layer edge crossings has been improved.

A similar possibility exists in our case. By simply replacing the between-layer crossing algorithm discussed in Section 4.1 for counting crossings in the whole layer with the algorithm by Spönemann et al. when layouting graphs with orthogonal hyperedges, the number of crossings might be further reduced. Since the algorithm *ApproxOpt* described by Spönemann et al. has been implemented in KLay Layered, we can reuse it for this case. It runs in $O(b + |H|(log|V| + log|H|)$, where H are the hyperedges, and $b = \sum_{(S,T)\in H}(|S| + |T|)$.

However, tt is not possible to use the algorithm to decide when to switch nodes without slowing down the algorithm significantly. Researching whether there is a possibility to estimate the change of crossings for hyperedges when only taking two neighbouring nodes into account is be beyond the scope of this thesis and is material for further research.

In this chapter several algorithms counting the crossings caused by different types of edges have been introduced and are in place for use in the implementation. The following chapter gives a short overview of KLay Layered and in what way the greedy heuristic and all the crossing counting algorithms have been integrated into it.

Chapter 5

Integration into KLay Layered

5.1 KLay Layered

Although KLay Layered principally follows the framework suggested by Sugiyama et al., it must be able to deal with significantly more complex situations and types of graphs than described in the original paper. The first and most obvious difference is that we cannot assume that all graphs which are given to the algorithm are acyclic. This means that KLay Layered must add another phase at the beginning of the algorithm in order to remove the cycles. Another issue dealt with by KLay Layered in a separate phase is edge routing, resulting in a total number of five phases.

The graphs processed by KLay Layered have many other use cases which must be dealt with. They are port-based graphs and contain labels, hyperedges and comments, to name just a few of the extra cases to be considered during the layout. None of these situations are covered by the Sugiyama algorithm. In order to keep complexity under control and to have sufficient flexibility in the implementation, all cases the five main phases were not specifically designed for are kept separate in so-called *Intermediate Processors* [Sch11], which are executed in between the phases. A simple overview is shown in Figure 5.1.

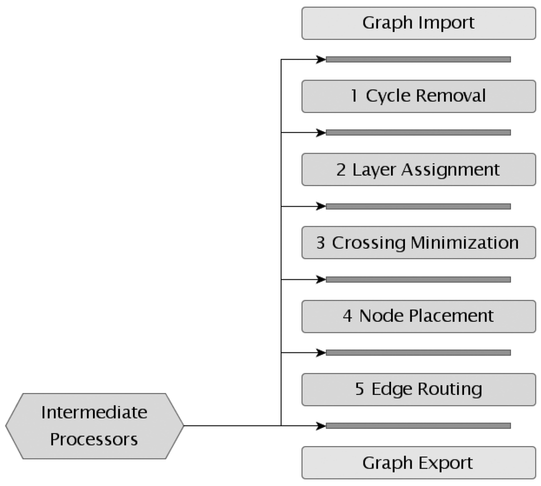


Figure 5.1. An overview of KLay Layered's architecture.[1]

In order for the algorithm to know when to execute the intermediate processors, each phase must specify their dependencies. This defines in between which phases which intermediate processor must be run. The execution order of processors which are in between the same phases must be manually defined by the programmer.

This modular structure of the algorithm enhances the freedom to adapt and extend the algorithm greatly and simplifies maintainability.

5.2 Greedy Switch Post-processor

Since we are using the greedy switch heuristic as a post-processor, the obvious place to implement it is as an intermediate processor behind the crossing minimization phase.

As described in the previous chapters, there are several different variants of the greedy switch heuristic to be compared. To summarize:

1. When to count the crossings:
   (a) Recounting all crossings for each switch
   (b) Calculating the crossing matrix
   (c) Calculating the crossing matrix on demand
2. Which crossings to count:
   (a) One-sided approach
   (b) Two-sided approach
3. Comparing results for either downward or upward in-layer sweep direction or not
4. Using the hyperedge crossing counter or not

This results in conceivably 24 different combinations. While it is certainly not interesting to actually compare all 24, for the experimental evaluation the implementation should give the flexibility to be able to choose any variant easily.

In general the algorithm was split up in three different parts: The intermediate processor GreedySwitchProcessor, the SwitchDecider classes and the CrossingsCounter classes. The configuration of the algorithm is defined by the GreedySwitchType enum. The GreedySwitchProcessor implements the most abstract view of the algorithm, sweeping across the graph and the layers. It delegates the decision whether to switch neighbouring nodes to its

[1]Taken from `http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KLay+Layered`, accessed 03/16/2015

SwitchDecider. Both the GreedySwitchProcessor and the SwitchDecider use the CrossingsCounter classes to calculate the crossings they need.

Let us first examine the implementation of the crossing counting algorithms. As described in Chapter 4, we have a different algorithm for each different type of edge and for each of these a different one for the case of counting crossings for neighbouring nodes and for the case of counting crossings in the complete graph. For these reasons, each crossing count algorithm is implemented in a separate class. Except for the current node norder in the layers in question, they share no data between each other, saving the positions of ports and nodes on instantiation. This structure maximizes cohesion in each class while keeping coupling at a minimum and thereby simplifies testability and maintainability. In this way it is simple to replace the between-layer crossing counter for counting straight edges with the hyperedge crossings counter or change the values for the pos_p or pos_v function mapping ports to position values as needed by each algorithm. As always when increasing modularization however, performance suffers. In our case, this is because first of all for each class the values of the pos_p and pos_v function are recalculated and saved separately. Secondly each algorithm iterates over the nodes, ports or edges it needs separately. Therefore the number of iterations over the graph, layer or nodes in question increases linearly with the number of classes called.

The following list shows the different crossing counter classes.

BetweenLayerStraightEdgeAllCrossingsCounter Counts all crossings of straight between-layer edges between two layers.

BetweenLayerHyperedgeAllCrossingsCounter Counts all crossings of between-layer hyperedges between two layers.

BetweenLayerEdgeTwoNodeCrossingsCounter Counts crossings for straight between-layer edges incident to two nodes.

InLayerEdgeAllCrossingsCounter Counts all crossings with in-layer edges in a specified layer.

InLayerEdgeTwoNodeCrossingsCounter Counts crossings with in-layer edges incident to two nodes.

NorthSouthEdgeAllCrossingsCounter Counts all crossings in a given layer caused by the order of north/south ports and between north/south edges and long edge dummies.

NorthSouthEdgeNeighbouringNodeCrossingsCounter Counts crossings between neighbouring nodes caused by the order of north/south ports and between north/south edges and long edge dummies.

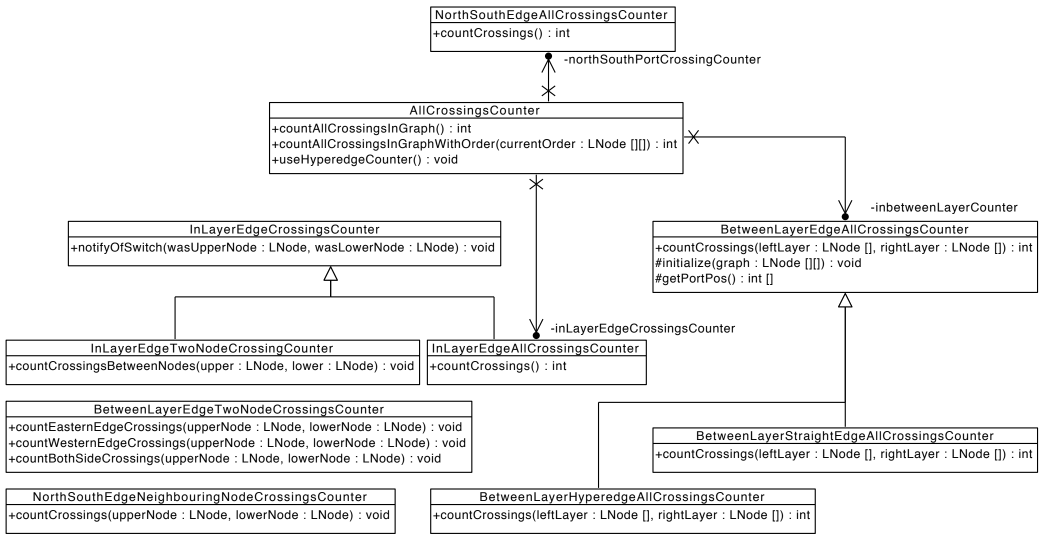

NorthSouthEdgeAllCrossingsCounter
+countCrossings() : int
-northSouthPortCrossingCounter
AllCrossingsCounter
+countAllCrossingsInGraph() : int
+countAllCrossingsInGraphWithOrder(currentOrder : LNode [][]) : int
+useHyperedgeCounter() : void
-inbetweenLayerCounter
InLayerEdgeCrossingsCounter
+notifyOfSwitch(wasUpperNode : LNode, wasLowerNode : LNode) : void
BetweenLayerEdgeAllCrossingsCounter
+countCrossings(leftLayer : LNode [], rightLayer : LNode []) : int
#initialize(graph : LNode [][]) : void
#getPortPos() : int []
-inLayerEdgeCrossingsCounter
InLayerEdgeTwoNodeCrossingCounter
+countCrossingsBetweenNodes(upper : LNode, lower : LNode) : void
InLayerEdgeAllCrossingsCounter
+countCrossings() : int
BetweenLayerEdgeTwoNodeCrossingsCounter
+countEasternEdgeCrossings(upperNode : LNode, lowerNode : LNode) : void
+countWesternEdgeCrossings(upperNode : LNode, lowerNode : LNode) : void
+countBothSideCrossings(upperNode : LNode, lowerNode : LNode) : void
BetweenLayerStraightEdgeAllCrossingsCounter
+countCrossings(leftLayer : LNode [], rightLayer : LNode []) : int
NorthSouthEdgeNeighbouringNodeCrossingsCounter
+countCrossings(upperNode : LNode, lowerNode : LNode) : void
BetweenLayerHyperedgeAllCrossingsCounter
+countCrossings(leftLayer : LNode [], rightLayer : LNode []) : int

Figure 5.2. Overview of the CrossingsCounter classes. Private methods, private members and constructors are not displayed.

AllCrossingsCounter Utility class which collects all other counters and offers methods for counting all crossings in-between two specified layers or in the whole graph.

The classes dealing with in-between layer edges and in-layer edges each have superclasses to avoid code duplication. See Figure 5.2 for an overview as a UML class diagram showing the hierarchies.

The number of in-layer crossings between any two nodes depends not only on the position of these nodes, but also on the position of all other nodes in the same layer connected with in-layer edges. Therefore, InLayerEdgeTwoNodeCrossingsCounter and InLayerEdgeAllCrossingsCounter need to be notified of a switch of nodes, so they can keep the position values up to date. This is done by calling a *notifyOfSwitch(LNode wasUpper, LNode wasLower)* method with the nodes in question as parameter. Due to the nature of the algorithms, this does not need to be done with NorthSouthEdgeAllCrossingsCounter or NorthSouthEdgeNeighbouringNodeCrossingsCounter.

The classes described so far deal with *how* to count crossings. Now let us step further up in the level of abstraction, and consider the decision *when* to count *which* crossings. The previous chapters have suggested six possibilities. First of all, we can differentiate between the one-sided and the two-sided approaches and secondly we can implement whether to recount all crossings, use a crossing matrix or calculate it on demand. Each of these will then use the CrossingsCounter classes needed for the occasion in order to decide whether a switch of two neighbouring nodes would improve the crossing number or not. These variants are implemented as subclasses of an abstract SwitchDecider class. To avoid code duplication a comparatively complex class hierarchy is used, a UML class diagram of which can be seen in Figure 5.3. The abstract SwitchDecider deals with constraints to the order of nodes. This includes preventing normal nodes being in between nodes with north/south ports and their respective north/south port dummies and vertex successor constraints, restricting the relative order of pairs of nodes. The following list describes the role of SwitchDecider class in detail:

SwitchDecider Abstract superclass. Checks constraints.

CrossingMatrixSwitchDecider Abstract class calculates entries in the crossing matrix.

CrossingMatrixOneSidedSwitchDecider Asks for all entries in the crossing matrix for crossings to one side of the free layer.

CrossingMatrixTwoSidedSwitchDecider Asks for all entries in the crossing matrix for crossings to both sides of the free layer.

OnDemandCrossingMatrixSwitchDecider Abstract class manages boolean matrix to save filled entries in the crossing matrix.

OnDemandCrossingMatrixOneSidedSwitchDecider Asks for needed entries in the crossing matrix for crossings to one side of the free layer.

OnDemandCrossingMatrixTwoSidedSwitchDecider Asks for needed in the crossing matrix for crossings to both sides of the free layer.

CounterSwitchDecider Abstract class for recounting all crossings for each possible switch.

CounterOneSidedSwitchDecider Recounts all crossings for each possible switch on one side of the free layer.

CounterTwoSidedSwitchDecider Recounts all crossings for each possible switch on both sides of the free layer.

The *notifyOfSwitch* method tells the switch decider when a switch of two nodes has actually taken place. It is not assumed, that each time *doesSwitchReduceCrossings* returns true, the order is actually switched. In order to separate logic from object creation, a SwitchDeciderFactory is used, which, depending on the type given to it in the form of a GreedySwitchType enumeration returns a newly created SwitchDecider of the type needed.

The third part of the implementation is the actual intermediate processor GreedySwitchProcessor. This class implements the most abstract form of the

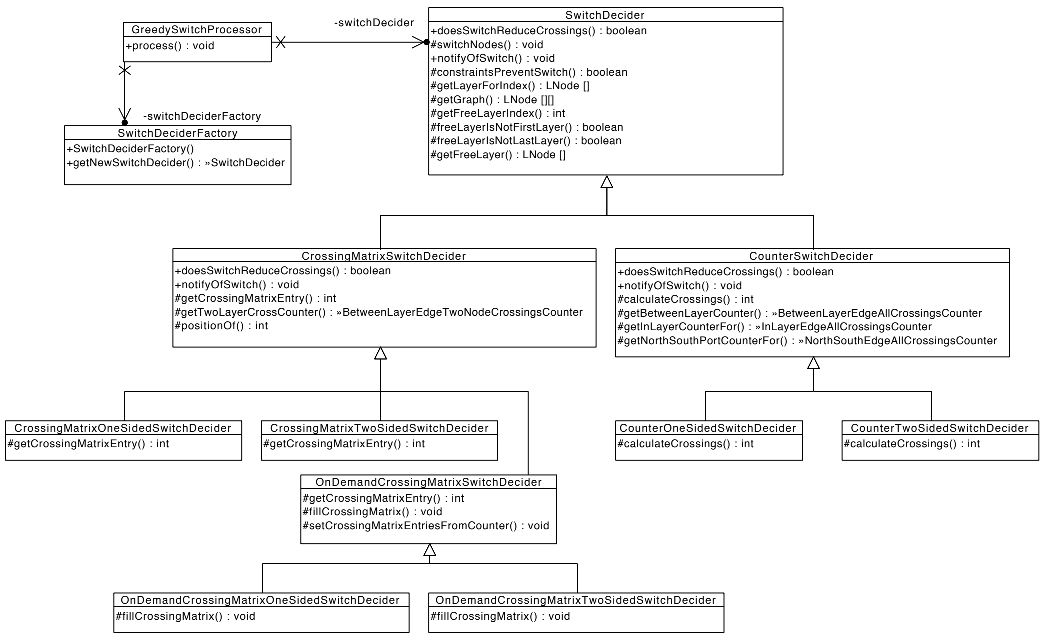

GreedySwitchProcessor
+process() : void
-switchDecider
SwitchDecider
+doesSwitchReduceCrossings() : boolean
#switchNodes() : void
+notifyOfSwitch() : void
#constraintsPreventSwitch() : boolean
#getLayerForIndex() : LNode []
#getGraph() : LNode [][]
#getFreeLayerIndex() : int
#freeLayerIsNotFirstLayer() : boolean
#freeLayerIsNotLastLayer() : boolean
#getFreeLayer() : LNode []
-switchDeciderFactory
SwitchDeciderFactory
+SwitchDeciderFactory()
+getNewSwitchDecider() : »SwitchDecider
CrossingMatrixSwitchDecider
+doesSwitchReduceCrossings() : boolean
+notifyOfSwitch() : void
#getCrossingMatrixEntry() : int
#getTwoLayerCrossCounter() : »BetweenLayerEdgeTwoNodeCrossingsCounter
#positionOf() : int
CounterSwitchDecider
+doesSwitchReduceCrossings() : boolean
+notifyOfSwitch() : void
#calculateCrossings() : int
#getBetweenLayerCounter() : »BetweenLayerEdgeAllCrossingsCounter
#getInLayerCounterFor() : »InLayerEdgeAllCrossingsCounter
#getNorthSouthPortCounterFor() : »NorthSouthEdgeAllCrossingsCounter
CrossingMatrixOneSidedSwitchDecider
#getCrossingMatrixEntry() : int
CrossingMatrixTwoSidedSwitchDecider
#getCrossingMatrixEntry() : int
CounterOneSidedSwitchDecider
#calculateCrossings() : int
CounterTwoSidedSwitchDecider
#calculateCrossings() : int
OnDemandCrossingMatrixSwitchDecider
#getCrossingMatrixEntry() : int
#fillCrossingMatrix() : void
#setCrossingMatrixEntriesFromCounter() : void
OnDemandCrossingMatrixOneSidedSwitchDecider
#fillCrossingMatrix() : void
OnDemandCrossingMatrixTwoSidedSwitchDecider
#fillCrossingMatrix() : void

Figure 5.3. Overview of the SwitchDecider hierarchy. Private methods, private members and constructors are not displayed.

algorithm, delegating the question whether or not to switch neighbouring nodes to the SwitchDecider class. For the one-sided variants it must save the current node order and compare the number of crossings after each forward and backward sweep. For this it is possible to set whether the processor should use either BetweenLayerStraightEdgeAllCrossingsCounter or BetweenLayerHyperedgeAllCrossingsCounter using the GreedySwitchType enumeration. Using the same enumeration, it is possible to specify whether the greedy switch algorithm should attempt to sweep only in one direction in a single layer or attempt both directions and use the solution which gives better results.

Integrating the greedy switch processor permanently into KLay Layered would make simplifications necessary. Since only the fastest version of functionally similar code is needed, many of the SwitchDecider classes can be removed. Finding which combination of settings would be best for production code depends on the experimental evaluation shown in the following chapter.

Chapter 6

Experimental Evaluation

The following sections show the results of the experimental evaluation of all different variants of the greedy switch heuristic discussed above. The experiments were executed on random graphs with different characteristics and on a set of 194 graphs taken from a set of demo models shipping with the Ptolemy tool[1]. These were chosen with the sole criterion that the graphs must have at least one hyperedge crossing. Ptolemy graphs use so-called *compound nodes* which the user can open to reveal a graph within a lower hierarchy level. This allows modularization and reduces visible edges and nodes in which the user is not interested in. Since these graphs would be relatively small, this structure was flattened for the purposes of this evaluation. As is usual for the layout of Ptolemy diagrams with KLay Layered, their edges were drawn as orthogonal hyperedges.

The first results discussed below are those pertaining to the speed of the algorithm, while the section after that will show the quality of the solutions.

6.1 Speed

Since KLay Layered is implemented in Java, testing execution speed is difficult. To counterbalance the effect of the Java Hotspot Compiler, a warm-up phase was executed before running the tests on the random graphs. All tests were run on relatively old hardware with a 2.1 Ghz IntelCore 2 Duo Processor and 4 GB of RAM.

As mentioned in Section 3.3.3, three different types of the greedy switch heuristic quickly come to mind when considering how to implement it. The simplest and most obvious is to recount all crossings for each considered switch. Since counting all crossings in a layer must be implemented anyway and is needed for the barycenter algorithm, this is very easily implemented. The second approach is to use the method typically found in previous research, which for between-layer edges precomputes a crossing matrix and then uses it to look up the crossing numbers when they are needed. The third approach fills this crossing matrix on demand. As mentioned, counting in-layer and north/south edge crossings must be repeated each time, because switching the order of nodes can influence these types of crossings for several different nodes. The theoretical

[1]See `Ptolemy.eecs.berkeley.edu`, accessed 03/28/2015

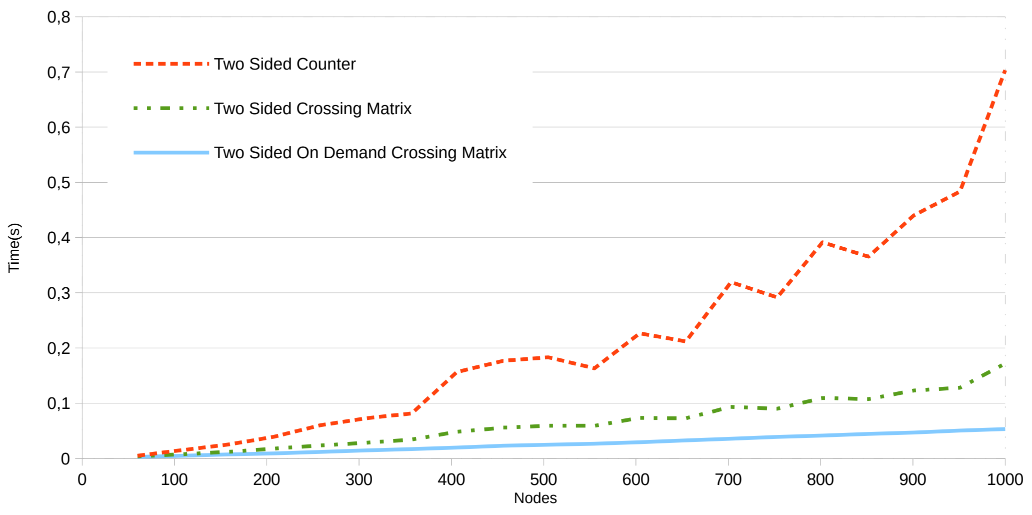


Figure 6.1. Running time for random graphs of increasing size with an average of 1.5 edges per node.

run-time analysis in Section 3.3.3 suggests that recounting all crossings each time is slow compared to calculating the crossing matrix. The same chapter also shows that the minimal running time decreases when calculating the crossing matrix on demand. Since the barycenter heuristic already reduces the amount of crossings quite well, it can be expected that the on-demand calculation should improve the performance.

As can be seen in Figure 6.1, recounting all crossings for each possible switch quickly results in unacceptable run-times, as was expected. Furthermore, calculating the crossing matrix on-demand gives a significant performance benefit. Since all three methods return the same results, there is no reason not to always use the on-demand version.

Section 3.3.1 discusses two different approaches to the greedy switch heuristic. The one-sided method, comparing crossings between one free layer and one of its neighbours. And the two-sided method, comparing crossings to both sides of the free layer. The one-sided method needs to recount the crossings in the complete graph after each forward and backward sweep and should therefore perform more slowly. Furthermore the one-sided method is set to always sweep forwards and backwards each time, since many crossings would be missed otherwise. The two-sided method however stops as soon as no switch occurred in a sweep across the graph. This should also improve performance. Comparing upward and downward sweeps obviously should take at least twice as long since the algorithm is simply run twice. Furthermore, after each run of the algorithm the crossings need to be recounted to be able to judge which run was more successful. The hyperedge crossing counter by Spönemann et al. roughly has the same theoretical

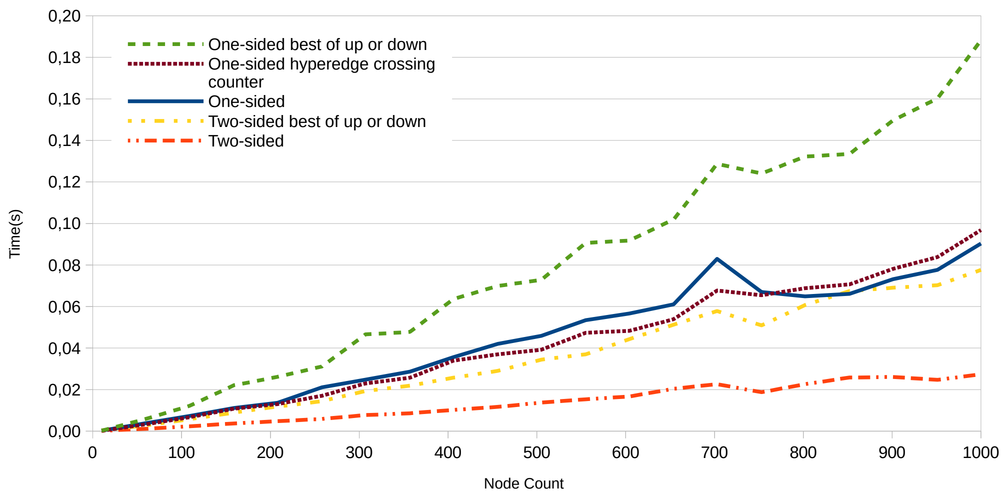


Figure 6.2. Running time for random graphs of increasing size with an average of 1.5 edges per node comparing one-sided and two-sided methods.

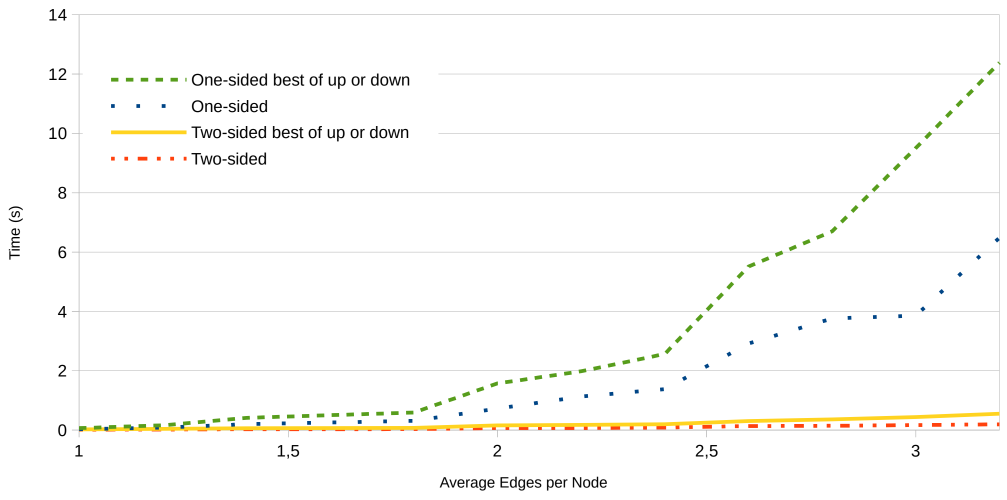


Figure 6.3. Running time for random graphs with 200 nodes of increasing density comparing one-sided and two-sided methods.

run-time as the straight edge counter by Barth et al. Therefore, performance should change little.

As expected, comparing the two-sided and one-sided methods shows a very clear performance advantage of the two-sided over the one-sided variant. This can already be seen when increasing the number of nodes and keeping the edge count constant (Figure 6.2), but results in extreme differences when increasing the density of the graph, seen in Figure 6.3. As has already been shown in

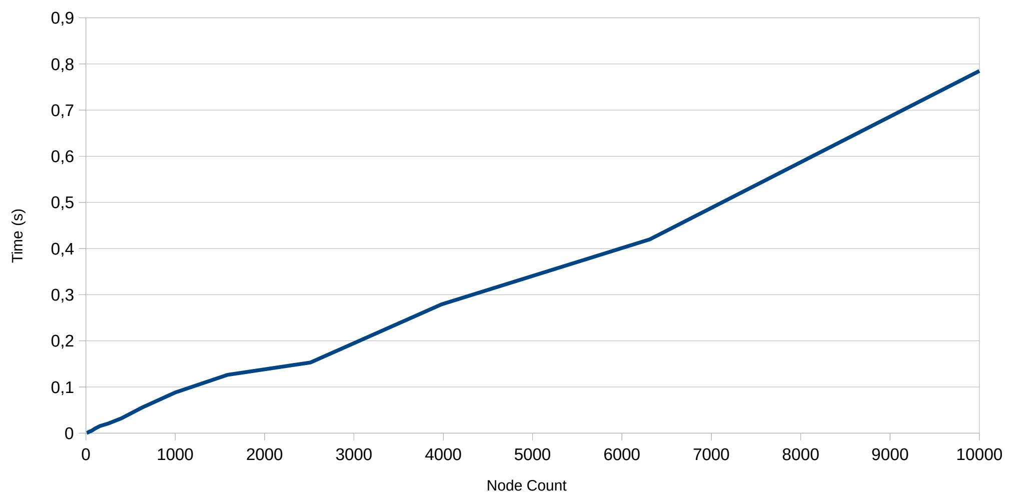


Figure 6.4. Extending the test of the two-sided methods to more extreme values, the diagram shows the run-time increase on graphs with 1.5 edges per node up to 10,000 nodes.

other research (see Section 1.1), the greedy switch algorithm very quickly slows when increasing graph density. The two-sided variant is much less prone to this problem however: While the one-sided variant already took an average of around 6.5 s for graphs with 200 nodes and an average of 3.2 edges per node, the two-sided variant still runs in acceptable speeds.

As expected, each algorithm takes more than twice as long when comparing upward or downward in-layer sweep direction. As shown in Figure 6.2, using the hyperedge crossing counter instead of the algorithm for counting straight line crossings does not significantly change the run-time.

To test the boundaries of the two-sided method, the run-time was evaluated when increasing the number of nodes and the density further.

As can be seen in Figure 6.4, increasing the node count to extreme measures on graphs with 1.5 edges per node results in a more or less linear increase in run-time, with reasonable speeds even for graphs with 10,000 nodes.

Performance suffers more quickly when increasing the density on graphs with 200 nodes, shown in Figure 6.5. Starting at around 3.5 edges per node, the duration of the algorithm starts to be a significant problem for real-time applications, taking 0.4 s. Higher densities might be feasible on more powerful hardware, however.

To put the density and node count numbers into perspective: The highest number of nodes in the collection of Ptolemy diagrams used for the second part of this evaluation was a diagram with 864 with an average of 1.02 edges per node. The highest density was 2.1 edges per node on a graph with 111 nodes.

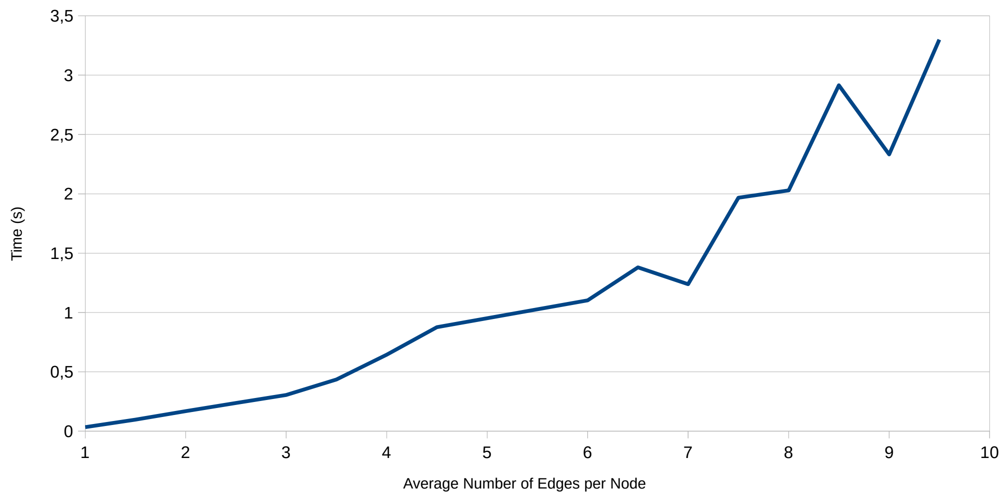


Figure 6.5. Extending the test of the two-sided methods to more extreme values, the diagram shows the run-time increase on graphs with 200 nodes when increasing the amount of edges per node up to 3.2.

	TS	TS UD	OS	OS UD	OS HCC
Maximum time	0.0299 s	0.0466 s	0.3820 s	0.3840 s	0.3870 s

Table 6.1. Maximum runtime using the Ptolemy diagrams. TS: Two-sided, OS: One-Sided, UD: Best of upward or downward sweep, HCC: Hyperedge crossing counter

Table 6.1 shows the maximum running time for five variants of the greedy switch heuristic layouting the 194 Ptolemy diagrams, Measuring the maximum run-time value is of course prone to errors, because single runs may be significantly slower due to circumstances in the system which are not under our control. In general however, these values confirm the previous evaluation on random graphs, showing a significant performance benefit of the two-sided methods. Note that the slowest run of the one-sided variant comparing upward and downward sweeps was on the graph with the highest density of 2.1 mentioned earlier. Using the hyperedge crossing counter instead of the normal crossing counter does not change the speed of the algorithm significantly.

In summary: The two-sided methods run fast enough so that they should rarely hamper user experience, while the one-sided methods may slow the performance of the KLay Layered algorithm significantly to be noticeable by a user especially when run on graphs with higher density. Comparing all upward and all downward sweeps comes with a clear performance penalty. Using the hyperedge crossing counter instead of the normal crossing counter does not change the running time significantly.

The following section compares the quality of the solutions proposed by the different greedy switch variants.

6.2 Quality

The following section measures solution quality of the greedy switch algorithm by the number of remaining crossings in the graph. The lower the number of crossings, the better the solutions. Note that we do not have an optimal algorithm for KLay Layered at our disposal. Comparing the remaining crossings to an optimal solution would give a more exact measure of solution quality.

Solution quality was first measured on random graphs with different configurations. For each configuration, be it a change in size or in density, five different random instances were created. The results are given using the average improvement of crossing numbers compared to using only barycenter.

Let us first consider the two most general approaches of taking into account two or one neighbouring layer when deciding whether to switch nodes. Section 3.3.1 showed hypothetical examples both where the one-sided method performs better than the two-sided method and other way around. Remember however that the one-sided method is the only one that can remove crossings caused by the order of north/south ports and is in general more *optimistic* in its approach, since the two-sided method only switches nodes when no new crossings can be created.

Indeed, when run on random graphs with increasing density (Figure 6.6) and with increasing number of nodes (Figure 6.7), the one sided method always shows a higher number of crossing reduction than the two sided method. On average for the increase of density shown in Figure 6.6 the one-sided method had 3,6% fewer crossings compared to the two sided method. The difference roughly stays the same independent of the density of the graph.

As seen in Figure 6.7, when keeping the number of edges per node constant at 1.5 and increasing the number of nodes up to graphs with 1000 nodes, greedy switch was more effective on small random graph instances. Once again, the one-sided method was superior to the two-sided method, showing an average difference of 3.9%.

Note that in both cases the amount of improvement was reduced when increasing the amount of nodes or the density of the graph. Previously published results showed that the performance of greedy switch improved with increasing density (see Section 1.1). Although Figure 6.6 seems to contradict these findings, bear in mind that the aforementioned research was examining the greedy switch heuristic as a separate heuristic replacing barycenter, not as a post-processing step after running barycenter. Furthermore the heuristics were being compared to an optimal algorithm, while Figure 6.6 shows the improvement over only using

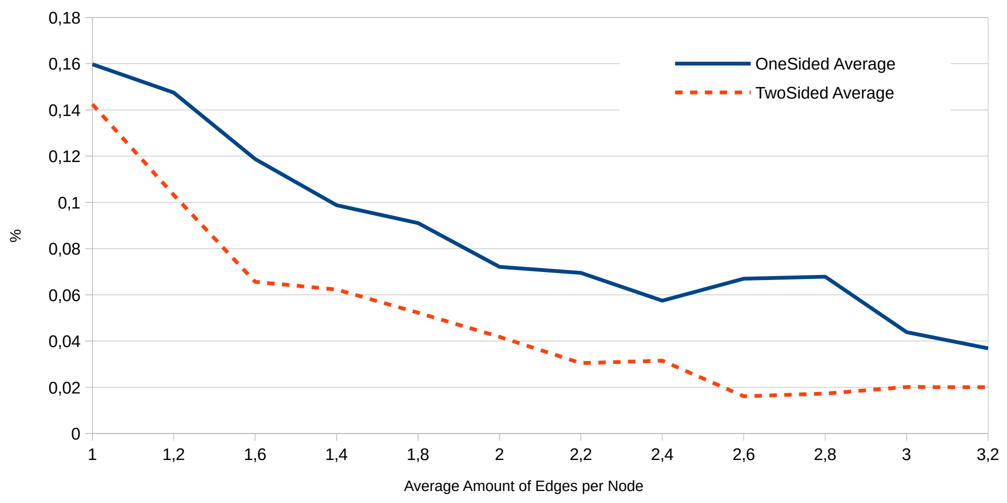


Figure 6.6. Percentage of improvement of crossing number to the barycenter algorithm relative to the number of edges per node run on random graphs with 200 nodes.

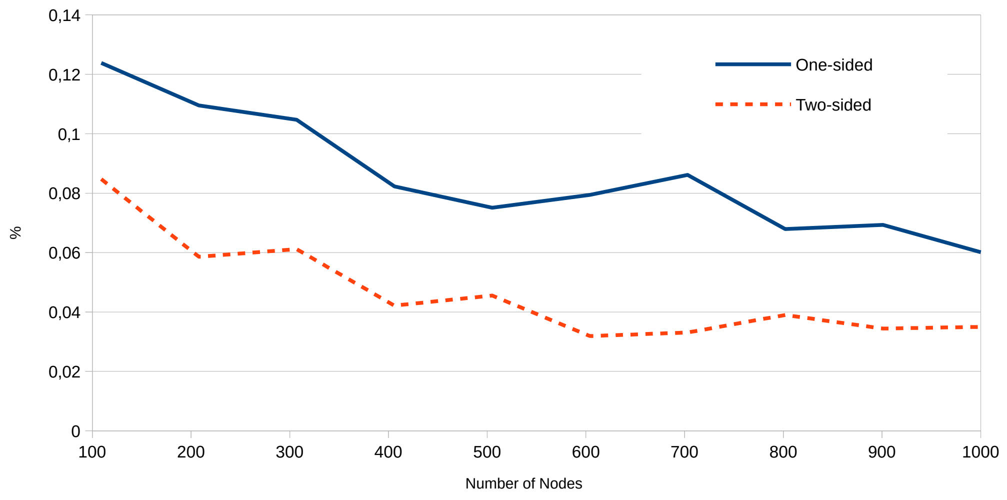


Figure 6.7. Improvement of the number of edge crossings when compared to the barycenter algorithm relative to the number of nodes in random graphs with an average of 1.5 edges per node.

barycenter.

Solution quality with and without trying both in-layer sweep directions was compared using the one-sided method and the same setup as for Figure 6.7. This version only gives an almost negligable further reduction in crossing number: On average just 0.4 % difference when compared to when this comparison was turned off. Moreover, as described earlier (see Figure 6.3), it comes with a significant

	TS	TS UD	OS	OS UD	OS HCC
Improv. of c	9.1%	8.7%	12%	11%	14%
Improv. of c_{total}	11%	10.7%	15.2%	14.1%	14.6%
Improv. of $\frac{c}{edgecount}$	12.7%	12.6%	14.3%	13.4%	15%
Improv. of $\frac{c_{total}}{edgecount_{total}}$	11.1%	10.8%	15.4%	14.3%	14.8%
c **increased**	5.2%	5.7%	5.2%	6.7%	1.5%

Table 6.2. TS: Two-sided, OS: One-Sided, UD: Best of upward or downward sweep, HCC: Using Hyperedge crossing counter, c: number of crossings, $edgecount$: number of edges in graph, c_{total}, $edgecount_{total}$: total number of crossings and edges in all 194 graphs. *Improv. of* c shows the average improvement of crossing numbers in the Ptolemy graphs. *Improv. of* c_{total} shows the sum of all crossings reduced compared to only running the barycenter heuristic. *Improv. of* $\frac{c}{edgecount}$ shows the average improvement of crossings per number of edges. *Improv. of* $\frac{c_{total}}{edgecount_{total}}$ once again sums up all crossings and edge counts. The last row shows the amount of graphs where the crossing number has *increased*.

performance penalty.

Let us now turn to the evaluation of quality when using the set of 194 Ptolemy graphs. Comparing solution quality using these graphs is not simple, because it is difficult to judge which data yields significant information. As an example, consider a graph with only one crossing and a graph with many crossings. If the one crossing in the first graph can be removed it will influence the result more heavily than if many crossings are removed in the second graph. Table 6.2 therefore shows a range of different data. Remember that Ptolemy diagrams are drawn with orthogonal hyperedges. In this case the number of resulting crossings can not be computed exactly beforehand. As discussed in Section 4.4, this can lead to layouts with more crossings than before executing the greedy switch heuristic.

As with the random graphs, we can see that the one-sided method results in fewer crossings than the two-sided variant.

Comparing upward and downward sweeps does not improve the graphs. In this case, these variants even perform worse on average. This can be explained by the higher number of graphs with an increased crossing number.

Using the hyperedge crossing counter reduces the number of instances with more crossings and therefore gives better average results. When comparing the sum of all crossings and edge counts however, the standard one-sided variant gives similar results. Note that while the results are not included in the table above, comparing up and down sweeps while using the hyperedge crossing counter returned exactly the same results, as when this feature was turned off.

In summary: The one-sided variants consistently return higher quality solutions than the two-sided methods. Comparing upward and downward sweeps does not improve the crossing number. And finally, using the hyperedge crossing counter on graphs with hyperedges does reduce the number of cases where the crossing number is worsened. Generally speaking, the reduction of edge crossings is large enough for both the one-sided and two side versions to justify using greedy switch as a post-processing step.

In general the improvement of the crossing number seems large enough to justify integrating the greedy switch heuristic permanently into KLay Layered. However there is a danger that the one-sided version can lead to a significant decrease of performance on graphs with high density. For this reason the two-sided method could be activated by default and the one-sided method enabled only when called by the user. Even though in most cases comparing the directions of sweeping in a single layer does not improve the number of crossings, the possibility should principally be kept in the source code, perhaps accessible to the user through an advanced option which is hidden by default. This is because there are some — albeit rare — cases where this option results in a clearly improved layout. When layouting graphs with hyperedges, the hyperedge crossing approximation algorithm should be activated by default, since this reduces the chance of creating worse layouts. Once again, the option to turn this off should be made possible in some form, since experience shows that in some graphs the results are better when crossings are counted as straight edge crossings.

The following chapter reviews the results from all previous chapters and suggests further venues of research.

Chapter 7

Conclusion

In this last chapter, short summarizing answers to three simple questions are given: What has been done in this thesis? What should be done using its results? And: What could be done in future?

The greedy switch heuristic is an algorithm which is slow and works badly when used separately to minimize crossings in layered graphs. However, the barycenter heuristic currently used in KLay Layered often results in characteristic errors which occasionally can be so obvious to users as to be especially bothersome. The solution to this problem examined here was to implement the greedy switch heuristic as a post-processing step to improve the number of crossings. For this, the performance of the greedy switch heuristic was improved by calculating the crossing matrix of the original heuristic on demand. To calculate the number of crossings exactly for graphs without hyperedges, several different algorithms were developed and implemented for the cases not already covered in KLay Layered. These deal with crossings caused by between-layer edges, in-layer edges and the ordering of north/south-ports. For each of these problems algorithms both for counting crossings in a complete layer and for counting crossings only for neighbouring nodes were developed. Two principally different variants were compared: The one-sided method fixes one neighbouring layer of a given free layer, while the two-sided method fixes both neighbouring layers. The one-sided method is slower while giving better results, while the two-sided method is faster while not improving the number of crossings as much. Furthermore, an attempt was made to improve the crossing count by comparing different directions to sweep in each layer. In some hypothetical cases sweeping from the lowest to the highest node in a layer can lead to an improved result. However, this improvement is not relevant enough to justify the performance penalty. Finally, it was found that using the heuristic for judging the amount of hyperedge crossings suggested by Spönemann et al. ([SSRvH14]) for preventing an increase of crossings in the one-sided method reduces the number of cases of increasing the crossing count when layouting graphs with hyperedges.

For a final integration into KLay Layered, the two-sided method could be activated by default and the one-sided method be available to the user. When layouting graphs with hyperedges the hyperedge crossing approximation algorithm should be used by default. Comparing upward and downward sweeps should only be kept as a hidden options for the rare cases it can actually improve the graph. A final integration would furthermore include the barycenter algorithm using the same crossing counting code as the greedy switch heuristic. This could also improve the results of the barycenter algorithm in some cases.

The following section discusses further venues of research and development which could extend these results.

7.1 Future Work

In order to avoid using the potentially slow one-sided variant, the barycenter heuristic could be tweaked to reduce the difference between the two variants. Remember that the barycenter algorithm does not take the positions of fixed north/south ports into account. This results in many unnecessary and obvious crossings. By changing the barycenter value of the connected north/south-dummies depending on the direction of their between-layer edge and the position of the north/south ports, this situation could be improved. Due to the nature of the algorithms, only the slower one-sided method can change the order of north/south dummies. Changing the barycenter heuristic in this way might be a more efficient way of removing these crossings.

Using the hyperedge crossing counter to decide if a layout has been worsened by the one-sided variant reduces the amount of errors caused by the greedy switch heuristic when layouting graphs with hyperedges. Conceivably there could be a way of already taking possible hyperedge crossings into account when deciding whether or not to switch two neighbouring nodes. This might improve the layout of both greedy switch variants when layouting graphs with hyperedges.

Gansner et al. [GNV88] also use greedy switch as a post-processing step. On every second forward and backward traversal they also switch nodes when the crossing amount does not change. This approach was not implemented in this thesis and could be examined in future research.

The crossing counting code implemented for the greedy switch algorithm in KLay Layered can quite easily be reused to implement other heuristics which use the crossing matrix as a basis for their work. These heuristics could be a possible replacement of or alternative to the current barycenter algorithm and could still be followed by the greedy switch post-processor. Note that all of these methods result in a slower run-time than the barycenter heuristic, but return better results according to the respective authors. The following two paragraphs describe some of these methods using the crossing matrix.

An elegant and simple heuristic is a sifting method originally suggested by Matuszewski et al. [MSM99]. Starting from a predefined order, for every node in a layer each possible position for that node is considered while keeping the relative order of the other nodes fixed. Then the best position is chosen. This heuristic obviously has quadratic run-time. Since the success of the algorithm does depend on the starting layout, this method could conceivably also be used as a post-processing step. However, contrary to greedy switch, the algorithm would not run faster when only few crossings remain. The approach was adapted by Bachmeier et al. to include in-layer edges [BF06] and even as an approach for minimizing all crossings in a graph without using the usual method of comparing only two neighbouring layers [BBBH10]. According to the authors, these methods give good solutions in reasonable computation time.

An interesting variant of the greedy insert approach was introduced by Yamaguchi et al. [YS99]. The greedy insert approach inserts one node at a time add the end of the layer, always choosing the node which has the fewest number of crossings to the nodes already inserted. The algorithm by Yamaguchi et al. instead uses the crossing matrix to calculate a rather intuitive heuristic function which takes into account the number of crossings caused by this vertex to vertices *not yet placed* and chooses the node which minimizes this value.

As a final interesting option, let us quickly consider the possibility of implementing optimal algorithms for small problem cases in KLay Layered. Jünger et al. come to the conclusion that for the two-layer one-sided crossing minimization problem, their suggested optimal algorithm is sufficiently fast for up to 60 nodes and therefore there is no need for further heuristics[JM97]. Their results suggest that for real-time applications, this does not hold. Note however, that the article was published in 1996 and recent hardware would certainly yield different results. Newer optimal algorithms have been suggested (e. g., [CMB08]). Most are integer linear programming solutions, others use an approach based on semidefinite programs (e. g., [CHJM12]). Integrating such approaches in KLay Layered would include a heuristic for judging whether the size of the graph could pose a run-time problem for the use of these algorithms.

Bibliography

[BBBH10] Christian Bachmaier, FranzJ. Brandenburg, Wolfgang Brunner, and Ferdinand Hübner. A global k-level crossing reduction algorithm. In Md.Saidur Rahman and Satoshi Fujita, editors, *WALCOM: Algorithms and Computation*, volume 5942 of *Lecture Notes in Computer Science*, pages 70–81. Springer Berlin Heidelberg, 2010. URL: http://dx.doi.org/10.1007/978-3-642-11440-3_7, doi:10.1007/978-3-642-11440-3_7.

[BF06] Christian Bachmaier and Michael Forster. Crossing reduction for hierarchical graphs with intra-level edges, 2006.

[BJM02] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting, 2002.

[Cat95] T. Catarci. The assignment heuristic for crossing reduction. *Systems, Man and Cybernetics, IEEE Transactions on*, 25(3):515–521, Mar 1995. doi:10.1109/21.364865.

[CHJM12] Markus Chimani, Philipp Hungerländer, Michael Jünger, and Petra Mutzel. An sdp approach to multi-level crossing minimization. *J. Exp. Algorithmics*, 17:3.3:3.1–3.3:3.26, September 2012. URL: http://doi.acm.org/10.1145/2133803.2330084, doi:10.1145/2133803.2330084.

[CMB08] Markus Chimani, Petra Mutzel, and Immanuel M. Bomze. A new approach to exact crossing minimization. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, pages 284–296, 2008. URL: http://dx.doi.org/10.1007/978-3-540-87744-8_24, doi:10.1007/978-3-540-87744-8_24.

[EK86] Peter Eades and David Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria*, 21:89–98, 1986.

[EW94] Peter Eades and Nicholas C Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. Dag—a program that draws directed graphs. *Software: Practice and Experience*, 18(11):1047–1062, 1988. URL: http://dx.doi.org/10.1002/spe.4380181104, doi:10.1002/spe.4380181104.

[JM97] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms, 1997. This article was published in 1997 by Brown University in the journal "Journal of Graph Algorithms and Applications", volume 1, pages 1-25. URL: http://gdea.informatik.uni-koeln.de/22/.

[LMV97] Manuel Laguna, Rafael Martí, and Vicente Valls. Arc crossing minimization in hierarchical digraphs with tabu search. *Computers and Operations Research*, 24:1175–1186, 1997.

[ML03] Rafael Martì and Manuel Laguna. Heuristics and meta-heuristics for 2-layer straight line crossing minimization. *Discrete Applied Mathematics*, 127(3):665 – 678, 2003. URL: http://www.sciencedirect.com/science/article/pii/S0166218X02003979, doi:http://dx.doi.org/10.1016/S0166-218X(02)00397-9.

[MSM99] Christian Matuszewski, Robby Schönfeld, and Paul Molitor. Using sifting for k-layer straightline crossing minimization. In Jan Kratochvíyl, editor, *Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 217–224. Springer Berlin Heidelberg, 1999. URL: http://dx.doi.org/10.1007/3-540-46648-7_22, doi:10.1007/3-540-46648-7_22.

[Mä90] Erkki Mäkinen. Experiments on drawing 2-level hierarchical graphs. *International Journal of Computer Mathematics*, 37(3-4):129–135, 1990. URL: http://dx.doi.org/10.1080/00207169008803941, arXiv:http://dx.doi.org/10.1080/00207169008803941, doi:10.1080/00207169008803941.

[Sch11] Christoph Daniel Schulze. Optimizing auto layout for data flow graphs, 2011.

[SFvHM10] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In David Eppstein and EmdenR. Gansner, editors, *Graph Drawing*, volume 5849 of *Lecture Notes in Computer Science*, pages 135–146. Springer Berlin Heidelberg, 2010. URL: http://dx.doi.org/10.1007/978-3-642-11805-0_14, doi:10.1007/978-3-642-11805-0_14.

[SSRvH14] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. Counting crossings for layered hypergraphs. In Tim Dwyer, Helen Purchase, and Aidan Delaney, editors, *Diagrammatic Representation and Inference*, volume 8578 of *Lecture Notes in Computer Science*, pages 9–15. Springer Berlin Heidelberg, 2014. URL: http://dx.doi.org/10.1007/978-3-662-44043-8_2, doi:10.1007/978-3-662-44043-8_2.

[SSvH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout*, 25(2):89–106, 2014. doi:10.1016/j.jvlc.2013.11.005.

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

[YS99] Atsuko Yamaguchi and Akihiro Sugimoto. An approximation algorithm for the two-layered graph drawing problem. In Takano Asano, Hideki Imai, D.T. Lee, Shin-ichi Nakano, and Takeshi Tokuyama, editors, *Computing and Combinatorics*, volume 1627 of *Lecture Notes in Computer Science*, pages 81–91. Springer Berlin Heidelberg, 1999. URL: `http://dx.doi.org/10.1007/3-540-48686-0_8`, `doi:10.1007/3-540-48686-0_8`.

List of Figures