Bachelor's Thesis

Automatic Layout Generation for Flow-Based Visual Programming Environments

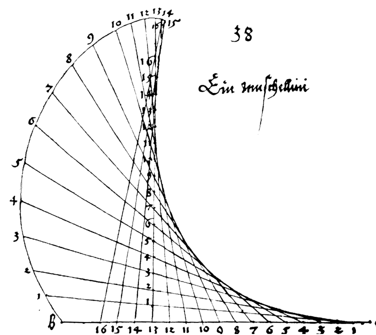
by

Tobias Mertz

Supervisor:
Prof. Dr. Gabriel Wittum
Michael Hoffer

Goethe Center for Scientific Computing (G-CSC)
Goethe University Frankfurt am Main

August 14, 2016



Declaration of Authorship

I hereby formally declare that the contents of this thesis were written by myself without the use of any sources other than those references within the attached bibliography. This thesis has not been submitted or published anywhere else in this or any other form.

Tobias Mertz; August 14, 2016; Frankfurt a. M.

Abstract

To accommodate the growth of the software industry, programming languages are getting increasingly easy to use. The latest trend in the simplification of the software development process is the usage of visual programming environments. To make visual programming effective, the graph-like representation of the source code must be clearly arranged. This thesis details some of the difficulties in automatic layout generation and proposes an interface as well as two different implementations of automatic layout generators to integrate into the *VWorkflows* visual programming framework.

Keywords: Java, Visual, Programming, Graph, Layout, JUNG

List of Figures

3.1 Result of the *JUNG DAG* Layout on the *Simple Example Graph.* 8
3.2 Result of the *JUNG KK* Layout on the *Simple Example Graph.* . 9
3.3 Result of the *JUNG FR* Layout on the *Simple Example Graph.* . 11
3.4 Result of the *JUNG ISOM* Layout on the *Simple Example Graph.* 12
4.1 Result of the Final Naive Algorithm on the *Simple Example Graph.* 19
4.2 Projection of a Distance Vector onto the *General Flow Direction.* 24
4.3 Semantics behind the Results for the Displacement Factor ϕ. . . 25
4.4 Results of the Smart Algorithm after each Step. 29
4.5 Result of the Final Smart Algorithm on the *Simple Example Graph.* 33
5.1 Symmetric and Planar drawings of a *graph.* Source:[12] 35
5.2 Percental Test Results of Phase I 36
5.3 Percental Test Results of Phase II 37
5.4 Example Result for Phase II test *'MainWithForAndMore'*. 38
5.5 Comparison of Naive (Orange) and Smart (Light Gray) Results for Testcase *'Sizes3'*. 39
5.6 Comparison of Naive (Orange) and Smart (Light Gray) Results for Testcase *'MainWithForAndMore'*. 39
5.7 Example Result of the Naive Algorithm with a Very Unnecessary Edge Crossing. 40
6.1 Comparison of Smart Results with (Orange) and without (Light Gray) *JUNG* Layout. 42

List of Tables

5.1 Test Results and Variances in Phase II 37
B.1 Phase I Test Results of the Smart Algorithm 51
B.2 Phase I Test Results of the Naive Algorithm 51
B.3 Phase II Test Results of the Smart Algorithm 52
B.4 Phase II Test Results of the Naive Algorithm 52

List of Listings

4.1 Step 1 of the Naive Algorithm. 15
4.2 Step 2 of the Naive Algorithm 15
4.3 Step 3 of the Naive Algorithm 16
4.4 The Cycle Removal Algorithm 17
4.5 Step 1 of the Smart Algorithm 21
4.6 Width of the Longest Path 22
4.7 Step 3 of the Smart Algorithm 26
4.8 Step 4 of the Smart Algorithm 27
4.9 Calculation of the Desired Node Distance 28

Contents

Declaration of Authorship **i**

Abstract **iii**

1 Introduction **1**

2 The Problem **2**
2.1 Criteria in Layout Quality Assessment 2
2.2 Definitions 3

3 Technological Background **5**
3.1 VWorkflows 5
3.2 The Java Universal Network/Graph Framework 6

4 Implementation **13**
4.1 The LayoutGenerator Interface 13
4.2 The Naive Layout Generator 14
4.3 The Smart Layout Generator 19

5 Evaluation **34**
5.1 Methodology 34
5.2 Test Phase I - General Graphs 35
5.3 Test Phase II - VRL Graphs 36
5.4 Discussion 38

6 Horizon **41**
6.1 LayoutGeneratorNaive 41
6.2 LayoutGeneratorSmart 41

7 Conclusion **43**

A Fields and Methods **46**
A.1 LayoutGeneratorNaive 46
A.2 LayoutGeneratorSmart 47

B Testresults **51**

C Glossary **53**

D List of CD Contents **54**

1 Introduction

With the steady evolution of computer technology, computers become increasingly small and powerful. This enables programmers to implement ever more complex programs on a constantly growing field of applications. As programs get more complex and the underlying source code gets larger, projects can become more and more confusing for the programmers working on them, especially for those joining a development in progress. To familiarize themselves with the code, before continuing development, programmers require time. In a market with an ever increasing demand for software, this time is difficult to allocate. This is why large software development companies often work on their own development tools, to further abstract code and fit the code itself to their special requirements. This lowers the amount of time required to understand the code and, therefore, increases productivity. Examples of this procedure are Mozilla's *Rust* [1] and Google's *Go* [2].
The first example of the further abstraction of source code to simplify software development was in the transition from assembly based programming to the widespread usage of higher level programming languages such as Java. The next step in this evolution could be the utilization of visual programming languages.

One such visual programming language is the *Visual Reflection Library* (VRL) introduced by *Hoffer et al.* in [3]. The *VRL* can be used to automatically create a graph-like view of the structure of a *Groovy* program. In this view dataflow and controlflow can be displayed as edges in the graph, while functions and variables are displayed as vertices. This enables programmers to more easily understand the process flow of a program, even if it was designed by someone else. Also, significant changes to the program can easily be implemented through changes in control- or dataflow edges. These advantages of visual programming languages can, however, only be effective if the graph is neatly arranged, so that individual structures within the graph are clearly visible and, therefore, easy to understand.

The environment designed at the *Goethe Center for Scientific Computing* to be used for the development in *VRL* is the *VRL-Studio* [4], which, in turn, uses the *VWorkflows* framework [5], which was also developed by *Hoffer*, for its graph model interaction and visualization features.

The task of this thesis is the development of an interface as well as two different implementations for automatic layout generation algorithms to integrate into the *VWorkflows* framework, so they can be used by the *VRL-Studio* and other *VWorkflows* based applications.

2 The Problem

2.1 Criteria in Layout Quality Assessment

The underlying problem of this task is the calculation of an aesthetically pleasing layout for any directed graph. This problem is hard to formally define, since the aesthetically pleasing nature of a graph layout is highly subjective.
When reading various papers of the field, such as [6], [7] or [8], one recognizes a few key attributes that are frequently used to compare graph layouts. While many aspects of layouts are still left to the personal preference of the observer, these attributes highlight certain common features among layouts that are often classified as aesthetically pleasing. These features are:

- The layout must contain a minimal number of edge crossings.
- The layout must uniformly distribute vertices in the available space.
- The layout must represent symmetric structures within the graph symmetrically.

These features are often considered to be of substantial importance when trying to optimize a graph layout for clarity. In the special scenario of graph layouts on visualized programming code, there are, however, some additional challenges and features to consider. While vertices in traditional graphs are just points, which can be displayed as any shape of any arbitrary scale, vertices in program flow graphs have contents. These contents can be values in the case of vertices representing variables or they can be whole other program flow graphs if the vertex in question is representing a function.

This implies that vertices in program flow graphs have to be scaled appropriately in regards to their contents. That, however, causes different vertices to have different sizes, which can not be arbitrarily scaled to fit the available coordinate space. As a consequence, the arrangement of vertices of varying sizes using traditional layout algorithms without modification can easily lead to overlaps between those vertices. These overlaps must be avoided, since they prevent the contents of those vertices from being read clearly and the edges between them from being visible.

Another important aspect to keep in mind when creating a layout for a program flow graph is that the graph represents a flow, for example a controlflow. The word flow typically describes a physical movement, which implies a representation as a vector with a certain length and a certain direction. This vector

can be partitioned into several partial vectors, which are represented as the edges in the graph. This means, however, that all edges of a program flow graph together amount to the overall flow vector of the program.

To make it easier for the observer to orient themselves within the program flow and to determine the overall flow vector of the program, all edges in the graph should have a direction similar to the direction of the overall flow vector. Progressing further through this thesis, the direction of the overall flow vector will be referred to as the *general flow direction*.

It is because of these reasons that the application of graph layouts on program flow graphs has more criteria to consider than on general graphs. The three additional important features to be extracted out of these problems are:

- The layout must scale vertices according to their contents.
- The layout must contain a minimal amount of overlap between vertices.
- The layout must abide by a constant *general flow direction*.

These six features will be the criteria used to determine the quality of layouts throughout this thesis.

2.2 Definitions

For future clarity, some terms used throughout this thesis need to be defined formally:

graph In this thesis, the term *graph* refers to directed graphs if not explicitly stated otherwise. Therefore, a *graph* G is a tuple (V, E) consisting of two sets V and E. V is a set of n vertices v_i while E is a set of m two-tuples (v_i, v_j) with $v_i, v_j \in V$ representing directed edges pointing from the vertex v_i to v_j.

The terms *vertex* and *node* as well as the terms *edge* and *connection* will be used synonymously.

origin *Origin* nodes are all nodes within the *graph*, which have an in-degree of zero, meaning they do not have any predecessors.

Furthermore the terms *in front* and *behind* are used throughout this thesis referring to the positioning of vertices relative to each other:

in front The term *in front* in this context means if vertex v_i is *in front* of vertex v_j, then v_i has a smaller coordinate on the axis of the *general flow direction*, so that an edge from v_i to v_j runs in the *general flow direction*.

behind Analogous to that, the term *behind* describes the opposite arrangement of the nodes v_i and v_j whereby an edge from v_i to v_j runs against the *general flow direction* if v_i is *behind* v_j.

Since functions in visual programming environments like *VRL-Studio* are displayed as program flows, function calls within programs are displayed as nodes, which contain another program flow. Consequently the term *subflow* needs to be defined.

subflow A *subflow* is a program flow, that is contained within a node of another program flow.

subflow node The nodes containing a *subflow* will be called *subflow nodes* for the remainder of this thesis. For future clarity it is important to make the distinction between *subflow nodes*, which are nodes that contain a *subflow*, and the nodes of a subflow, which are regular nodes contained within a *subflow*.

3 Technological Background

3.1 VWorkflows

The *VWorkflows* framework is an interface based library that provides the necessary *graph* modeling functionality to implement a flow based visual programming environment. The library itself is implemented in Java and uses JavaFX for its visual components. Since the project in its entirety is too large and complex to be explained in full in this section, only the parts necessary to the understanding of this thesis will be explained.
Flow based programming is based on the visualization of the flow of a program in the form of a *graph*. *Graphs* consist of nodes and edges. These different *graph* objects are provided by the *VWorkflows* library through the following interfaces:

3.1.1 VFlowModel

The `VFlowModel` interface is an extension of the `FlowModel` interface and is used to model a general workflow *graph*. A basic implementation of the `VFlowModel` interface is given by the `VFlowModelImpl` class.
Each `VFlowModel` object provides the functionality to create nodes and edges within the *graph* as well as gather lists of the already existing nodes and edges. The `FlowModel` interface also provides the method `connect()`, which takes two `Connectors` and connects them using a newly created `Connection`.
The `VFlowModel` interface is, however, also an extension of the `VNode` interface, which means that every *graph* can also be displayed as a node containing this *graph*.
Each `VFlowModel`, therefore, has the attribute *depth*. The *depth* of a `VFlowModel` describes the hierarchical position of the *graph* within the entirety of the project. The highest level flow *graph* (also called *root flow*) has a *depth* of zero, while each *subflow* of a certain *graph* always has a *depth* of one larger than its parent flow.

3.1.2 VNode

The `VNode` interface describes a general node of the *graph*. An implementation is provided in the `VNodeImpl` class. Each `VNode` has certain attributes that can be accessed through individual getter and setter methods. These attributes include: a unique id, x- and y-coordinates, width, height and a list of output- as well as a list of input-`Connectors`.

3.1.3 Connector

These `Connectors` are given by the `Connector` interface and the `ConnectorImpl` class. They represent the slots on each `VNode` that `Connections` can be attached to.
Each `Connector` can be connected to other `Connectors` of the same type through the `connect()` method of the `FlowModel` interface and each `Connector` can return the `VNode` it is attached to.

3.1.4 Connection

The template for the implementation of *graph* edges is provided by the interface `Connection` with its basic implementation in the `ConnectionBase` class. Each `Connection` has a *sender* and *receiver* of type `Connector` as attribute, as well as a unique id and a type, which are both String arguments.
Through the remainder of this thesis the terms *sender* and *receiver* will, however, for the sake of simplicity not refer to the individual `Connector` objects, but the `VNodes` they are attached to.

3.2 The Java Universal Network/Graph Framework

The Java Universal Network/Graph Framework [9] (or *JUNG* for short) implements a wide array of functionality to draw and display different types of data visualizations in Java. This includes *graph* visualization as well as *graph* layout generation, which is why some of *JUNG's* algorithms were used in the creation of this thesis.
The library provides a lot of content, of which only a small subset is used. Therefore, this section will be limited to the explanation of the important used classes and functions.

3.2.1 DirectedGraph

The library contains a `DirectedGraph` interface, which is an extension of the `Graph` interface. An implementation of the `DirectedGraph` can be found in the class `DirectedSparseGraph`. This class can be used to model a directed *graph* using node and edge objects. The interface uses Java generics to set the types of these node and edge objects on declaration. Furthermore, the interface declares methods to add nodes and return the already existing nodes in a list. The predecessors and successors of certain nodes as well as their amount can also be returned. Edges can be added, removed, returned and counted. Also, the graph object can return lists of all incident edges to a certain node or of all edges that connect two nodes directly.

3.2.2 Pair

In some methods, data must be arranged as tuples. Luckily the *JUNG* library provides a simple implementation of a tuple with two components of the same

type in the class `Pair`. The class uses Java generics to set the type of the components on declaration. The components can be written in the constructor of the class and can later be accessed through the methods `getFirst()` and `getSecond()`.

3.2.3 Layout

The *JUNG* library also provides a `Layout` interface with many different implementations. Each layout can be instantiated and then applied to a *graph*. The interface uses Java generics to declare the type of vertices and edges within the *graph*, analogous to the `Graph` interface. To apply the `Layout` to a `Graph`, the generic type declaration of the `Graph` object must, therefore, coincide with the types declared for the `Layout` object. The coordinates of a certain vertex of the *graph* can then be extracted through the use of the `transform()` method of the `Layout` object, which takes a vertex of the generic type as argument.

The layouts that were applicable to `DirectedGraph` objects and also seem to provide helpful functionality are:

1. Directed Acyclic Graph Layout (*DAG* layout)
2. Kamada and Kawai Layout (*KK* layout)
3. Fruchterman Reingold Layout (*FR* layout)
4. Inverse Self Organizing Map Layout (*ISOM* layout)

which will be described in the following.

Directed Acyclic Graph Layout

The *DAG* layout takes a layered hierarchical approach to *graph* layout as it was first described by *Sugiyama et al.* in [10]. Nodes are arranged on different layers[1] that are stacked on top of each other. Nodes with an out-degree of zero will be on the highest layer. Each predecessor to a node on this layer will be on the next lower layer. If a node is the *sender* of edges to multiple nodes on different layers, it is placed on the layer beneath the lowest of its successor nodes.
After all nodes are assigned their layers, dummy vertices are introduced for each edge, that spans multiple layers. To elaborate:

If an edge $e = (v_1, v_2)$ spans multiple layers, for example $l(v_2) = l(v_1) + 2$, whereby $l(v_x)$ returns the index of the layer that was assigned to the vertex v_x, the edge is removed and in its stead a dummy vertex v_d and two edges e_{d1} and e_{d2} are introduced so that $e_{d1} = (v_1, v_d)$, $e_{d2} = (v_d, v_2)$ and $l(v_d) = l(v_1)+1$.

As the next step, the horizontal ordering of nodes on each layer is calculated to reduce the amount of edge crossings. There are many different approaches to this problem. *Sugiyama et al.* suggest a theoretical and a heuristic algorithm for this purpose, since this optimization problem is of a combinatorial nature and becomes quadratically more expensive in computanional time, as the amount of

[1] parallel horizontal lines

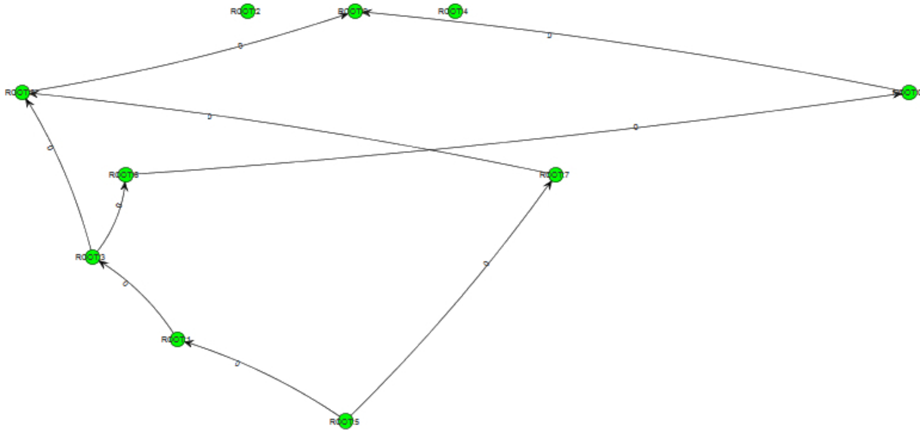


Figure 3.1: Result of the *JUNG DAG* Layout on the *Simple Example Graph*.

nodes increases. Which approach the layout calculation algorithm implemented in the *JUNG* framework takes, is, however, not stated in its documentation [11]. In the next step of the algorithm, horizontal node positions on each layer will be calculated to provide even spacing and a balance between all inputs and outputs of each node, making sure that each predecessor node is centered below all of its successor nodes.

The last step calculates vertical positions for each layer, since all nodes on the same layer also have the same vertical position. Figure 3.1 shows the result of the application of the *DAG* layout to a simple *graph*. This *graph* will be used as an example multiple times throughout this thesis. It will be referred to as the *Simple Example Graph* from here on.

Kamada and Kawai Layout

The *KK* layout is a force directed layout algorithm presented by *Kamada* and *Kawai* in [12]. It calculates positions of nodes by simulating a system of physical forces between these nodes.

To start off the algorithm, all nodes are placed on initial positions. These can be randomly assigned or the result of another layout algorithm, for example a circular arrangement. The *JUNG* framework, however, does not specify in its documentation [11], which kind of initial layout is used in its implementation.

In the model of the algorithm, each node is interpreted as an iron ring. Each of these rings is connected to all other rings via springs.
The algorithm then calculates the relaxed distance of each of these springs. To that end it determines an ideal edge length by dividing the size of the available drawing space through the maximum length of a path within the *graph* as follows:

$$L = L_0 / \max_{i \leq j} d_{ij}$$

Whereby L_0 is the the length of one of the sides of the coordinate space, d_{ij} is the graph theoretical distance between the vertices v_i and v_j and L is the desired length of a single edge.

This ideal edge length is then multiplied by the graph theoretical distance between each pair of nodes and the result is set as the relaxed length of the spring of the model that connects these two nodes.

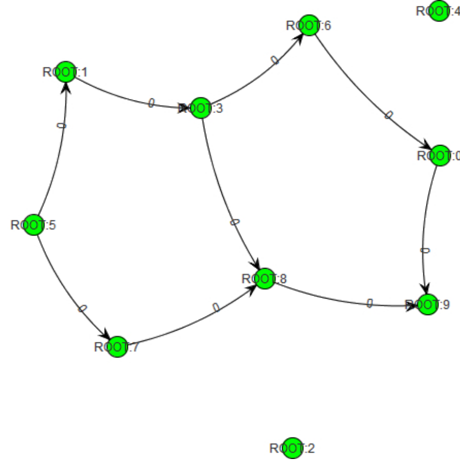


Figure 3.2: Result of the *JUNG KK* Layout on the *Simple Example Graph*.

The algorithm then iteratively moves nodes to minimize the amount of energy the simulated physical system of springs contains, whereby the energy is calculated by:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} \left(|p_i - p_j| - l_{ij} \right)^2$$

p_x is hereby the position of vertex v_x, l_{ij} the relaxed length of the spring connecting the two vertices v_i and v_j and $k_{ij} = K/d_{ij}^2$ with a constant K, which *KK* determined experimentally. Figure 3.2 shows the result of the *JUNG* implementation of the *KK* layout on the *Simple Example Graph*.

Fruchterman Reingold Layout

The *FR* algorithm uses a force directed approach as well, but as opposed to the *KK* layout, the algorithm proposed by *Fruchterman* and *Reingold* in [13] does not rely on graph theoretical distances, which require a lot of computational time to be determined.

The *FR* algorithm initializes all nodes on a set of coordinates that can be given by any positioning algorithm, as in the *KK* layout's algorithm as well. As is the case with the other algorithms, the *JUNG* documentation [11] does not reveal, which initial placement is used in their implementation of this algorithm.

After the initial placement, however, the *FR* algorithm differs from the *KK* method. Instead of relying on graph theoretical node distances, the *FR* layout computes repellent forces between all nodes, similar to the gravitational forces between atoms or molecules, and attracting forces between nodes which are directly connected via an edge. The displacement vector for a node v for the repulsive forces from node u is hereby calculated as:

$$v.disp = v.disp + \frac{d_{uv}}{|d_{uv}|} \cdot \frac{k^2}{|d_{uv}|}$$

whereby d_{uv} is the distance vector between the nodes u and v, and k is the optimal distance between the two nodes, which, in turn, is:

$$k = C \cdot \sqrt{\frac{W \cdot L}{|V|}}$$

In this case W is the width of the drawing space, L is the length of the drawing space and $|V|$ is the amount of nodes. C is a constant that has been determined experimentally by *FR*. $v.disp$ is initialized with the value zero.

The attractive force between the nodes v and u that are connected through the edge $e = (v, u)$ is calculated as:

$$v.disp = v.disp - \frac{d_{uv}}{|d_{uv}|} \cdot \frac{|d_{uv}|^2}{k}$$
$$u.disp = u.disp + \frac{d_{uv}}{|d_{uv}|} \cdot \frac{|d_{uv}|^2}{k}$$

FR also propose to use what they call *simulated annealing*, which describes a process in which a *heat* value gives the maximum distance a node can be moved in a single iteration of the algorithm. Each iteration, this *heat* value is decreased via a *cooling* function. This procedure is inspired by the slow cooling process of heated metals in the field of metallurgy, called *annealing*, which is also the origin of the name *simulated annealing*.

For the cooling process *FR* propose to split the algorithm into two phases. The first phase is called *quenching* and it rapidly decreases *heat* using a linear falloff. This provides a good initial placement of nodes after only a few iterations. After that, the algorithm is executed again with a constant very low *heat* value to make small adjustments. *FR* state in their paper that this approach shows better results in some subjective tests than a constant linear decline in temperature, while requiring less iterations.

To save computational time, *FR* decide that the repellent forces of nodes with a great distance between each other are negligibly small, which is why they propose to imagine a grid over the drawing space with a static size. Each node is only affected by nodes within the same or one of the neighboring cells of the grid. As a consequence, the amount of forces that need to be calculated is much smaller. Figure 3.3 shows the result of the *JUNG* implementation of the *FR* layout algorithm on the *Simple Example Graph*.

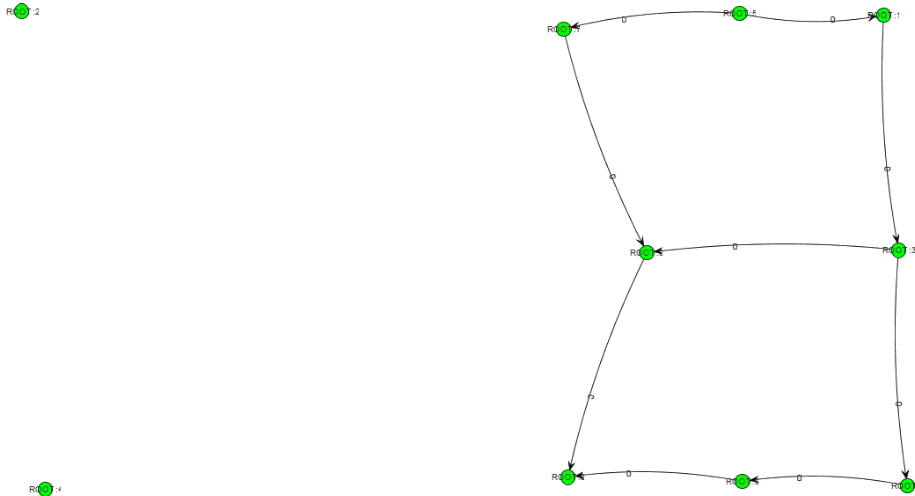


Figure 3.3: Result of the *JUNG FR* Layout on the *Simple Example Graph*.

Inverse Self Organizing Map Layout

The *ISOM* layout as first proposed by *Meyer* in [14] uses a neural network as model for the *graph* topology. The neural network is first set up with the same structure as the *graph* to be laid out, so that each vertex of the *graph* has a corresponding computational unit within the neural network. Then, a randomly generated, uniformly distributed set of coordinates of the coordinate space is given into the network. The amount of coordinate tuples in this set equals the amount of nodes. This same input is provided to the network multiple times. To train the neural network without the need for supervision, *competetive learning* is used.

Since neural networks usually have feature detection as their prime field of usage, they are designed so each unit of the network will accept all inputs with a certain strength. *Competetive learning* techniques select the unit with the strongest response to a given input as the ”winner” of a competition. As a consequence, the eagerness of the ”winner” unit to accept that same input again will be increased, while the eagerness of all other units in the network will be decreased. The further away a unit is from the ”winner” in the network topology, the further reduced is its eagerness to accept that same coordinate. Through the repeated input of the same coordinates, at some point all units will have selected a single coordinate out of the set to respond to, while the eagerness of all other units to respond to that coordinate tuple will have fallen to zero. Because the eagerness of neighboring units in the network is reduced less than that of non-neighboring units, neighbors are likely to accept neighboring coordinate tuples.

This mapping of neural network units is then applied to the original *graph*. Since the topology of the neural network is the same as that of the *graph* and each unit of the neural network corresponds to a node of the *graph*, the coordinate tuple that is accepted by a certain unit can just be applied to the corresponding node. Meanwhile the output of the neural network each iteration

is irrelevant and can be ignored.

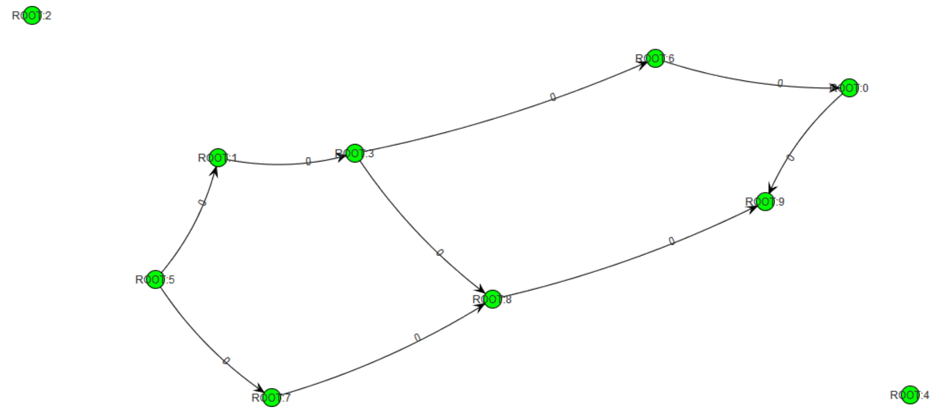


Figure 3.4: Result of the *JUNG ISOM* Layout on the *Simple Example Graph*.

This algorithm returns very similar results to the force directed approaches, but according to *Meyer's* tests, it requires less time, since no expensive force directed or graph theoretical calculations are needed. Figure 3.4 shows the result of the *JUNG* implementation of *Meyer's ISOM* layout on the *Simple Example Graph*.

4 Implementation

4.1 The LayoutGenerator Interface

For the creation of different layout generators to fit each application's own context, an interface is suggested, that describes the minimum of necessary functionality that such a layout generator should provide.

To generate a layout on a *graph* within the *VWorkflows* library, a layout generator must accept a `VFlowModel` object and calculate a layout for it. To provide this functionality within the `LayoutGenerator` interface, a getter- and a setter-method were declared to handle the transfer of the `VFlowModel`. Furthermore, a method to harbor the layout calculation algorithm was declared. The minimalist `LayoutGenerator` interface, therefore, consists of the three methods:

```
public void setWorkflow(VFlowModel pworkflow)
public VFlowModel getWorkflow()
public void generateLayout()
```

Additionally to these methods, there are three other features each layout generator should provide.

Recursive Execution

Since each `VNode` within the `VFlowModel` object can also contain a *subflow*, the layout generator must be able to apply its layout algorithm to each of these *subflows* recursively. But this recursive execution is not always desirable, because it can become computationally very expensive for flow *graphs* with a large *depth*. Therefore, a boolean parameter by the name of `recursive` was introduced that can be manipulated via its own getter- and setter-methods. If this parameter equals *true*, the algorithm shall be applied to each *subflow* of the `VFlowModel`.

Automatic Scaling of Nodes

As mentioned earlier, each `VNode` can house a *subflow*. To make these *subflows* clearly readable, the `VNode` object must be of an appropriate size. For this reason, each layout generator should include the functionality to automatically scale nodes according to their contents. This feature may, however, not always be desired, either — for example, if the drawing space in the particular application is rather limited. That is why this component should also be able to be switched on or off via a boolean parameter.

Debugging Output

To make the `VWorkflows` framework as friendly to other developers as possible, each layout generator should always supply some sort of debugging functionality. To toggle the display of additional debugging output, another boolean parameter was introduced.

Including all three of these additional features, the `LayoutGenerator` interface now consists of the methods:

```
public void setWorkflow(VFlowMode pworkflow)
public VFlowModel getWorkflow()
public void setRecursive(boolean precursive)
public boolean getRecursive()
public void setAutoscaleNodes(boolean pautoscaleNodes)
public boolean getAutoscaleNodes()
public void setDebug(boolean pdebug)
public boolean getDebug()
public void generateLayout()
```

The `generateLayout()` method runs all additional methods depending on their parameters. The layout is directly applied to the given `VFlowModel` object, so a return type is not necessary. Within the method, the check for the parameters' status must be implemented and reacted upon accordingly.

4.2 The Naive Layout Generator

4.2.1 The Idea

The idea behind the creation of the `LayoutGeneratorNaive` class was to implement a simple algorithm without prior knowledge. The resulting algorithm is a simplified implementation of the *DAG* layout described in (3.2.3).

4.2.2 The Algorithm

The basic algorithm of the naive layout generator consists of three steps.

1. Separation of the *Graph* into Layers
2. Calculation of Vertical Coordinates of Nodes within those Layers
3. Calculation of Horizontal Coordinates of the Layers

These three steps will now be explained in further detail.

Separation of the Graph into Layers

In this step, each node is assigned a layer index. The layer indices for each node are initialized as zero. The algorithm then iterates over all edges. For each edge $e = (v_i, v_j)$ the layer index $l(v_j)$ of node v_j is set to $l(v_i) + 1$. After the iteration terminates, all nodes, which still have the layer index $l(v_x) = 0$, are locked via a boolean parameter. The layer indices of locked nodes will not be changed going forward.

This process will be repeated and with each cycle the layer index to be locked will be increased by one, until all nodes are locked.

Listing 4.1: Step 1 of the Naive Algorithm.

```
1  lockable = 0
2  while not allLocked() {
3    for i in connections {
4      if not locked(i.getSecond())
5          and l(i.getSecond()) < (l(i.getFirst()) + 1) {
6        l(i.getSecond()) = l(i.getFirst()) + 1
7      }
8    }
9    for i in nodes {
10     if l(i) == lockable {
11       lock(i)
12     }
13   }
14   lockable += 1
15 }
```

Listing 4.1 shows a pseudo code example of the implementation for this step of the algorithm. The function in the class `LayoutGeneratorNaive` that implements this code in Java is called `createLayering()`.

Calculation of Vertical Coordinates

In the second step of the algorithm, the vertical coordinate for each node is calculated. To that end, the algorithm first constructs a list of nodes for each layer out of the layer index mapping created in the first step. Next, the algorithm iterates over the nodes of each layer and sets their position. The position is determined by the position of the node previously placed, the `height` of that node and an additional parameter to determine the distance between nodes.

This additional parameter is called `scaling`. It is a double precision floating point value. If the value is negative, its absolute value is used as a relative factor multiplied with the `heigth` of the previously placed node. If the `scaling` parameter holds a positive value, it is added to the position.

Listing 4.2: Step 2 of the Naive Algorithm

```
1  for l in layers {
2    pos = 0
3    for n in l {
4      n.setY(pos)
5      if scaling < 0 {
6        pos += n.getHeight() · scaling · (−1)
7      } else {
8        pos += n.getHeight() + scaling
9      }
10   }
11 }
```

Listing 4.2 describes the positioning algorithm of the second step in pseudo code. This code is implemented in the function `calculateVerticalPositions()` within the `LayoutGeneratorNaive` class.

Calculation of Horizontal Coordinates

The last step of the basic naive algorithm calculates horizontal coordinates for each layer. For this purpose, the algorithm iterates over all nodes of each layer and sets their position. At the same time, the maximum width of all nodes on that layer is determined. The position for the nodes on the next layer is increased by the maximum width of all nodes on the current layer modified by the `scaling` parameter, analogous to the second step.

Listing 4.3: Step 3 of the Naive Algorithm

```
1  pos = 0
2  for l in layers {
3    maxwidth = −∞
4    for n in l {
5      n.setX(pos)
6      if n.getWidth() > maxwidth {
7        maxwidth = n.getWidth()
8      }
9    }
10   if scaling < 0 {
11     pos += maxwidth · scaling · (−1)
12   } else {
13     pos += maxwidth + scaling
14   }
15 }
```

Listing 4.3 shows a pseudo code implementation of the positioning algorithm in this step. A Java implementation of this algorithm can be found in the function `calculateHorizontalPositions()` in the `LayoutGeneratorNaive` class.

4.2.3 Additional Features

The basic algorithm alone is not enough to satisfy the expectations of a general layout generator. This requires some additional features to be implemented, which will be detailed in the following.

Removal of Cycles

In step one of the algorithm, nodes are mapped to layers, so that each edge $e = (v_i, v_j)$ points from a layer with a lower index to a layer with a higher index, or mathematically speaking: so that $l(v_i) < l(v_j)$. If the *graph* contains cycles, such a mapping is not possible. This means, an additional step must be introduced to remove edges from the model *graph*, so all cycles will be eliminated.

The cycle removal algorithm performs a depth first search on the model *graph*. Each node that is reached is marked. If a node is already marked, it is skipped. Also the path from the root of the search tree to the current node is saved. If the current node is already contained within the path, the most recently traversed edge is removed.

Listing 4.4: The Cycle Removal Algorithm

```
1  removeCycle(currNode, path) {
2    if not checked(currNode) {
3      path.add(currNode)
4      checked(currNode) = true
5      succs = currNode.getSuccessors()
6      for s in succs {
7        if path.contains(s) {
8          removeAllConnections(currNode, s)
9        } else {
10         removeCycle(s, path)
11       }
12     }
13     path.remove(currNode)
14   }
15 }
```

Listing 4.4 shows a simplified pseudo code variant of the cycle removal algorithm. The algorithm is implemented by the method `remCycR()`. It is executed with each node as root for the search tree in the `removeCycles()` method.

Recursive Execution

According to the `LayoutGenerator` interface defined in (4.1), all layout generators must be able to apply their algorithm to each *subflow* of the *graph*. This is implemented in the `runSubflows()` method, which simply iterates over all `VNode` objects of the given `VFlowModel` and checks, which nodes are an instance of the `VFlowModel` interface as well. An instance of the `LayoutGeneratorNaive` class is then created and all parameters of the parent generator are carried over to the child generator. Afterwards, the child generator is supplied with the *subflows* and generates a layout for them.

Automatic Scaling of Nodes

In the definition of the `LayoutGenerator` interface it was required that each layout generator shall supply the user with the ability to automatically scale `VNode` objects according to their contents.

This functionality is implemented in the `autoscaleNodes()` method. The algorithm iterates over all nodes of the `VFlowModel` and determines, which nodes hold *subflows*, by checking each `VNode` object if it is an instance of the `VFlowModel` interface.

For each found *subflow* the algorithm then iterates over the *subflow's* nodes and determines the maximum and minimum coordinates on both axes. The dimensions of the corresponding *subflow node* are then calculated by the following formulas:

$$Height = \frac{y_{max} - y_{min}}{n} \cdot \texttt{subflowscale}$$
$$Width = \frac{x_{max} - x_{min}}{n} \cdot \texttt{subflowscale}$$

Whereby n is the amount of nodes within the *subflow* and `subflowscale` is a double precision floating point parameter.

4.2.4 Customizability

To make this layout applicable to as many situations as possible, it must be provided in a highly customizable state. That is why each step of the algorithm and each of the additional features can be toggled on or off via a boolean parameter and numerical parameters can be changed as well. A full list of all fields and methods of the class can be found in (A.1).

During the initialization of the generator object, all parameters are set to their default values. These values are:

```
recursive: true
autoscaleNodes: true
graphmode: 0
launchRemoveCycles: true
launchCreateLayering: true
launchCalculateVerticalPositions: true
launchCalculateHorizontalPositions: true
scaling: -1.5
subflowscale: 2
```

4.2.5 The Model

The model of the `LayoutGeneratorNaive` class consists of the following fields:

`nodes` An array of `VNode` objects, representing all nodes.

`workflow` A `VFlowModel` object containing the overall flow.

`nodecount` An int value representing the number of nodes and the length of the `nodes` array.

`connectionList` An array of `Pairs` of integers representing the `Connections` within the `workflow`, whereby the first integer contains the index of the *sender* node and the second integer the index of the *receiver* node within the `nodes` array.

`conncount` An int value representing the number of `Connections` within the *workflow*.

`cycle` A boolean value that indicates whether the *graph* contains cycles.

`layering` An array of int values of the same length as the `nodes` array, assigning a layer index to each node.

`layercount` An int value representing the amount of layers in the model.

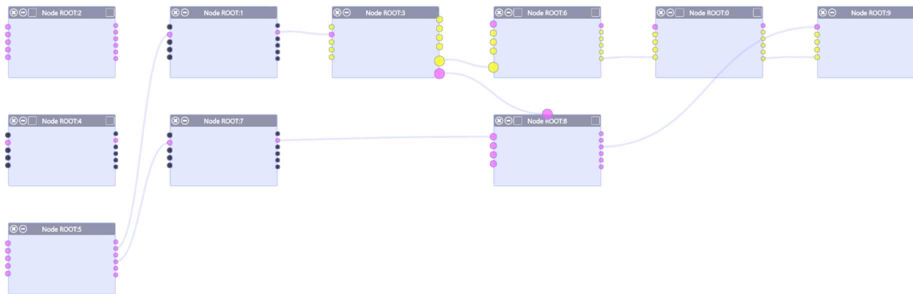


Figure 4.1: Result of the Final Naive Algorithm on the *Simple Example Graph*.

The `workflow` and `nodes` fields can be used as input. Which of the two is used as input on execution is determined by the `graphmode` parameter. The field not used as input as well as the `nodecount`, `connectionList` and `conncount` are initialized in the `setUp()` method. The `cycle` field is set by the `checkCycles()` function, which is called from the `setUp()` method. The `layering` and `layercount` fields are populated during the first step of the algorithm in the `createLayering()` method.
Figure 4.1 shows the result of the final naive algorithm on the *Simple Example Graph*.

4.3 The Smart Layout Generator

4.3.1 The Idea

The idea behind the `LayoutGeneratorSmart` class is to include already existing layout algorithms from the popular *JUNG* framework into *VWorkflows*. When examining the list of layouts provided by the *JUNG* library, four of them seem to be applicable to the problem at hand and seem to deliver useful results. The four layouts in question are:

- *DAG* Layout
- *KK* Layout
- *FR* Layout
- *ISOM* Layout

For descriptions of the algorithms employed by these layouts, see (3.2.3).

A quick comparison of a few results on simple *graphs* gives some insight into the quality of these layouts. The first thing to note is that all four *JUNG* implementations are nondeterministic, which causes the quality of their results to vary greatly. Furthermore, the *KK*, *FR* and *ISOM* algorithms produce similar

layouts, while the *DAG* layout's results are very different. This comes to no surprise when considering the different approach the *DAG* layout takes in comparison to the others. Unfortunately, the results of the *JUNG* implementation of the *DAG* layout are not only very different from the results of the other three algorithms, but also worse.

The vertices on different layers are not well aligned with each other and the positioning of the nodes within each layer leads to a lot of unnecessary edge crossings. Figure 3.1 shows one such result.

These problems seem not to stem from the algorithm itself but from the *JUNG* implementation of the algorithm. The documentation of the *JUNG* framework [11] is, however, very sparse and the algorithm is not customizable enough to integrate another implementation of the second step of the algorithm to fix these problems. That is why the other three algorithms were mainly used in the creation of the `LayoutGeneratorSmart` class.

Since all three of these algorithms provide similar results and according to *Meyer* [14] the *ISOM* layout is less computationally expensive, the decision was made to focus mainly on the utilization of this algorithm.

4.3.2 The Algorithm

The basic algorithm implemented within the `LayoutGeneratorSmart` class consists of four steps.

1. Application of the *JUNG* Implemented Layout
2. Rotation of the *Graph* around its Center Point
3. Movement of all Successor Nodes *behind* their Predecessors
4. Mutual Repulsion of all Nodes

These steps will be further explained in the following.

Application of the JUNG Implemented Layout

In step one of the algorithm, the *JUNG* implemented layout indicated by the `layoutSelector` parameter is applied to the given `VFlowModel`. This parameter is of type int and is mapped to the four different *JUNG* layouts available as follows:

0 - *ISOM* layout

1 - *FR* layout

2 - *KK* layout

3 - *DAG* layout

Regardless of which layout is used, the `Layout` object must be provided with the size of the drawing space. To determine the size of the initial drawing space, the longest path within the *graph* is searched and the width of all nodes on

this path summed up. This cumulative width is then divided by the `scaling` parameter, which is a double precision floating point value, and a constant factor, that was experimentally determined to be 2.

The height of the drawing space is calculated by dividing the width by the `aspectratio` parameter, which is a double precision floating point value as well.

After the drawing space size has been supplied to the `Layout` object, the layout can be applied to the *graph*. This is achieved by iterating through all nodes of the `VFlowModel` and calling the `transform()` method of the `Layout` object with each `VNode` as argument. The return value of the `transform()` method is a `Point2D` object that holds the new coordinates for the respective `VNode`.

This step of the algorithm is implemented in the method `stepLayoutApply()` and can be seen as pseudo code in Listing 4.5.

Listing 4.5: Step 1 of the Smart Algorithm

```
1 maxpath = findMaxPathWidth() / (scaling · 2)
2 height = maxpath / aspectratio
3 layout.setSize(new Dimension(maxpath, height))
4 for n in nodes {
5   coords = layout.transform(n)
6   n.setX(coords.getX())
7   n.setY(coords.getY())
8 }
```

The method `findMaxPathWidth()` of the `LayoutGeneratorSmart` class implements the algorithm to find the width of the longest path within the *graph* and is shown in Listing 4.6.

Listing 4.6: Width of the Longest Path

```
1  fifo = new LinkedList()
2  for n in nodes {
3    if n.getSuccessorCount() == 0 {
4      maxPathFollowing[n] = 0
5      fifo.add(n)
6    }
7  }
8  # find the length of the longest path following each node
9  while not fifo.isEmpty() {
10   currNode = fifo.removeFirst()
11   succs = currNode.getSuccessors()
12   for s in succs {
13     tempFollowing = maxPathFollowing[s] + 1
14     if tempFollowing > maxPathFollowing[currNode] {
15       maxPathFollowing[currNode] = tempFollowing
16     }
17   }
18   pre = currNode.getPredecessors()
19   for p in pre {
20     fifo.add(p)
21   }
22 }
23 # find origin node of the longest path
24 maxPath = −∞
25 maxPathIndex = 0
26 for n in nodes {
27   if maxPathFollowing[n] > maxPath {
28     maxPath = maxPathFollowing[n]
29     maxPathIndex = n
30   }
31 }
32 # find longest path and its width
33 maxPathWidth = nodes[maxPathIndex].getWidth()
34 succs = nodes[maxPathIndex].getSuccessors()
35 maxPath−−
36 for s in succs {
37   if maxPathFollowing[s] == maxPath {
38     maxPathWidth += s.getWidth()
39     maxPath−−
40     succs = s.getSuccessors()
41   }
42 }
```

Rotation of the Graph around its Center Point

Since the algorithm should provide a constant *general flow direction* and the *KK*, *FR* and *ISOM* layouts are designed to be applied to undirected *graphs*, this direction must be established thereafter.

To guarantee this direction, the *graph* is rotated around its center point, so that the average direction of all edges equals the desired *general flow direction*. The algorithm for this step first determines the center point of the *graph* using the method `getGraphCenter()`. Then, the cumulative edge vector is determined by summing up all edge vectors. After that, the angle between the x-axis and the cumulative edge vector is calculated and the *graph* rotated around its center point by the negative of this angle.
The new average edge direction is now 0, which means the cumulative edge vector points in the direction of the x-axis.

Now, the *graph* is again rotated so that the new cumulative edge vector points in the desired *general flow direction*, which is given by the `direction` parameter. For the rotation a simple rotational matrix of the form

$$\begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix}$$

is used, so that the final rotated coordinates can be calculated as:

$$\begin{aligned} x' &= x \cdot \cos(\phi) + y \cdot \sin(\phi) \\ y' &= y \cdot \cos(\phi) - x \cdot \sin(\phi) \end{aligned}$$

whereby ϕ is the angle by which the coordinates shall be rotated.

The rotation of the *graph* is implemented in the method `stepRotate()` of the `LayoutGeneratorSmart` class.

Movement of all Successor Nodes behind their Predecessors

The third step of the algorithm is also used to guarantee the constant *general flow direction*. The idea behind this step is that if all nodes are moved *behind* their predecessors, by the definition in (2.2), all edges of the *graph* will point in the *general flow direction*.

To achieve this, the algorithm adds all *origin* nodes to a queue and iterates through this queue.
Of the current node all successors are added to the queue as well. As a consequence, the iteration through all nodes follows the structure of the *graph*. Furthermore, for each predecessor of the current node, a potential displacement of the current node is calculated.

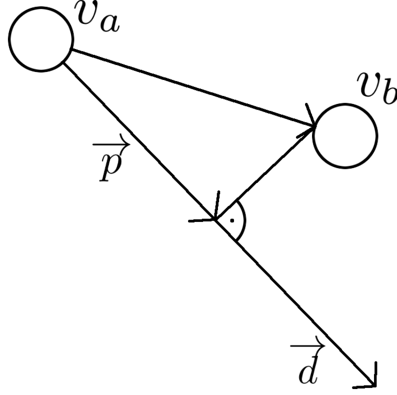


Figure 4.2: Projection of a Distance Vector onto the *General Flow Direction*.

To calculate this displacement, the algorithm first calculates the desired distance between both nodes. This desired distance is calculated as the length of one half of the diagonal of the predecessor node, multiplied by the `scaling` parameter. When the predecessor node is declared as v_a, the current node is v_b and w_x and h_x represent the width and height of node v_x, the desired distance d_{des} between both nodes equates to:

$$d_{des} = \frac{\sqrt{w_a^2 + h_a^2}}{2} \cdot \texttt{scaling}$$

The `scaling` parameter is a double precision floating point parameter analogous to (4.2.2).

After the desired distance is found, the current distance is determined. Since this step of the algorithm, however, tries to adhere to the *general flow direction*, the length of the projection of the distance vector onto the *general flow direction* vector is used instead.

Figure 4.2 shows the projection $\overrightarrow{p}$ of the distance vector between the predecessor node v_a and the current node v_b onto the direction vector $\overrightarrow{d}$. According to [15], the vector of the projection is calculated as:

$$\overrightarrow{p} = \frac{(\overrightarrow{v_b} - \overrightarrow{v_a}) \cdot \overrightarrow{d}}{|\overrightarrow{d}|^2} \cdot \overrightarrow{d} = \frac{(\overrightarrow{v_b} - \overrightarrow{v_a}) \cdot \overrightarrow{d}}{\overrightarrow{d} \cdot \overrightarrow{d}} \cdot \overrightarrow{d}$$

Separated into the x and y coordinates this results in:

$$x_p = \frac{x_d \cdot (x_b - x_a) + y_d \cdot (y_b - y_a)}{x_d^2 + y_d^2} \cdot x_d$$
$$y_p = \frac{x_d \cdot (x_b - x_a) + y_d \cdot (y_b - y_a)}{x_d^2 + y_d^2} \cdot x_d$$

The length of the projection is then compared with the desired distance. Furthermore, a direction test is performed by adding the direction vector to the

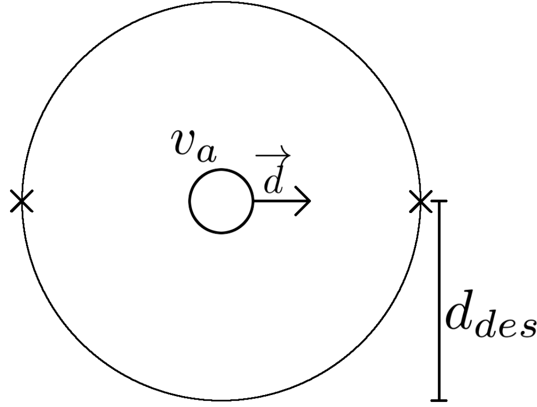


Figure 4.3: Semantics behind the Results for the Displacement Factor ϕ.

projection. If the length of the projection increases through this addition, the projection points in the *general flow direction*, if the length decreases it points against the *general flow direction*.

If the projection is shorter than the desired distance or points against the *general flow direction*, a displacement for the current node is calculated. To move the current node, the direction vector is added to the location of the current node until the length of the projection is as large as the desired distance. To that end, the formula

$$d_{des} = |\overrightarrow{p} + \phi \cdot \overrightarrow{d}| \tag{4.1}$$

must be solved for ϕ.
After calculating the length of the vector as $|\overrightarrow{v}| = \sqrt{x_v^2 + y_v^2}$, the equation (4.1) is equivalent to

$$0 = \phi^2 + \phi \cdot \frac{2x_d x_p + 2y_d y_p}{x_d^2 + y_d^2} + \frac{x_p^2 + y_p^2 - d_{des}^2}{x_d^2 + y_d^2}$$

which can be solved using the p-q-formula.

$$\phi_{1/2} = -\frac{x_d x_p + y_d y_p}{x_d^2 + y_d^2} \pm \sqrt{\left(\frac{x_d x_p + y_d y_p}{x_d^2 + y_d^2}\right)^2 - \frac{x_p^2 + y_p^2 - d_{des}^2}{x_d^2 + y_d^2}}$$

Figure 4.3 shows the semantics behind the two results for ϕ. The two points marked with x in the figure represent the two locations resulting from the equation $\overrightarrow{p} + \phi \cdot \overrightarrow{d}$ with ϕ_1 and ϕ_2 respectively substituted for ϕ. One of these locations is always *in front* while the other is always *behind* v_a. The larger of ϕ_1 and ϕ_2 will always result in the coordinate *behind* v_a, which is the desired location. Since the result of a square root is always positive, ϕ_1 will always be larger than ϕ_2. Therefore, ϕ_2 is never needed, which is why only ϕ_1 is calculated and used in the algorithm.

Listing 4.7: Step 3 of the Smart Algorithm

```
1  fifo  = new LinkedList()
2  fifo .add(origin)
3  while not fifo.isEmpty() {
4    currNode = fifo.removeFirst()
5    succs = currNode.getSuccessors()
6     fifo .add(succs)
7    pre = currNode.getPredecessors()
8    for p in pre {
9      desDist = sqrt(pow(p.getWidth(), 2) + pow(p.getHeight(), 2))
10            · scaling  / 2
11     # predecessor location
12     xa = p.getX()
13     ya = p.getY()
14     # current node location
15     xb = currNode.getX()
16     yb = currNode.getY()
17     # direction  vector
18     xd = cos(direction)
19     yd = sin(direction)
20     # projection vector
21     xp = calcProjX()
22     yb = calcProjY()
23     # projection length
24     plen  = sqrt(pow(xp, 2) + pow(yp, 2))
25     ptest  = sqrt(pow(xp+xd, 2) + pow(yp+yd, 2))
26     if  plen  < desDist or ptest < plen {
27       phi = calcPhi1()
28       xb += phi · xd
29       yd += phi · yd
30       currNode.setX(xb)
31       currNode.setY(yb)
32     }
33   }
34 }
```

Listing 4.7 shows pseudo code of the algorithm of this step. The complete implementation of this pseudo code in Java can be found in the method `stepPushBack()` of the `LayoutGeneratorSmart` class.

Mutual Repulsion of all Nodes

The last step of the algorithm applies repellent forces between each pair of nodes to remove overlaps between them. This step is heavily inspired by the *FR* algorithm, since it applies repellent forces to each node. There are, however, some differences.

The algorithm iterates over each pair of nodes. Each iteration a desired distance between the two nodes is calculated by the method `getDesiredNodeDist()`. This desired node distance depends on the sizes of both nodes and the direction of the distance vector between them. The desired distance is the minimum distance necessary between the two nodes, to avoid overlaps. The algorithm also calculates the real distance between the two nodes by calculating the length of

the distance vector between them using the `getRealNodeDist()` function.

If the real distance is smaller than the desired distance, a displacement of one of the nodes is calculated to reduce the difference between the two distances to zero. This is done over multiple iterations, until either the maximum amount of iterations, given by the `maxiterations` parameter, has been reached or no nodes have been moved in the last iteration.

Listing 4.8: Step 4 of the Smart Algorithm

```
1  change = true
2  for i in range(0, maxiterations) {
3    if not change {
4      return
5    }
6    change = false
7    for a in nodes {
8      for b in nodes {
9        realDist = getRealNodeDist(a, b)
10       desDist = getDesiredNodeDist(a, b)
11       if realDist < desDist {
12         change = true
13         # node a location
14         xa = a.getX()
15         ya = a.getY()
16         # node b location
17         xb = b.getX()
18         yb = b.getY()
19         # distance vector
20         xd = xb − xa
21         yd = yb − ya
22         # displacement factor
23         phi = (desDist − realDist) / sqrt(pow(xd, 2) + pow(yd, 2))
24         # new positions
25         xb += xd · phi
26         yb += yd · phi
27         b.setX(xb)
28         b.setY(yb)
29       }
30     }
31   }
32 }
```

Listing 4.8 shows a pseudo code implementation of step four of the algorithm, which is implemented in the `forcePush()` method.

The function `getDesiredNodeDist()`, which holds the calculation of the desired node distance, is shown in pseudo code in Listing 4.9.

Listing 4.9: Calculation of the Desired Node Distance

```
1  # node a
2  xa = a.getX()
3  ya = a.getY()
4  wa = a.getWidth()
5  ha = a.getHeight()
6  # node b
7  xb = b.getX()
8  yb = b.getY()
9  wb = b.getWidth()
10 hb = b.getHeight()
11 # distance vector
12 xd = xb − xa
13 yd = yb − ya
14 # is the direction of d closer to vertical or horizontal
15 xcomp = abs(2 · xd / wa)
16 ycomp = abs(2 · yd / ha)
17 if xcomp >= ycomp {
18   # calculate distance between center and edge of node
19   # horizontal case:
20   fa = wa / (cos(atan(yd / xd)) · 2)
21 } else {
22   # vertical case:
23   fa = ha / (cos(atan(xd / yd)) · 2)
24 }
25 # repeat for the second node
26 xcomp = abs(2 · xd / wb)
27 ycomp = abs(2 · yd / hb)
28 if xcomp >= ycomp {
29   fb = wb / (cos(atan(yd / xd)) · 2)
30 } else {
31   fb = hb / (cos(atan(xd / yd)) · 2)
32 }
33 return (fa + fb) · scaling
```

The evolution of the layout through the execution of this algorithm can be seen in Figure 4.4.

4.3.3 Additional Features

Analogous to the *naive* layout generator, there are some features which have to be provided additionally to the generation of a simple layout.

Removal of Cycles

Some steps of the algorithm do not terminate on cyclic *graphs*. This includes the *JUNG* implemented *DAG* layout, the `findMaxPathWidth()` method and the entirety of step three of the algorithm. As a consequence, an additional step to remove cycles from the *graph* must be implemented to run before the rest of

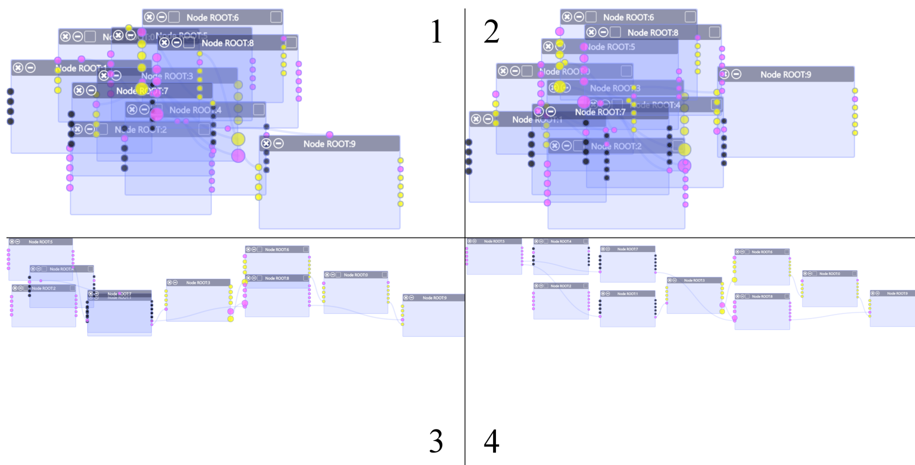


Figure 4.4: Results of the Smart Algorithm after each Step.

the algorithm. This step functions analogous to the cycle removal algorithm in the naive layout generator described in (4.2.3).

Recursive Execution

Likewise analogous to the naive implementation is the requirement to be able to execute the algorithm on all *subflows* of the *graph*. The implementation in the `LayoutGeneratorSmart` class also works the same way as in the naive implementation described in (4.2.3).

Automatic Scaling of Nodes

The smart layout generator also has to have the functionality to automatically scale *subflow nodes* according to the size of the *subflow* they contain. This implementation uses the same algorithm as described in (4.2.3) as well.

Alignment of Nodes

A *graph* layout is especially aesthetically pleasing if neighboring nodes only differ in one of their two coordinates and are aligned on the other. While the naive generator achieves a horizontal alignment of nodes on the same layer automatically, this can not be said about the smart approach. To achieve this with the smart layout generator, an additional step must be included into the algorithm. This step is implemented in the `alignNodes()` method of the layout generator.

There are two ways to align nodes with each other:

1. Align nodes pairwise with each other by setting them on the same coordinate if their difference is below a certain threshold.

2. Lay out all nodes on a global grid of a certain size.

Both of these approaches are implemented within the same method. The user can switch between them by changing the double precision `alignmentThreshold`

parameter. If this parameter contains a positive value, that value is used as size for the global grid to align all nodes on. Is the value of the parameter negative, the absolute value of the parameter will be used as threshold under which the nodes will be aligned pairwise.

This step can, however, create overlaps between nodes, which should have already been eliminated by the `forcePush()` method. To combat this behavior, the alignment process is executed after the first half of the `forcePush()` iterations have already completed. The other half of the iterations is run after the alignment step to remove the possibly newly created overlaps.

Displacement of Nodes in Identical Positions

Step four of the algorithm can not be performed on pairs of nodes in identical positions. If two nodes have the same coordinates, the length of the distance vector between them is zero. Since $\sqrt{x_d^2 + y_d^2}$ is the length of the distance vector, the calculation

```
phi = (desDist - realDist) / sqrt(pow(xd, 2) + pow(yd, 2))
```

in the algorithm of step four, shown in Listing 4.8 line 23, would have to divide by zero when trying to process two nodes in the same location. Therefore, a step must be introduced to slightly displace nodes, which have the same coordinates. This step is implemented in the `displaceIdents()` method.

Separation of Disjunct Subgraphs

If the *graph* contains multiple disjunct subgraphs, these subgraphs can overlap and influence each other in step four of the basic algorithm. This can result in a very unsatisfactory layout, which can be seen in Figure 4.4. The nodes *ROOT:4* and *ROOT:2* are not connected to the rest of the *graph*, but are placed between *ROOT:5*[1] and its successor nodes, and thereby causing *ROOT:5's* edges to cut through them.

This can be avoided by calculating layouts for disjunct subgraphs separately and laying them out over each other afterward. This is implemented in the method `separateDisjunctGraphs()`.

The method first assigns a number to each node that is initialized as -1. The algorithm then iterates through the *graph*, starting at an *origin* node, and assigns the number 0 to every node it can reach through edge traversal. Next, it checks which nodes are still assigned the number -1 and chooses one of those to start again, but increments the number it assigns by one. This continues until all nodes are assigned a number that is not -1.

If the algorithm reaches a node already marked with a number that is not -1, all nodes that have been marked with the current number will be changed to the number of that node.

The result of this algorithm is a mapping of each node to a number that represents the id of the subgraph it belongs to. The algorithm then creates a `LayoutGeneratorSmart` object with the same parameters as the parent generator and lets this child generator calculate a layout for each subgraph. After

[1] *ROOT:5* is the *origin* node of the *graph*.

that, the bounding box of each resulting layout is calculated and the subgraphs are arranged over each other ascending in ids from top to bottom.

Placement of Origin Nodes

Though the rotation of the *graph* does guarantee the *general flow direction*, it does not guarantee that all *origin* nodes are placed *in front* of all other nodes in the *graph*, which would be preferable in a good layout.

Since the most commonly desired *general flow direction* is from left to right, the method `stepOrigin()` was included. It automatically places all *origin* nodes at the leftmost edge of the drawing space.

However, the combination of the methods `separateDisjunctGraphs()` and `stepPushBack()` achieves the same goal for all desired *general flow directions*, which is why the `stepOrigin()` method is deactivated by default.

Separation of Edge Types

One special feature of program flow *graphs* is that there are multiple different types of edges. In the *VWorkflows* framework, these include dataflow edges, controlflow edges and event edges. Furthermore, additional or completely new sets of edge types can be implemented. Through the addition of edge types to a visual programming language, the total number of necessary edges within a *graph* increases. With more edges in the *graph* it is, however, more difficult to find a pleasing layout that minimizes edge crossings and guarantees the *general flow direction* for all edges.

For this reason, the edge types should be treated differently and assigned priorities. This way, the *general flow direction* can be satisfied for edge types of the highest priority. For the edge types of further decreasing priorities, some edges might violate the requirement of the *general flow direction* to improve the overall arrangement of the *graph*.

A prototype implementation for this feature already exists within the method `separateEdgeTypes()`. This method does, however, not achieve the desired results in its current version. To make the integration of a working implementation of this feature in the future easier, the prototype has not been removed, but is toggled off by default.

4.3.4 Customizability

Analogous to the `LayoutGeneratorNaive` class, all steps of this algorithm and all of the additional features can be turned on or off via boolean parameters and most numerical parameters can be changed as well. This ensures that the user can tweak the generator to their liking. Additionally, the applied *JUNG* layout can be chosen parametrically out of the four different layouts described in (3.2.3). A full list of all fields and methods of the class can be found in (A.2).

During the initialization of the generator object, all parameters are set to the following default values:

```
recursive: true
autoscaleNodes: true
layoutSelector: true
aspectratio: 1.778[2]
graphmode: 0
launchRemoveCycles: true
launchSeparateDisjunctGraphs: true
launchSeparateEdgeTypes: false
launchJungLayout: true
launchRotate: true
launchOrigin: false
launchPushBach: true
launchDisplaceIdents: true
launchForcePush: true
launchAlignNodes: true
maxiterations: 500
scaling: 1.2
subflowscale: 2
direction: 0
alginmentTreshold: (scaling - 1) · (-1)
```

4.3.5 The Model

The model of the `LayoutGeneratorSmart` class consists of the following fields:

`nodes` An array of `VNode` objects, representing all nodes.

`workflow` A `VFlowModel` object containing the overall flow.

`nodecount` An int value representing the number of nodes and the length of the `nodes` array.

`jgraph` A `DirectedGraph` object containing the model *graph*, that corresponds to the *graph* within the `worfklow`.

`conncount` An int value representing the number of `Connections` within the *workflow*.

`origin` An array of `Pairs` containing two integers. The first integer represents the index of the *origin* node in the `nodes` array and the second integer represents the number of that node's successors.

`graphcenter` A `Point2D` object representing the coordinates of the center point of the *graph*.

`cycle` A boolean value that indicates whether the *graph* contains cycles.

[2]Representing an aspect ratio of 16:9, which is a common display resolution.

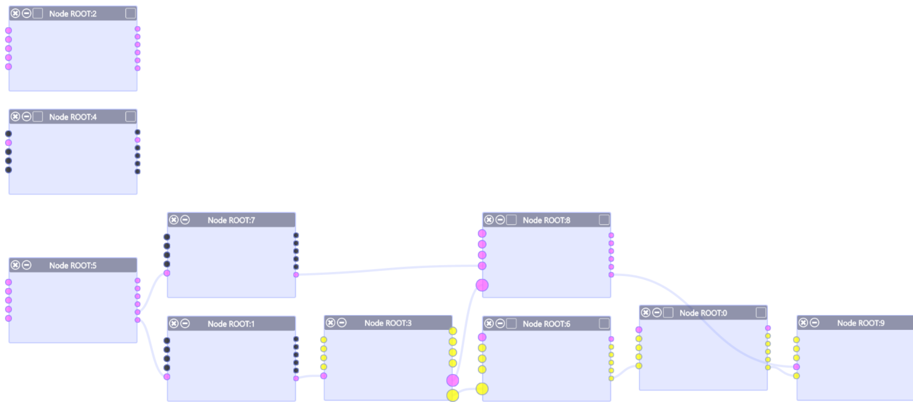


Figure 4.5: Result of the Final Smart Algorithm on the *Simple Example Graph*.

The fields `workflow`, `jgraph` and `nodes` can all be used as input for the generator. Which of these fields is actually used as input is indicated by the `graphmode` parameter. The other two of these fields as well as the `nodecount` and `conncount` fields are initialized during the `allNodesSetUp()` method. The `origin` array is populated in the `getOrigin()` method, which is called within the `allNodesSetUp()` method. The `cycle` field is set by the `checkCycles()` method, which is called by the `allNodesSetUp()` method as well. The `graphcenter` field, on the other hand, is initialized during the `getGraphCenter()` method, which is run during the execution of the `generateLayout()` function.
Figure 4.5 shows the result of the final algorithm with default parameters on the *Simple Example Graph*.

5 Evaluation

Considering the difficulties in formulating the problem and the high amount of subjectivity when evaluating *graph* layouts explored in (2), this evaluation gives no guarantees for correctness. It tries to give a good estimation of the quality of the results of both described implementations, but personal experience may vary.

5.1 Methodology

The results of both implementations are evaluated on the basis of the six important criteria of an aesthetically pleasing layout discussed in (2).

1. The layout must contain a minimal amount of overlap between vertices.
2. The layout must abide by a constant *general flow direction*.
3. The layout must contain a minimal number of edge crossings.
4. The layout must scale vertices according to their contents.
5. The layout must represent symmetric structures within the graph symmetrically.
6. The layout must uniformly distribute vertices in the available space.

These attributes can, however, stand in conflict to each other.

Figure 5.1 is a display of two different layouts of the same *graph*. Layout **b** is a planar drawing, meaning a drawing with zero edge crossings. *KK* argue that layout **a**, on the other hand, shows the symmetric structure within the *graph* more clearly. Both of these layouts are equally valid and whichever one the observer prefers is entirely up to their own personal preference, depending on which of these two features they value more.
To eliminate conflicts of this kind, the features are given in a set order. The order shown above is of descending priorities as perceived by the author of this thesis.

Both implementations are tested by the application on the same *graph*. A snapshot of the resulting layout is then saved and the amount of nodes that need to be moved or resized for the layout to be optimal in respect of the above mentioned features is determined. The percentage of to-be-altered nodes out of

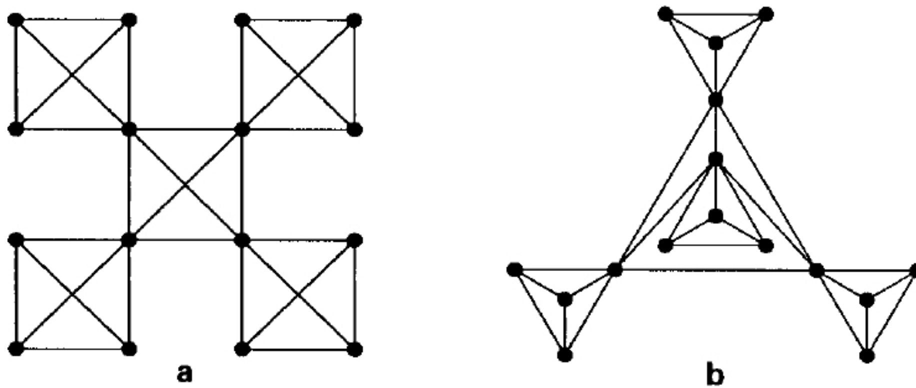


Figure 5.1: Symmetric and Planar drawings of a *graph*. Source:[12]

all nodes of the *graph* is then calculated and given as score for this particular result.

Since the algorithm implemented in the `LayoutGeneratorSmart` class is non-deterministic, each test for this layout will be executed ten times and an average value will be calculated to serve as score. Furthermore, the overall testing will be split into two phases.

5.2 Test Phase I - General Graphs

In Phase I, the performance of both layout algorithms on general *graphs* shall be determined. The field of general *graphs* will, therefore, be split into three categories

- trees
- acyclic *graphs*, which are not trees
- cyclic *graphs*

and an additional category for general *graphs* containing nodes of different sizes. For each of these categories three examples have been designed.

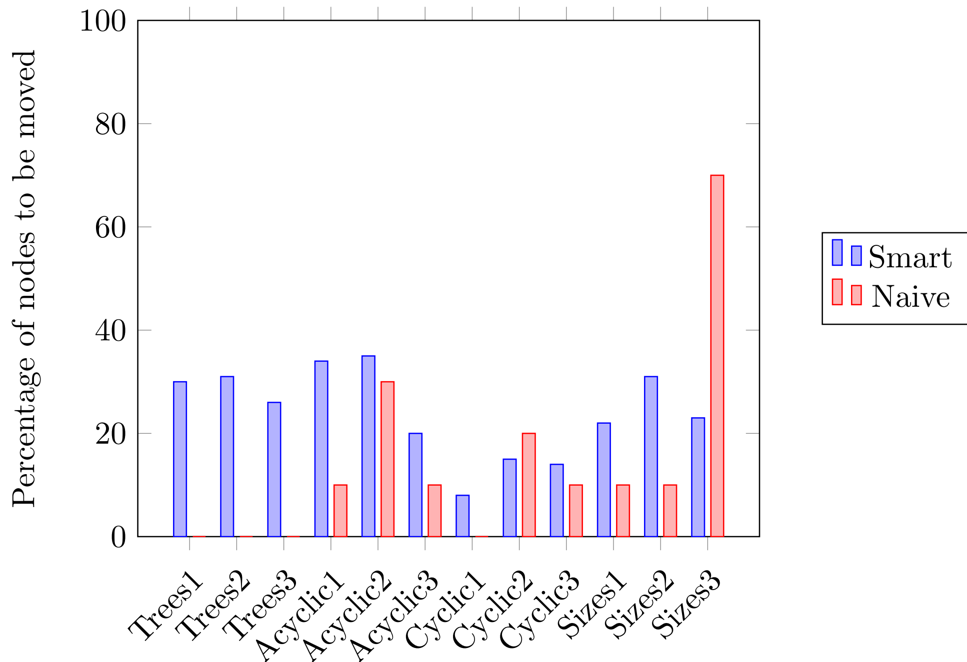


Figure 5.2: Percental Test Results of Phase I

As shown in Figure 5.2, the naive implementation provides better results with 14.17% of nodes to be moved on average, while the smart implementation results in an average of 24.08%. When calculating the variances of the results, which yield 3.57% for the naive algorithm and 0.68% for the smart algorithm with standard deviations of 18.91% and 8.25% respectively, it becomes apparent that the quality of the resulting layouts of the smart implementation is comparatively constant, while the naive implementation has some very large outliers. The full results of this phase can be found in the appendices at (B).

5.3 Test Phase II - VRL Graphs

The *VRL-Studio GitHub*-Project [16] comes with a set of example code files. In Phase II of this evaluation, the algorithms are tested on these files. This phase shows the performance of both algorithms on the specific use cases they were designed for. Out of the 26 example files ten files of varying complexity were selected as test set. Since the examples contain *graphs* of largely varying *depth* and in most cases *graphs* at a certain *depth* are scaled too small to be visible, this test is limited to a *depth* of three.

As shown in Figure 5.3, the smart algorithm produced the better results in every tested code file in this phase of the test. The averages and variances of this phase, as shown in Table 5.1, make it evident that the smart implementation not only delivers better results for this use case, but these results are also more stable than those of the naive algorithm.

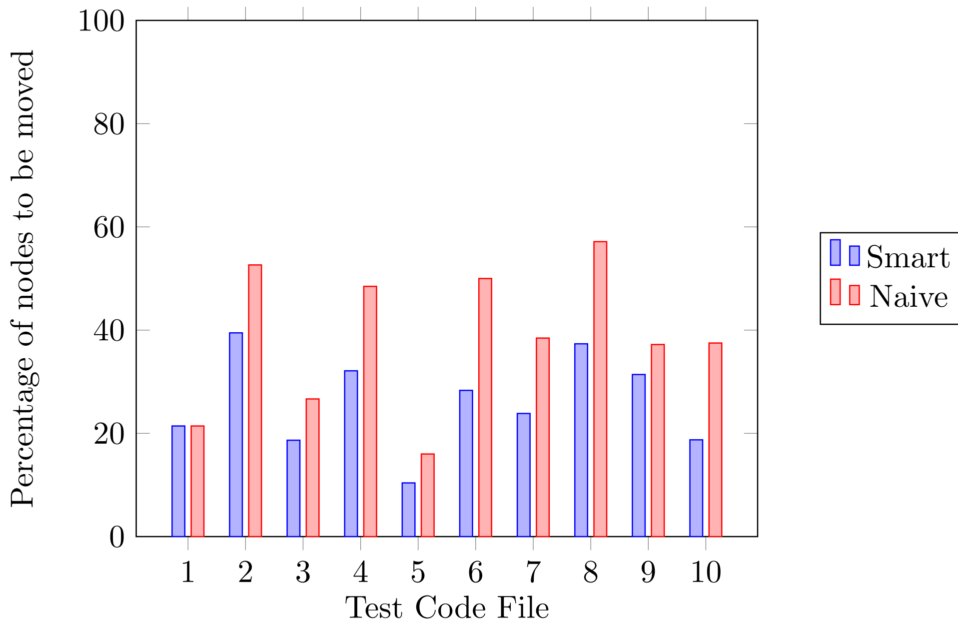


Figure 5.3: Percental Test Results of Phase II

Algorithm	Average Result	Variance	Standard Deviation
Smart	26.18%	0.76%	8.70%
Naive	38.55%	1.72%	13.11%

Table 5.1: Test Results and Variances in Phase II

The full results of this phase can be found in the appendices at (B).

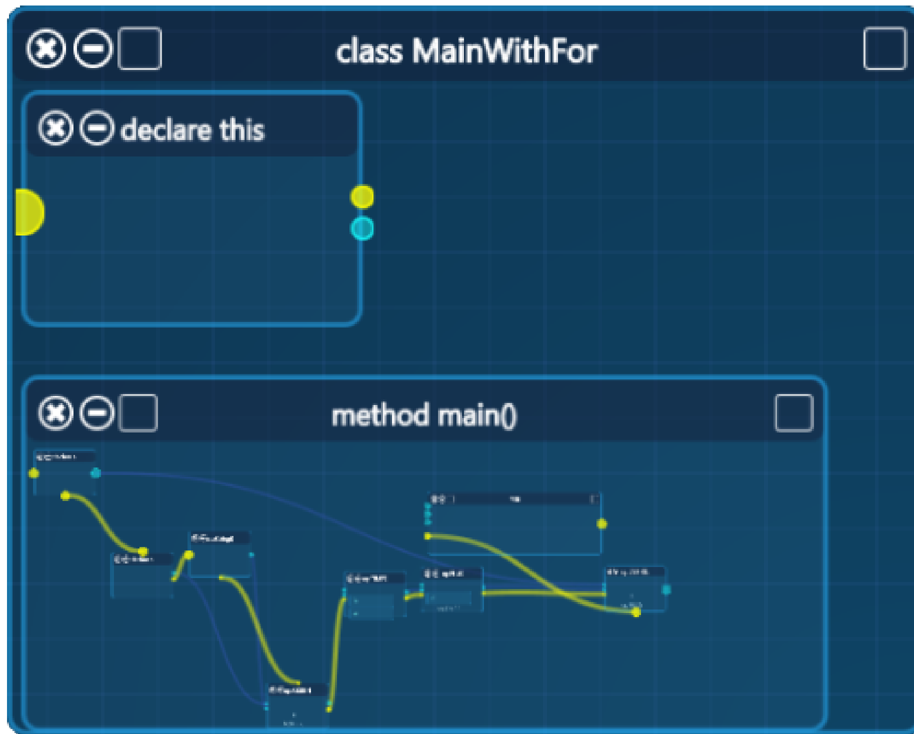


Figure 5.4: Example Result for Phase II test *'MainWithForAndMore'*.

5.4 Discussion

Figure 5.4 shows an example of one result within the *VRL-Studio*. It is clearly visible that a result like this is close to a good layout, but there is still need for some manual improvement. This is also clearly shown by the numerical test results. Cumulatively over all tests and both generators 72.44% of nodes were placed correctly and 27.56% needed to be either repositioned or resized. To find the root of these remaining mistakes, both algorithms must be examined individually.

5.4.1 LayoutGeneratorNaive

The naive algorithm had two very distinguishable peaks within its test results. When looking for clues as to where in this implementation improvements can be made, it seems obvious to take a closer look at these two test cases.

Figure 5.5 shows the result of the naive implementation in orange overlaid with a good result of the smart algorithm in light gray for the test *Sizes3*, which is the third example for the category '*Graphs* with Nodes of Different Sizes' in Phase I of this evaluation.

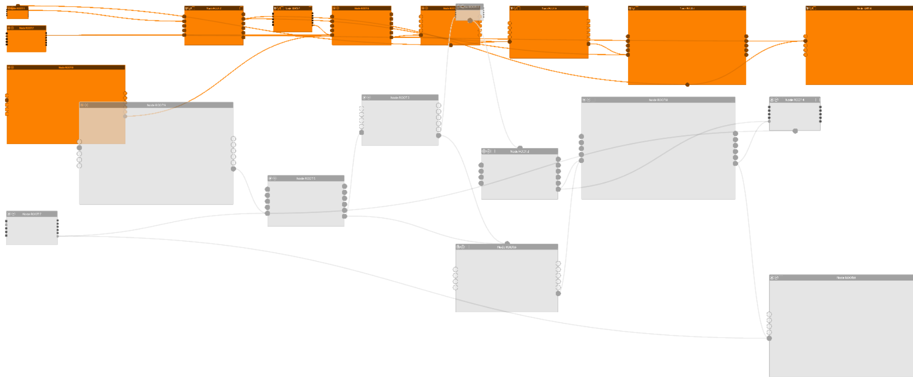


Figure 5.5: Comparison of Naive (Orange) and Smart (Light Gray) Results for Testcase *'Sizes3'*.

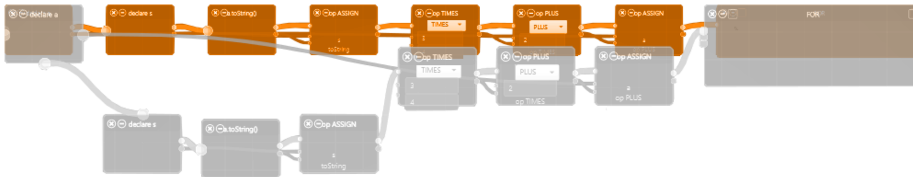


Figure 5.6: Comparison of Naive (Orange) and Smart (Light Gray) Results for Testcase *'MainWithForAndMore'*.

Figure 5.6 shows the result of the naive implementation for a single *subflow* of the test code file *MainWithForAndMore.groovy*, which showed the largest discrepancy in the results between the two algorithms, in orange overlaid with a good result of the smart algorithm for the same *subflow*.

In both examples it is clearly visible that the nodes in the smart layout are more spread out over the two dimensional area, while the naive implementation places most nodes at the top of the drawing space in a horizontal line. In Figure 5.5 the node with the id *ROOT:2* is the only vertex on the second layer of the naive result. It has a connection to the node *ROOT:8*, which was placed on the seventh layer. This placement causes this connection to run through all nodes on the layers three, four, five and six. This happens in most cases with the naive algorithm and can also be observed in Figure 5.6.

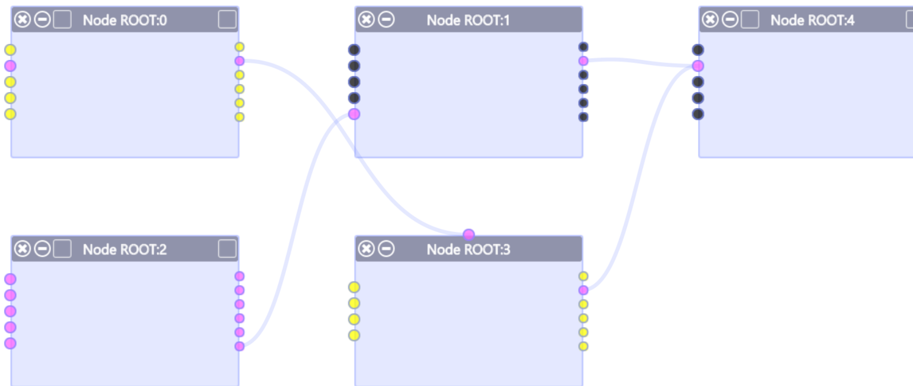


Figure 5.7: Example Result of the Naive Algorithm with a Very Unnecessary Edge Crossing.

Another problem that can be observed in Figure 5.7 is that the two edges connecting the first two layers are crossing, even though this crossing could very easily be avoided. This shows another limitation within the naive algorithm. Nodes within a single layer are sorted by their id, that is why the node *ROOT:0* is above the node *ROOT:2* on the first layer and the node *ROOT:1* is above *ROOT:3* on the second layer. If one of these two node pairs was swapped, the layout would be much better.

5.4.2 LayoutGeneratorSmart

The smart layout generator provides mostly good results that can be manually optimized with only a few changes. To find the weaknesses of this algorithm one must think about the underlying concept. Most errors produced by the algorithm are introduced during step four. The algorithm of this step does not consider the overall structure of the *graph* and only tries to push nodes away from each other to remove overlaps between them. The desired effect is achieved almost every time[3], but without the consideration of the overarching *graph* structure, the algorithm might move the wrong of the two overlapping nodes or move a node in an undesirable direction and therefore introduce edge crossings.

[3]Cumulatively over all tests with 3861 nodes only two instances of an overlap occurred.

6 Horizon

There are some clear improvements that can be made to both algorithms to create better layouts in the future.

6.1 LayoutGeneratorNaive

To best tackle the naive algorithm's weaknesses discussed in (5.4.1), this thesis suggests two changes.

6.1.1 Repositioning Nodes on Layers

A lot of edge crossings or edges running through nodes on other layers can be prevented by repositioning nodes within their respective layers. Currently, nodes are ordered by their id and are placed in an appropriate distance from each other, so they do not overlap. Nodes can, however, also be reordered to reduce the amount of edge crossings. This can be done either via an optimization algorithm or through heuristic approaches. Both variants have already been studied in the paper of *Sugiyama et al.* [10] and can easily be integrated within the `LayoutGeneratorNaive` class.

Also, nodes can be centered around their own predecessors and successors to reduce the length of edges and further optimize the continuity of the *general flow direction*.

6.1.2 Introduction of Dummy Nodes

Since edges often do not necessarily lead to the layer next to their *sender's*, they often cut through nodes on the layers in between their *sender's* and their *receiver's* layer. To eliminate these conflicts, so-called dummy nodes can be introduced on these layers. These would work analogous to the dummy nodes described in (3.2.3).

6.2 LayoutGeneratorSmart

Since the weaknesses of the smart algorithm are not as easily identifiable, some more general improvements to the `LayoutGeneratorSmart` class are suggested in this section.

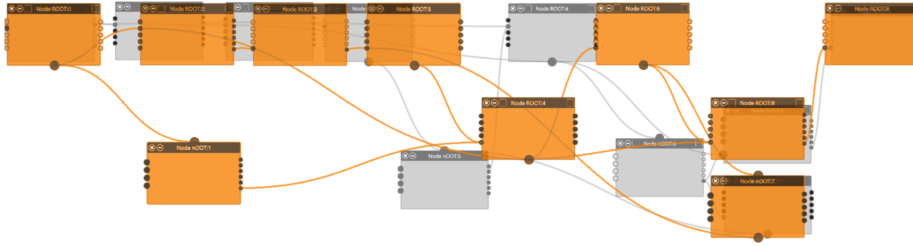


Figure 6.1: Comparison of Smart Results with (Orange) and without (Light Gray) *JUNG* Layout.

6.2.1 Combination of Step 3 and Step 4

As suggested in (5.4.2) the most problems of the smart algorithm originate in its fourth step. Step three and step four, however, both are concerned with the repulsion of nodes. They could be combined into one step that considers the relation of both nodes before calculating a fitting direction to move these nodes in. This way, the overall structure of the *graph* would determine the way nodes repel each other, which, in turn, could reduce the amount of accidentally created edge crossings in these steps.

6.2.2 Exclusion of the JUNG Algorithm

The usage of the *JUNG* algorithm and its nondeterministic nature make the whole smart algorithm nondeterministic as well. This causes a large level of inconsistency in the quality of the resulting layout, which was easy to observe by the high variance in the test results of both phases in the evaluation, even though the smart algorithm did not produce any large outliers.

Also, some additional tests revealed that the algorithm produced results of similar quality without the application of the *JUNG* layout. Figure 6.1 shows a comparison between the application of the smart algorithm with the *JUNG* layout included in orange and the application of the same algorithm without the usage of the *JUNG* layout in light gray.

That is why this thesis suggests to replace the application of the *JUNG* layout with another method for the initial node placement, that is deterministic and might provide better performance.

6.2.3 Separation of Edge Types

The class `LayoutGeneratorSmart` contains a prototype implementation of an algorithm that handles the different types of edges in the *graph* separately and can apply different priorities to them to guarantee a constant *general flow direction* to at least one of these edge types. Since the existing implementation is not able to achieve this goal, it is not run by default.

If a functioning implementation for this feature can be integrated into the algorithm, it might greatly improve its overall results.

7 Conclusion

Both algorithms provide the user with layout suggestions that can be quite close to an optimal layout — though in most cases, they do require some manual adjustment. Both algorithms have their own strengths and weaknesses and since no single layout can always be the optimal layout for every use case, as was shown with Figure 5.1, both algorithms have their uses and should continue to coexist going forward.

Since the possible improvements on the naive algorithm, however, are more directly targeted at its weaknesses, this author is of the opinion that an improved version of the `LayoutGeneratorNaive` class could very quickly be developed and surpass the quality of the currently existing algorithms, which is why it is suggested that the further development in the near future should be focused on that algorithm.

Bibliography

[1] Mozilla. *The Rust Programming Language*. published online. last access 2016-08-14. URL: https://doc.rust-lang.org/book/README.html.

[2] Google Inc. *The Go Programming Language Specification*. published online. last access 2016-08-14. URL: https://golang.org/ref/spec.

[3] Michael Hoffer, Christian Poliwoda, and Gabriel Wittum. "Visual reflection library: a framework for declarative GUI programming on the Java platform". In: *Computing and Visualization in Science* 16.4 (2013), pp. 181–192. ISSN: 1433-0369. DOI: 10.1007/s00791-014-0230-y. URL: http://dx.doi.org/10.1007/s00791-014-0230-y.

[4] Michael Hoffer. *VRL-Studio*. published online. last access 2016-08-03. URL: http://vrl-studio.mihosoft.eu/.

[5] Michael Hoffer. *VWorkflows*. published online. last access 2016-08-03. URL: http://vworkflows.mihosoft.eu/.

[6] Franz J Brandenburg, Michael Himsolt, and Christoph Rohrer. "An experimental comparison of force-directed and randomized graph drawing algorithms". In: *International Symposium on Graph Drawing*. Springer. 1995, pp. 76–87.

[7] Mohamed A. El-Sayed, Sayed Abdel-Khalek, and Hanan H. Amin. "Study of Neural Network Algorithm for Straight-Line Drawings of Planar Graphs". In: *CoRR* abs/1401.5330 (2014). URL: http://arxiv.org/abs/1401.5330.

[8] Stephen G. Kobourov. *Force-Directed Drawing Algorithms*. 2004.

[9] Joshua O'Madadhain, Daniel Fisher, and Scott White. *Java Universal Network/Graph Framework*. published online. last access 2016-08-03. URL: http://jung.sourceforge.net/index.html.

[10] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (1981), pp. 109–125.

[11] Joshua O'Madadhain, Daniel Fisher, and Scott White. *Jung2 API Javadoc*. published online. last access 2016-08-03. URL: http://jung.sourceforge.net/doc/api/index.html.

[12] Tomihisa Kamada and Satoru Kawai. "An algorithm for drawing general undirected graphs". In: *Information processing letters* 31.1 (1989), pp. 7–15.

[13] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph Drawing by Force-directed Placement". In: *Software: Practice and experience* 21.11 (Nov. 1991), pp. 1129–1164. ISSN: 0038-0644. DOI: 10.1002/spe.4380211102. URL: http://dx.doi.org/10.1002/spe.4380211102.

[14] Bernd Meyer. "Competitive learning of network diagram layout". In: *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*. IEEE. 1998, pp. 56–63.

[15] Department of Mathematics Oregon State University. *Dot Products and Projections*. published online. las access 2016-08-03. URL: http://www.math.oregonstate.edu/home/programs/undergrad/CalculusQuestStudyGuides/vcalc/dotprod/dotprod.html.

[16] Michael Hoffer. *VRL-Studio*. published on GitHub. last access 2016-08-14. URL: https://github.com/VRL-Studio/VRL.

A Fields and Methods

A.1 LayoutGeneratorNaive

Parameters:

```
private boolean recursive
private boolean autoscaleNodes
private int graphmode
private boolean launchRemoveCycles
private boolean launchCreateLayering
private boolean launchCalculateHorizontalPositions
private boolean launchCalculateVerticalPositions
private double scaling
private double subflowscale
```

Internal Fields:

```
private VFlowModel workflow
private LinkedList<Pair<Integer>> connectionList
private VNode[] nodes
private int nodecount
private int conncount
private boolean cycle
private int[] layering
private int layercount
```

Getter Methods:

```
public VFlowModel getWorkflow()
public Collection<VNode> getNodelist()
public boolean getRecursive()
public boolean getAutoscaleNodes()
public int getGraphmode()
public boolean getLaunchRemoveCycles()
public boolean getLaunchCreateLayering()
public boolean getLaunchCalculateHorizontalPositions()
public boolean getLaunchCalculateVerticalPositions()
public double getScaling()
public double getSubflowscale()
public boolean getDebug()
public LinkedList<Pair<Integer>> getModelGraph()
public int[] getLayering()
```

Setter Methods:

```
public void setWorkflow(VFlowModel pworkflow)
public void setNodelist(Collection<VNode> pnodelist)
public void setRecursive(boolean precursive)
public void setAutoscaleNodes(boolean pautoscaleNodes)
public void setGraphmode(int pgraphmode)
public void setLaunchRemoveCycles(boolean plaunchRemoveCycles)
public void setLaunchCreateLayering(
    boolean plaunchCreateLayering)
public void setLaunchCalculateHorizontalPositions(
    boolean plaunchCalculateHorizontalPositions)
public void setLaunchCalculateVerticalPositions(
    boolean plaunchCalculateVerticalPositions)
public void setScaling(double pscaling)
public void setSubflowscale(double psubflowscale)
public void setDebug(boolean pdebug)
public void setModelGraph(
   LinkedList<Pair<Integer>> pconnectionList)
public void setLayering(int[] playering)
```

Constructors:

```
public LayoutGeneratorNaive()
public LayoutGeneratorNaive(boolean pdebug)
```

Other Methods:

```
private void initialize()
public boolean setUp()
pivate boolean checkCycles()
private void removeCycles()
private void remCycR(Integer curr, LinkedList<Integer> path,
    boolean[] checked)
public void generateLayout()
private void runSubflows()
private void autoscaleNodes()
private void createLayering()
private boolean allLocked(boolean[] locked)
private void calculateVerticalPositions()
private void calculateHorizontalPositions()
private Integer getNodeID(VNode pnode)
```

A.2 LayoutGeneratorSmart

Parameters:

```
private boolean recursive
private boolean autoscaleNodes
private int layoutSelector
private double aspectratio
private int graphmode
private boolean launchRemoveCycles
```

```
private boolean launchSeparateDisjunctGraphs
private boolean launchSeparateEdgeTypes
private boolean launchJungLayout
private boolean launchRotate
private boolean launchOrigin
private boolean launchPushBack
private boolean launchDisplaceIdents
private boolean launchForcePush
private boolean launchAlignNodes
private int maxiterations
private double scaling
private double subflowscale
private double direction
private double alignmentThreshold
private boolean debug
```

Internal Fields:

```
private VFlowModel workflow
private DirectedGraph<VNode, Connection> jgraph
private VNode[] nodes
private Layout<VNode, Connection> layout
private int nodecount
private int conncount
private Point2D graphcenter
private Pair<Integer>[] origin
private boolean cycle
```

Getter Methods:

```
public VFlowModel getWorkflow()
public DirectedGraph<VNode, Connection> getModelGraph()
public Collection<VNode> getNodelist()
public boolean getRecursive()
public boolean getAutoscaleNodes()
public int getLayoutSelector()
public double getAspectratio()
public int getGraphmode()
public boolean getLaunchRemoveCycles()
public boolean getLaunchSeparateDisjunctGraphs()
public boolean getLaunchSeparateEdgeTypes()
public boolean getLaunchJungLayout()
public boolean getLaunchRotate()
public boolean getLaunchOrigin()
public boolean getLaunchPushBack()
public boolean getLaunchDisplaceIdents()
public boolean getLaunchForcePush()
public boolean getLaunchAlignNodes()
public int getMaxiterations()
public double getScaling()
public double getSubflowscale()
public double getDirection()
```

```
  public double getAlignmentThreshold()
  public boolean getDebug()
```

Setter Methods:

```
  public void setWorkflow(VFlowModel pworkflow)
  public void setModelGraph(
      DirectedGraph<VNode, Connection> pjgraph)
  public void setNodelist(Collection<VNode> pnodelist)
  public void setRecursive(boolean precursive)
  public void setAutoscaleNodes(boolean pautoscaleNodes)
  public void setLayoutSelector(int playoutSelector)
  public void setAspectratio(double paspectratio)
  public void setGraphmode(int pgraphmode)
  public void setLaunchRemoveCycles(boolean plaunchRemoveCycles)
  public void setLaunchSeparateDisjunctGraphs(
      boolean plaunchSeparateDisjunctGraphs)
  public void setLaunchSeparateEdgeTypes(
      boolean plaunchSeparateEdgeTypes)
  public void setLaunchJungLayout(boolean plaunchJungLayout)
  public void setLaunchRotate(boolean plaunchRotate)
  public void setLaunchOrigin(boolean plaunchOrigin)
  public void setLaunchPushBack(boolean plaunchPushBack)
  public void setLaunchDisplaceIdents(
      boolean plaunchDisplaceIdents)
  public void setLaunchForcePush(boolean plaunchForcePush)
  public void setLaunchAlignNodes(boolean plaunchAlignNodes)
  public void setMaxiteration(int pmaxiterations)
  public void setScaling(double pscaling)
  public void setSubflowscale(double psubflowscale)
  public void setDirection(double pdirection)
  public void setAlignmentThreshold(double palignmentThreshold)
  public void setDebug(boolean pdebug)
```

Constructors:

```
  public LayoutGeneratorSmart()
  public LayoutGeneratorSmart(boolean pdebug)
```

Other Methods:

```
  private void initialization()
  private boolean allNodesSetUp()
  private void createGraph(
      ObservableMap<String, Connections> allConnections)
  private Pair<Integer>[] getOrigin()
  private Point2D getGraphCenter()
  private boolean checkCycle()
  private void removeCycles()
  private void remCycR(VNode curr, LinkedList<VNode> path,
      boolean[] checked)
  private void separateDisjunctGraphs()
  private void separateEdgeTypes()
```

```
public void generateLayout()
private void runSubflows()
private void autoscaleNodes()
private void stepLayoutApply()
private double findMaxPathWidth()
private void stepRotate()
private double getAvgDir()
private void stepOrigin()
private void stepPushBack()
private void displaceIdents()
private void forcePush()
private double getRealNodeDist(VNode node1, VNode node2)
private double getDesiredNodeDist(VNode node1, VNode node2)
private void alignNodes()
private Integer getNodeID(VNode pnode)
```

B Testresults

	# nodes to be moved (of 10)										avg
Tree1	3	3	2	5	3	2	4	3	2	3	3
Tree2	3	2	3	3	4	3	5	3	3	2	3.1
Tree3	3	2	1	2	4	3	4	3	1	3	2.6
Acyclic1	3	3	5	2	6	5	3	4	1	2	3.4
Acyclic2	3	4	2	3	3	4	4	4	4	4	3.5
Acyclic3	3	1	0	2	3	3	2	1	2	3	2
Cyclic1	1	1	1	0	1	0	1	1	1	1	0.8
Cyclic2	1	1	3	1	2	1	2	1	3	0	1.5
Cyclic3	1	1	2	1	2	1	2	2	1	1	1.4
Sizes1	2	3	2	2	2	2	3	2	2	2	2.2
Sizes2	4	3	3	3	3	2	3	3	3	4	3.1
Sizes3	3	2	2	1	3	3	3	2	2	2	2.3

Table B.1: Phase I Test Results of the Smart Algorithm

Test	# nodes to be moved (of 10)
Tree1	0
Tree2	0
Tree3	0
Acyclic1	1
Acyclic2	3
Acyclic3	1
Cyclic1	0
Cyclic2	2
Cyclic3	1
Sizes1	1
Sizes2	1
Sizes3	7

Table B.2: Phase I Test Results of the Naive Algorithm

Test	# nodes to be moved										avg	# nodes
1	2	2	4	2	4	3	4	3	3	3	3	14
2	8	9	9	7	8	7	6	5	7	9	7.5	19
3	4	4	2	2	2	3	3	3	3	2	2.8	15
4	8	13	11	14	8	10	10	12	10	10	10.6	33
5	3	2	3	2	2	4	2	3	3	2	2.6	25
6	4	3	4	3	6	2	3	5	3	1	3.4	12
7	4	3	3	4	4	2	2	3	1	5	3.1	13
8	16	16	22	21	12	15	24	26	15	16	18.3	49
9	18	13	17	19	15	11	13	8	16	5	13.5	43
10	0	2	2	1	1	3	2	1	1	2	1.5	8

Table B.3: Phase II Test Results of the Smart Algorithm

Test	# nodes to be moved	# nodes
1	3	14
2	10	19
3	4	15
4	16	33
5	4	25
6	6	12
7	5	13
8	28	49
9	16	43
10	3	8

Table B.4: Phase II Test Results of the Naive Algorithm

C Glossary

VRL The *Visual Reflection Library* can be used to create visual representations of source code files.

VRL-Studio The *VRL-Studio* is a visual programming environment developed at the *Goethe Center for Scientific Computing* and is based on *VRL* and *VWorkflows*.

VWorkflows *VWorkflows* is a Java library implementing a graph model with visualization and interactivity features. See (3.1).

graph A *graph* G is a tuple (V, E) consisting of two sets V and E. V is a set of n vertices v_i while E is a set of m two-tuples (v_i, v_j) with $v_i, v_j \in V$ representing directed edges pointing from the vertex v_i to v_j.

origin *Origin* nodes are all nodes within a *graph*, which have an in-degree of zero, meaning they do not have any predecessors.

in front The term *in front* in this context means if vertex v_i is in front of vertex v_j, then v_i has a smaller coordinate on the axis of the *general flow direction*, so that an edge from v_i to v_j runs in the *general flow direction*.

behind The term *behind* describes the arrangement of the nodes v_i and v_j whereby an edge from v_i to v_j runs against the *general flow direction* if v_i is *behind* v_j.

subflow A *subflow* is a program flow, that is contained within a node of another program flow.

subflow node The term *subflow node* refers to a node that contains a *subflow*. For future clarity it is important to make the distinction between *subflow nodes*, which are nodes that contain a *subflow*, and the nodes of a subflow, which are regular nodes contained within a *subflow*.

depth The *depth* of a `VFlowModel` describes the hierarchical position of the *graph* within the entirety of the project. The highest level flow *graph* (also called *root flow*) has a *depth* of zero, while each *subflow* of a certain *graph* always has a *depth* of one larger than its parent flow.

sender The *sender* of an edge is the node in which the edge originates.

receiver The *receiver* of an edge is the node, which the edge points towards.

D List of CD Contents

- readme.txt
  Contains this list of contents and some tutorial text.
- thesis.pdf
  Contains a digital version of this thesis.
- Test Code Files/
  Contains the ten code files used as test examples in Phase II of the evaluation.
  Also contains the file "vrl-tests.txt" which assigns the individual test code files the indices, which are used in the result tables.
- testimages/
  Contains the snapshots of all test results in the following directory tree:
  - smart/
    Contains the results for the smart algorithm.
    - Each folder is named after the corresponding test.
      - The individual test images are named in the pattern:
        number of the test.png: the root flow of the test
        *number of the test*_*subflow id*.png: each subflow that is not visible in the root flow view
  - naive/
    Contains the results for the naive algorithm.
    - Each folder is named after the corresponding test.
      - The individual test images are named in the pattern:
        0.png: the root flow
        0_*subflow id*.png: each subflow that is not visible in the root flow
- VRL/
  Contains the Gradle project of VRL-Studio in the version that was used in the evaluation.
  Execute "`gradlew :vrl-ui:run`" to run VRL-Studio.
- VWorkflows-master/
  Contains the Gradle project of VWorkflows in the version that was used in the evaluation.
- vworkflows-demo/
  Contains a compiled version of the VWorkflows project.
  Execute "vworkflows-demo/vworkflows-demo-0.2.4.2.jar" to start VWorkflows.