The Five Phases

This page describes the algorithm's five main phases and the available implementations. You can find a list and descriptions of the intermediate processors over here.

Phase 1: Cycle Removal

The first phase makes sure that the graph doesn't contain any cycles. In some papers, this phase is an implicated part of the layering. This is due to the supporting function cycle removal has for layering: without cycles, we can find a topological ordering of the graph's nodes, which greatly simplifies layering.

An important part to note is that cycles may not be broken by removing one of their edges. If we did that, the edge would not be routed later, not to speak of other complications that would ensue. Instead, cycles must be broken by reversing one of their edges. Since the problem of finding the minimal set of edges to reverse to make a graph cycle-free is NP-hard, (or NP-complete? I don't remember from the top of my head) cycle removers will implement some heuristic to do their work.

The reversed edges have to be restored at some point. There's a processor for that, called ReversedEdgeRestorer. All implementations of phase one must include a dependency on that processor, to be included after phase 5.

Preconditions	• No node is assigned to a layer yet.
Postconditions	• The graph is now cycle-free. Still, no node is assigned to a layer yet.
Remarks	• All implementations of phase one must include a dependency on the `ReversedEdgeRestorer`, to be included after phase five.
Implementations	• `GreedyCycleBreaker`. Uses a greedy approach to cycle-breaking, inspired by ◦ Peter Eades, Xuemin Lin, W. F. Smyth, A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters* 47(6), pp. 319-323, 1993. • `InteractiveCycleBreaker.` Detects feedback edges according to the current layout, hence it reacts to the user's placement.
Tests	• The graph contains no cycles.

Phase 2: Layering

The second phase assigns nodes to layers. (also called *ranks* in some papers) Nodes in the same layer are assigned the same x coordinate. (give or take) The problem to solve here is to assign each node x a layer i such that each successor of x is in a layer j>i. The only exception are self-loops, that may or may not be supported by later phases.

It must be differentiated between a *layering* and a *proper layering*. In a layering, the above condition holds. (well, and self-loops are allowed) In a proper layering, each successor of x is required to be assigned to layer i+1. This is possible only for the simplest cases, but may be required by later phases. In that case, later phases use the LongEdgeSplitter processor to turn a layering into a proper layering by inserting dummy nodes as necessary.

Note that nodes can have a property associated with them that constraints the layers they can be placed in.

Preconditions	• The graph is cycle-free. • The nodes have not been layered yet.
Postconditions	• The graph has a layering.
Remarks	• Implementations should usually include a dependency on the `LayerConstraintHandler`, unless they already adhere to layer constraints themselves.

Implementations	<ul><li>`LongestPathLayerer`. Layers nodes according to the longest paths between them. Very simple, and doesn't usually give the best results.</li><li>`NetworkSimplexLayerer`. A way more sophisticated algorithm whose results are usually very good, inspired by<ul><li>Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, Kiem-Phong Vo, A technique for drawing directed graphs. *Software Engineering* 19(3), pp. 214-230, 1993.</li></ul></li><li>`InteractiveLayerer.` Detects layers according to the current layout, hence it reacts to the user's placement.</li></ul>
Tests	<ul><li>The set of layerless nodes is empty.</li><li>For every edge that points from layer `i` to layer `j, i<j` holds.</li><li>No empty layer exists.</li><li>All nodes that existed prior to this phase are assigned to a layer.</li></ul>

Phase 3: Crossing Reduction

The objective of phase 3 is to determine how the nodes in each layer should be ordered. The order determines the number of edge crossings, and thus is a critical step towards readable diagrams. Unfortunately, the problem is NP-hard even for only two layers. Did I just hear you say "heuristic"? The usual approach is to sweep through the pairs of layers from left to right and back, along the way applying some heuristic to minimize crossings between each pair of layers. The two most prominent and well-studied kinds of heuristics used here are the barycenter method and the median method. We have currently implemented the former.

Our crossing reduction implementations may or may not support the concepts of node successor constraints and layout groups. The former allows a node x to specify a node y!=x that may only appear after x. Layout groups are groups of nodes. Nodes belonging to different layout groups are not to be interleaved.

Preconditions	<ul><li>The graph has a proper layering. (except for self-loops)</li><li>An implementation may allow in-layer connections.</li><li>Usually, all nodes are required to have a least fixed port sides.</li></ul>
Postconditions	<ul><li>The order of nodes in each layer is fixed.</li><li>All nodes have a fixed port order.</li></ul>
Remarks	<ul><li>If fixed port sides are required, the `PortPositionProcessor` may be of use.</li><li>Support for in-layer connections may be required to be able to handle certain problems. (odd port sides, for instance)</li></ul>
Implementations	<ul><li>`LayerSweepCrossingMinmizer`. Does several sweeps across the layers, minimizing the crossings between each pair of layers using a barycenter heuristic. Supports node successor constraints and layout groups. Node successor constraints require one node to appear before another node. Layout groups specify sets of nodes whose nodes must not be interleaved. See this page for more information.</li><li>`InteractiveCrossingMinimizer.` Detects the order of nodes according to the current layout, hence it reacts to the user's placement.</li></ul>
Tests	<ul><li>All nodes remain in their respective layer.</li></ul>

Phase 4: Node Placement

So far, the coordinates of the nodes have not been touched. That's about to change in phase 4, which determines the y coordinate. While phase 3 has an impact on the number of edge crossings, phase 4 has an influence on the number of edge bends. Usually, some kind of heuristic is employed to yield a good y coordinate.

Our node placers may or may not support node margins. Node margins define the space occupied by ports, labels and such. The idea is to keep that space free from edges and other nodes.

Preconditions	<ul><li>The graph has a proper layering. (except for self-loops)</li><li>Node orders are fixed.</li><li>Port positions are fixed.</li><li>An implementation may allow in-layer connections.</li><li>An implementation may require node margins to be set.</li></ul>

Postconditions	• Each node is assigned a y coordinate such that no two nodes overlap. • The height of each layer is set. • The height of the graph is set to the maximal layer height.
Remarks	• Support for in-layer connections may be required to be able to handle certain problems. (odd port sides, for instance) • If node margins are supported, the `NodeMarginCalculator` can compute them. • Port positions can be fixed by using the `PortPositionProcessor`.
Implementations	• `LinearSegmentsNodePlacer`. Builds linear segments of nodes that should have the same y coordinate and tries to respect those linear segments. Linear segments are placed according to a barycenter heuristic. Inspired by ◦ Georg Sander, A fast heuristic for hierarchical Manhattan layout. In *Proceedings of the Symposium on Graph Drawing (GD'95)*, LNCS vol. 1027, pp. 447-458, Springer, 1996. • `BKNodePlacer.` Assembles nodes into blocks placed in straight lines in an attempt to minimize the number of edge bends, similar to the linear segments node placer. However, instead of using a barycenter heuristic to place nodes, the placement also tries to minimize the number of edge bends, usually resulting in diagrams that require more space. ◦ Ulrik Brandes and Boris Köpf, Fast and simple horizontal coordinate assignment. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, LNCS vol. 2265, pp. 33-36, Springer, 2002. • `InteractiveNodePlacer`. Tries to keep the preset coordinates of nodes from the original layout. For dummy nodes, a guess is made to infer their coordinates. Requires the other interactive phase implementations to have run as well.
Tests	• The nodes of a layer are strictly ordered with regards to their y coordinate. • Any two nodes of a layer do not overlap with regards to their bounding box (height + margins).

Phase 5: Edge Routing

In the last phase, it's time to determine x coordinates for all nodes and route the edges. The routing may support very different kinds of features, such as support for odd port sides, (input ports that are on the node's right side) orthogonal edges, spline edges etc. Often times, the set of features supported by an edge router largely determines the intermediate processors used during the layout process.

Preconditions	• The graph has a proper layering. (except for self-loops) • Nodes are assigned y coordinates. • Layer heights are correctly set. • An implementation may allow in-layer connections.
Postconditions	• Nodes are assigned x coordinates. • Layer widths are set. • The graph's width is set. • The bend points of all edges are set.
Remarks	None.
Implementations	• `OrthogonalEdgeRouter`. Routes edges orthogonally. Supports routing edges going into an eastern port around a node. Tries to minimize the width of the space between each pair of layers used for edge routing. Inspired by ◦ Georg Sander, Layout of directed hypergraphs with orthogonal hyperedges. In *Proceedings of the 11th International Symposium on Graph Drawing (GD '03)*, LNCS vol. 2912, pp. 381-386, Springer, 2004. • `PolylineEdgeRouter`. Simplest routing style that just inserts bend points at the position of long edge dummy nodes. • `SplineEdgeRouter`. A simple method for routing the edges with splines. Uses the long edge dummy nodes as reference points for spline calculation.
Tests	• `OrthogonalEdgeRouter`: For any two succeeding bendpoints of an edge either the y coordinates or the x coordinates are the same. Starting with the same x coordinates, alternating equality is required.