České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Tušla Tomáš

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Algoritmus pro rozmístění uzlů hierarchických vývojových diagramů

Pokyny pro vypracování:

Seznamte se s metodami pro výpočet rozmístění uzlů grafu, speciálně vývojových diagramů. Na základě analýzy navrhněte a implementujte algoritmus pro rozmístění uzlů hierarchického vývojového diagramu. Hierarchický vývojový diagram je takový, kde jsou jednotlivé uzly seskupeny do bloků a je tedy třeba vyřešit jak rozmístění uzlů uvnitř bloků, tak rozmístění bloků samotných. Algoritmus navrhněte a implementujte tak, aby zajistil koherentní přechod od zobrazení celého grafu k zobrazení podgrafu. Koherentním přechodem je myšleno, že bude zachována relativní pozice uzlů a bloků (např. uzel U1 je napravo od uzlu U2 či blok B2 je pod blokem B1). Výsledný algoritmus otestujte alespoň na 5 hierarchických diagramech různé složitosti (20 až 100 uzlů).

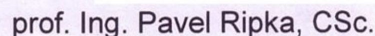
Seznam odborné literatury:

[1] TAMASSIA, Roberto (ed.). Handbook of graph drawing and visualization. CRC press, 2013.
[2] T. REINHARD. Complexity Management in Graphical Models. Switzerland, Zurich, University of Zurich: 2010. Doctoral Thesis. University of Zurich, Faculty of Economics, Business Administration and Information Technology.

Vedoucí: Ing. Ladislav Čmolík, Ph.D.

Platnost zadání do konce zimního semestru 2018/2019

L.S.



prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 19.4.2017

master's thesis

Layout of hierarchical flow charts

Bc. Tomáš Tušla

May 2017

Ing. Ladislav Čmolík, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Computer Graphics and Interaction

Acknowledgement

I would like to thank Ing. Ladislav Čmolík Ph.D. for his guidance when supervising this thesis.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

Abstract

Tato práce se zabývá tvorbou rozvržení pro hierarchické vývojové diagramy. Práce obsahuje analýzu několika algoritmů a jejich nedostatků při tvorbě tohoto typu rozvržení. Práce dále popisuje návrh a implementaci upraveného hierarhického rozvrhovacího algoritmu, který může být použit k tvorbě rozvržení pro hierarchické vývojové diagramy. Tento algoritmus zachovává stabilitu rozvržení u podgrafů a také podporuje různé strategie pro agregaci hran nebo například dynamickou výšku vrstev. Algoritmus je implementován v rámci aplikace Ideal Graph Visualizer, která umožňuje zobrazit hierarchické vývojové diagramy. Práce pak obsahuje zhodnocení vytvářených rozvržení a navrhuje možná vylepšení algoritmu.

Klíčová slova

Hierarchický vývojový diagram; kreslení grafů; hierarchické rozvržení; stabilita rozvržení

Abstract

This thesis deals with creation of layouts for hierarchical flow charts. The thesis contains analysis of several algorithms and their shortcomings, when creating this type of layout. The thesis then describes design and implementation of a modified hierarchical layout algorithm, which can be used to create layouts of hierarchical flow charts. This algorithm maintains layout stability in subgraphs and it also supports different edge aggregation strategies or, for example, dynamic height of layers. The algorithm is implemented as part of application Ideal Graph Visualizer, which allows to visualize hierarchical flow charts. Thesis then contains evaluation of created layouts and suggests possible improvements of the algorithm.

Keywords

Hierarchical flow chart; graph drawing; hierarchical layout; layout stability

Contents

1 Introduction **1**
1.1 Goal and motivation . . . 1
1.2 Thesis structure . . . 2

2 Analysis **3**
2.1 Hierarchical flow chart . . . 3
2.2 Layout algorithm requirements . . . 4
2.3 Layout algorithms . . . 4
2.3.1 Comparison of algorithms . . . 8

3 Design **11**
3.0.1 Hierarchy support . . . 11
3.0.2 Filtering support and layout stability . . . 11
3.1 Layout algorithm . . . 12
3.2 Data structure . . . 13
3.3 Outer layout algorithm . . . 13
3.4 Inner layout algorithm . . . 14
3.4.1 Removing cycles . . . 14
3.4.2 Assigning layers to nodes . . . 15
3.4.3 Proper layering . . . 16
3.4.4 Aggregate edges . . . 17
3.4.5 Crossing reduction . . . 18
3.4.6 Assigning coordinates . . . 19
3.4.7 Edge routing . . . 20

4 Implementation **23**
4.1 Ideal Graph Visualizer . . . 23
4.1.1 Features of IGV . . . 24
4.2 Code structure . . . 26
4.2.1 Implemented code . . . 27
4.3 Implementation features . . . 28
4.3.1 Dynamic gap between layers . . . 28
4.3.2 Interrupting too long connectors . . . 28

5 Testing **31**
5.1 Performance overview . . . 31
5.2 Running time of individual steps . . . 32
5.3 Running time with and without clusters . . . 33
5.4 Conclusion . . . 33

6 Results **35**
6.1 Node filtering . . . 35
6.2 Comparison with the original algorithm . . . 37
6.3 Edge aggregation strategies . . . 38
6.4 Layout with clusters . . . 40

7 Coneclusion 43
7.1 Future improvements 43
7.1.1 Two step algorithm 43
Bibliography 45

Abbreviations

IGV	Ideal Graph Visualizer
JVM	Java Virtual Machine

1 Introduction

Flow charts are widely used type of diagrams in many different use cases. Be it some real world business process or visualization of an algorithm. While there are many tools that allow easy creation of flow charts, often these tools leave the responsibility of creating visually pleasing layout in hands of the flow chart creator.

1.1 Goal and motivation

Goal of this thesis is to create an algorithm that can generate a flow chart layouts from an input graph automatically. The algorithm should also have some useful properties. One of these properties is allowing nodes to be grouped into clusters.

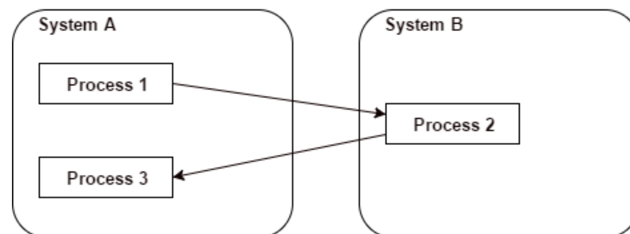


Figure 1 Hierarchical flow chart

Then in the layout created by the algorithm, all the nodes that are part of the same cluster will be placed in vicinity of each other. Figure 1 shows such flow chart. Having all the processes grouped together makes it easier for users to find all the processes that are part of the same system.

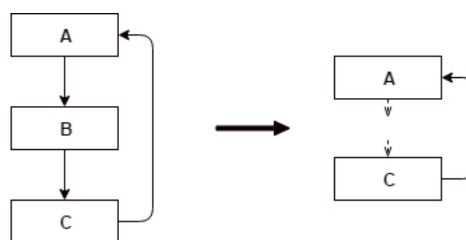


Figure 2 Stability of layout

Another very useful property is maintaining stability of the layout when only part of the whole graph is shown. The layout can be considered stable, if relative positions of nodes in the original layout are maintained in the layout created for subgraph of the original graph as shown in figure 2. The last important property is interactive computation for instances with few thousands of nodes.

An application called Ideal Graph Visualizer is a Java application used to visualize hierarchical flow charts. This application allows users to visualize different steps of

bytecode optimizations done by JVM. In these visualizations, nodes represent instructions and edges represent passed values. Users can then use these visualizations to check if, for example, a result value of an instruction is correctly passed to next instruction, or they can check that there aren't any unwanted instructions used. To achieve this, users have to be able to find nodes they are interested in and then they have to be able to track where their edges lead to. This can be often very difficult due to graphs being too large. The IGV helps with this, by allowing users to turn off visibility of nodes they aren't interested in, which makes the graph smaller. Unfortunately the layout is recreated every time when visibility of node changes and this new layout can be very different when compared to the previous one. It would be helpful if the layout stayed similar when switching visibility of nodes, as it would decrease time an user needs to orient himself, after the layout is modified.

1.2 Thesis structure

The following text of this thesis is divided into five chapters. The second chapter contains description of hierarchical flow chart and properties, which should be achieved by final implementation of the layout algorithm. In the second part of the chapter, there is an analysis of different graph layouts and their algorithms. In the third chapter is a design and explanation of individual steps of the implemented algorithm. The fourth chapter contains details of IGV environment and implementation details together with some implemented features of the algorithm. The fifth chapter contains details of testing of running time the implemented algorithm. In the sixth chapter, results of the implemented algorithm are shown on several different examples of diagrams. In the last chapter, there is a conclusion of this thesis.

2 Analysis

This chapter contains description of a hierarchical flow chart and it's properties that should be taken into consideration when creating a hierarchical flow chart layout. The description is then followed by an analysis of several possible layout algorithms, which could be used to create an automatic layout for the hierarchical flow charts.

2.1 Hierarchical flow chart

A hierarchical flow chart has the same structure as a standard flow chart, meaning graph structure that contains nodes and edges that connect these nodes. All edges have assigned direction and there can be both multiple ingoing and outgoing edges per each node. Also cycles, including loops, are allowed. The nodes themselves can have different dimensions and shapes. The main usage of a flow chart is to visualize a sequence of individual steps or decisions, which are made while working on some process. Each node then represents an action or a decision and edges represent transitions between these actions.

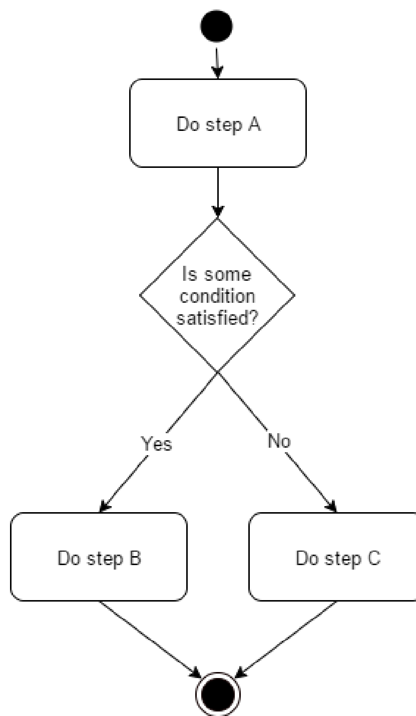


Figure 3 Flow chart

Flow charts also generally have one direction (see figure 3), most commonly from up to down, in which the steps of a process are heading, as progress through the process is made. While it is not a strict rule, it makes these charts easier to follow and it can even help users to guess how far in the process they are.

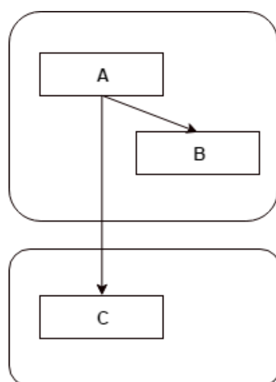


Figure 4 Hierarchy of nodes

Unlike in the standard flow chart, in case of the hierarchical flow chart, it is allowed for nodes to be subnodes of some other node. The edges can be not only between nodes on the same level in this hierarchy, but also across different levels of different nodes as shown in figure 4.

As for the nodes, flow charts use differently shaped nodes for actions and decisions, so nodes can have different dimensions. The edges are not limited and can connect any two nodes so they can sometimes return even several steps back in the process. This means that it is possible for the flow chart to contain cycles and also, that it is not always possible to have all edges heading in top-down direction.

2.2 Layout algorithm requirements

Layout algorithm should support creating layouts for hierarchical flow charts. This means that it has to be able to work with nodes of different sizes. It also has to support nodes being grouped into clusters. The algorithm will then automatically keep the nodes together in created layout. These properties are not the only ones that should be achieved by the layout algorithm. As mentioned earlier, the algorithm should also create stable layouts, meaning relative position of nodes should be maintained when creating layouts of a subgraph, to minimize disruptions to users mental map. Another important property is that the algorithm allows to create layouts interactively. All of the algorithms have implementations that can be used interactively. Although it may not be possible to maintain this property after adding some of features that the algorithm is missing. The algorithm should be also able to work at least with hundreds of nodes interactively.

Then there are aesthetic requirements, while these may not be necessarily as critical as requirements that were already described, the aesthetic requirements are still very important. As shown in case of the Ideal Graph Visualizer, users have to be able to find nodes easily and they have to be able to track where edges lead to. This can be difficult especially when there are overlapping nodes, or an edge and a node overlap. Another problems are edge cluttering and edge crossings. There are also some minor improvements, like having the edges go in top-down direction, which also help.

2.3 Layout algorithms

There are many layout algorithms, but most of them are not suitable for creating hierarchical flow chart layouts. In this section, there will be presented only those

suitable algorithms which could be considered as a basis of the final algorithm. There is force-directed layout algorithm, which is very fast algorithm, even though it's results are least controllable. Another possibility is hierarchical layout algorithm, this algorithm has much higher control over the result, but it is slower. The last discussed algorithm is compact layout algorithm, which support grouping of nodes and stability of layout, but it's also much more complicated and has even worse performance, then the hierarchical layout algorithm. While none of these algorithms fits perfectly, it may be possible to modify them.

Force-directed Layout Algorithm Force-directed layout algorithm is one of the most popular layout algorithms. While being extremely easy to implement, it has also reasonable asymptotic running time of $O(n^3)$, n representing number of nodes, when using naive approach. This running time can be lowered[10] even to $O(n*log(n))$ with heuristic, making the algorithm very fast. This allows the algorithm to work in real time even with thousands of nodes.

An advantage of the force-directed layout algorithm is that it works with any type of graph. The downside is relatively small control over the resulting layout. The algorithm is also iterative. At the beginning, it creates some initial state, which is then improved in every iteration by decreasing overall energy of the graph [4]. If bad initial state is selected, then the algorithm can find only local minimum. This local minimum can still be a very poor solution compared to the best possible solution that would exist for given graph.

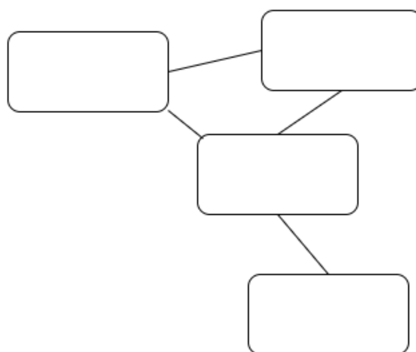


Figure 5 Force-directed layout

As the name suggests, important part of the algorithm are forces. There are two types of forces, the first is an attractive force and the second is a repulsive force. The attractive force is generally a force between two nodes connected by an edge. This edge can have some priority assigned. The priority represents strength of the attractive force which pulls both nodes together. The repulsive force is generally a force between every two nodes, which tries to push the nodes apart. By combining these two forces, the algorithm tries to balance having nodes with connections between themselves close together, while preventing nodes overlapping with each other using the repulsive force.

At the beginning of the algorithm, all nodes are placed at random locations. Then the algorithm computes forces that are affecting each node and moves nodes accordingly. This step is repeated until there are either no unbalanced forces, or if these unbalanced forces are lower than some threshold. Figure 5 shows such possible layout.

Hierarchical Layout Algorithm Hierarchical layout is, as its name suggests, creating hierarchy of nodes. With each node having assigned some layer. Unlike the Force-directed layout algorithm, which can be used with any type of graph, the hierarchical layout algorithm requires directed acyclic graph. Because this requirement cannot be always achieved, it is possible to transform an input graph containing cycles to acyclic graph by switching direction of ideally minimal number of edges. The hierarchical layout algorithm also tries to achieve some aesthetic criteria [3], for example minimal width, minimal height, but also more complex criteria like minimal number of edge bends or having minimal edge crossings. Often these criteria oppose each other, so commonly each implementation will try to focus only on a few aesthetic rules. Another disadvantage of hierarchical layout algorithm is that many of its steps are NP difficult. This makes the algorithm unsuitable when trying to create layouts in real time, or when working with graphs that contain many nodes and edges.

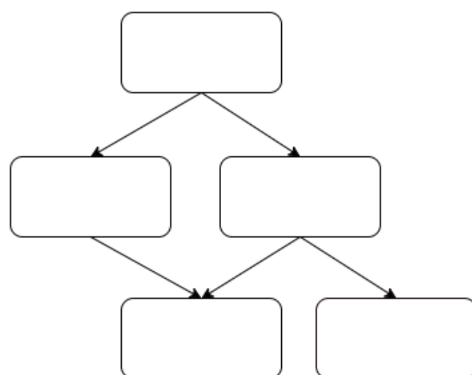


Figure 6 Hierarchical layout

Due to these constraints, it is better to use heuristic algorithms. The most common heuristic solution is using Sugiyama's framework. This algorithm splits creation of the layout to several steps. One of the biggest advantages of this approach is an ability to configure the resulting layout based on which implementation of each step is chosen. With some of the steps being optional and the others having many implementations [1] it is possible to adjust both asymptotic running time and aesthetic criteria.

The steps of the Sugiyama's framework [2] are:

Remove cycles In this step, it's required to convert an input graph to directed acyclic form. This is done by either switching direction of problematic edges, or by removing them from the graph during the algorithm.

Assign layers to nodes In this step, every node is assigned to some layer. Generally, the goal is to satisfy aesthetic condition, that all incoming edges come from nodes, which have assigned lower layer number. Also assigning layers to nodes in a way that reduces length of the edges benefits both readability of layout and performance.

Reduce long edges If there is an edge between two nodes which are not in adjacent layer, then this edge is converted to a path between original nodes with special dummy node in each layer it goes through. This is done to help navigate long edges through graph and avoid edges overlapping nodes.

Aggregate edges Optional step. If group of nodes shares a parent node, or a child node, it is possible to aggregate such edges to one. This is helpful especially if the edges are long. This will decrease number of dummy nodes and edges, making the algorithm faster and resulting layout easier to read.

Crossing reduction Probably the most important step of the algorithm. In this step, nodes are assigned positions relative to each other in order to find such permutation which would have minimal number of edge crossings. Because ideal solution requires to compute all possible permutations, heuristic approaches are used. The original Sugiyama framework uses barycentric method, which places each node on average position computed using positions of children of the node. But it's possible to use many different implementations.

Assigning coordinates This step can be combined with previous step, if not, then in this, step each node is assigned y coordinate according to its layer and x coordinate in such a way that does not violate relative positioning created in previous step. After this step, the resulting layout will be similar as the one in figure 6.

Compact Layout Algorithm Unlike previous algorithms which were from the category of automatic graph drawing algorithms, this algorithm is a layout adjustment algorithm [6]. The difference is that the automatic graph drawing algorithms receive all nodes and edges and then create a layout. On the other side, the layout adjustment algorithm has some initial layout and then receives changes. These changes can be anything from zooming some node, changing size of a node, to adding completely new nodes and edges. This makes these algorithms more suitable in places where there are a lot of changes to the graph in real time.

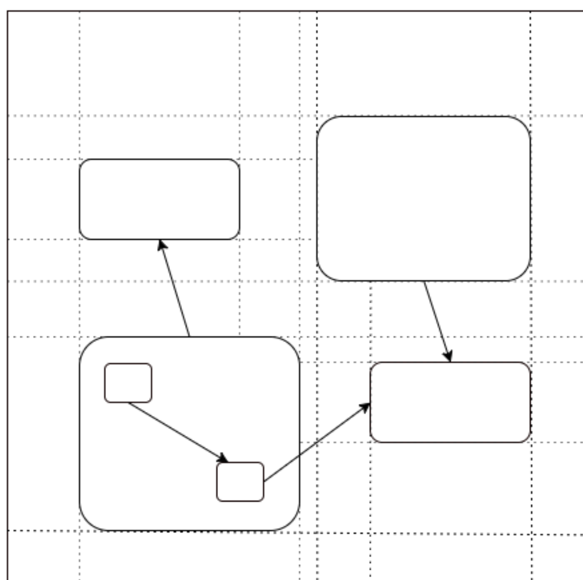


Figure 7 Compact layout including interval lines

The compact layout [6] uses intervals to denote both position and dimension of node. Each node has four these intervals (see figure 7) which create a bounds of the node. These intervals can change during life cycle of the graph, for example when scaling node size or making it invisible. While these intervals do not create strict grid, they are used to achieve some grid like properties. It is possible to use them to maintain relative positions when some of the nodes are resized. Node can be made invisible by simply setting appropriate intervals to zero size. This stability is guaranteed only as long as the change applied to the layout is not node creation or removal. In these cases, structure of intervals has to be changed.

The algorithm has several problems. The first is that the algorithm required fixed

screen size. This is of course impossible to achieve when working with different sizes of graphs. The algorithm also doesn't have any procedure, that would decide where new nodes should be added. This makes reading the graph harder. It also means that creating paths for the edges is more complicated. Among possible solutions to this is creating a grid and then using some shortest path algorithm to find a possible route, other possibility is marking every corner of each node and then navigating the edges around. The first solution can be optimized by not using some random grid, but by using the original structure of the intervals. Also it's possible to create metrics for the edges to minimize how many times the edge will bend or how long it will be. It is also possible to minimize how many other edges will an edge cross. But finding such path can be rather time consuming.

While this layout allows better manipulation with graph, it is also more complicated to implement and requires to have robust data structures and algorithms to be able to run in real time. The algorithm by itself also doesn't bring much more than the other algorithms, but rather allows implementation of other features which would be able to make good use of this layout structure. This is quite a big disadvantage since many of these features would have to be implemented by creators of applications, which would use this layout, to be compatible with both the algorithm and the application itself.

2.3.1 Comparison of algorithms

While none of the algorithms is ideal on its own, each of them has some positive traits which should be in the final algorithm.

Force-directed algorithm is very versatile, easy to implement and works very fast in real time, but it lacks any sort of structure. The algorithm supports nodes of different dimensions and has mechanism to prevent node overlapping. Normally a repulsive force is based on distance of nodes. This works well, if nodes have shape of a circle, but it can be problematic to balance the forces for a rectangle shaped node with very different height and width. Having too strong repulsive force can then create big gaps in layout, on the other side, too weak forces can sometimes result in overlapping nodes. Another problem is that edge crossings are not minimized and the edges themselves can overlap with nodes, which is undesirable. The algorithm has also no support for subnodes, although it can be very easily added. The bigger problem is that a result of the algorithm relies heavily on initial positions of nodes.

Hierarchical layout algorithm, compared to the force-directed algorithm, brings solid structure and support for flow charts. It has also no problem with different shapes of nodes, prevents node overlapping and tries to minimize the number of edge crossings. The hierarchical layout algorithm doesn't support subnodes by itself, but it is quite flexible and it is possible to add support for subnodes quite easily. The hierarchical algorithm has two big problems. The first problem is its high time complexity, which can be solved by using heuristic. The second problem is that hierarchical algorithm will produce bigger layouts than force-directed algorithm and compact layout, especially if there are big differences between dimensions of nodes.

The compact layout algorithm, on the other side, allows better manipulation with resulting layout, it also supports different dimensions of nodes and prevents nodes from overlapping. It's also the only algorithm, that is able to work with subnodes and it also creates stable layout. But same as for the force-directed algorithm, it doesn't naturally form flow charts. Another disadvantage is that the algorithm is more complicated. Even basic edge routing is much more complex than the one in the hierarchical algorithm, even though it can be modified to minimize the edges better than in hierarchical approach.

But it requires much more processing power. The compact layout was tested by its authors using graphs with about sixty nodes and about same number of edges [6]. These are relatively small instances, on the other side both force-directed algorithm and hierarchical algorithm can be used with much bigger graphs.

Out of these algorithms, the hierarchical layout algorithm is the most viable solution in this case. It is a reasonable compromise between simplicity and complexity. It satisfies many of required features that force-directed algorithm is missing, while at the same time, it's faster and less complicated than the compact layout algorithm. The hierarchical layout algorithm can be very easily modified, so it's possible to add missing features while maintaining positive properties of hierarchical layout.

3 Design

This chapter contains description of new algorithm, which is based on hierarchical layout algorithm that selected in previous chapter. This algorithm is built upon Sugiyama's framework with several modifications. These modifications should allow the algorithm to fulfill requirements specified in the previous chapter and thus be possibly used to create hierarchical flow charts.

As described in previous chapter, the main weak point of hierarchical algorithm is a lack of support for grouping nodes into clusters. This algorithm also doesn't naturally create stable layouts and has high time complexity without heuristics. In this chapter, these problems are addressed with possible solutions.

Before explaining the algorithm itself, it is important to be able to distinguish nodes without subnodes from the nodes which have some subnodes. In the following text, when writing about a node, it means that it is irrelevant if the node has any subnodes or not. On the other side, if a node is called a cluster, then it contains some subnodes.

3.0.1 Hierarchy support

Because the hierarchical layout by itself doesn't have any support for subnodes, it has to be added. This can be done by doing separately an inner layouts of subnodes in every cluster. Then, in the second phase a layout on a level of the clusters is created. The order of doing the layout from inner subnodes is required since it is important to know dimensions of subgraph, as it also decides dimensions of the cluster.

3.0.2 Filtering support and layout stability

The designed layout algorithm also supports node filtering. This means that user can focus only on selected set of nodes and their neighbours. If layout of filtered nodes is done independently of rest of the graph structure, it can lead to big changes in layout when switching between these different views.

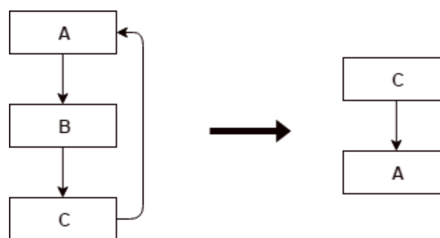


Figure 8 Filtering without stable layout

Node which was at the top of the original layout can get to the bottom of filtered layout, since when considering only filtered nodes and edges, such layout would be considered better as shown in figure 8. These changes then disrupt users mental map, which was formed during work with original graph layout. To avoid disruption of mental map and aid user, the layout should be stable. Stable layout is a layout that follows these two rules.

- The first rule is vertical stability. This means that if node was in higher layer than some other node, then this positioning will be maintained in every possible view.
- The second rule is horizontal stability. This means that if node was on the right of some other node, then it will be always on the right of that node. But unlike the first rule, this rule can be relaxed. It is important that this stability is maintained among nodes of same layer, but nodes of different layers can break this rule in different views. By relaxing this rule it's possible to, for example, better balance child nodes under their parent node. Condition for this relaxation is that while the strict horizontal stability rule can be broken, the change in relative positioning of nodes won't be significant. To avoid significant changes, appropriate algorithms for creating ordering of nodes and assigning coordinates to them have to be chosen.

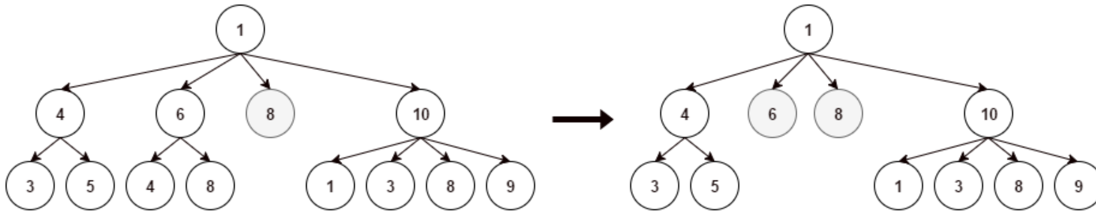


Figure 9 Stable layout in case of tree graph

Creation of a stable graph layout can be easily achieved if there is some limitation of structure of graph, for example input graphs are only binary trees. Stability can be also achieved if values of nodes are mutually comparable. Figure 9 shows graph with tree structure and comparable values of the nodes. In this case, creating ideal layout is trivial. Even if a structure of the input graph is not a tree, the layout can be created as result of sorting of nodes. But this method promotes stability of layout at expense of aesthetic quality of the layout itself. Creating stable layout which also tries to minimize edge crossings and maintain other aesthetic criteria for an arbitrary graph, where nodes have arbitrary values, cannot be solved by these simple methods. To be able to create stable layout, the algorithm has to know the structure of whole graph even in case if only subgraph is currently visible, otherwise it cannot maintain stability throughout changes in filtering. This means that the algorithm always has to get whole structure of the graph and each node and edge gets indicator showing if it is visible in current view, or not. The algorithm can then repeatedly create same structure where relative positions of nodes are maintained and then separately assign real coordinates which are specific to currently generated view.

The need to be able to recreate always same relative ordering creates limitation on used data structures and algorithms. No algorithms that use some form of random selection can be used. Also no changes in data structure are allowed if they depend on visibility of some element and could affect relative ordering. Another limitation is usage of data structures that have unstable order of returned elements when iterating over them. Namely sets and maps are problematic if used wrong. Especially when using iterative heuristic approaches, since even small change in input data can lead to very different results after several iterations.

3.1 Layout algorithm

Layout creation will use an inner layouting algorithm to create a layout of nodes on the same level, for example all subnodes of some cluster and even the nodes on the top

level itself. The outer layout algorithm doesn't create a layout on its own, but rather prepares clusters to be processed, as the layout of each cluster has to be done separately. The outer layouting algorithm will create separated subgraphs for each cluster and use the inner layouting algorithm to create these layouts, then it will use the inner layouting algorithm to create layout of clusters themselves. After this is done, the outer layouting algorithm combines these layouts to final layout. The order of creating these layouts is fixed, since in time of creating cluster layout, the dimensions of clusters have to be already known. Also since the outer layout doesn't do layouting itself it's possible to use the inner layouting algorithm independently. If the outer layouting algorithm is used the final layout will be created with consideration of clusters. If the inner layouting algorithm is used directly, the clusters will be ignored and resulting layout will be created solely with consideration of edges.

3.2 Data structure

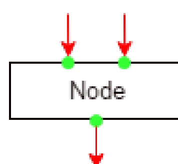


Figure 10 Graph data structure

The algorithms work with following input data representations shown in figure 10.

- Node is represented by its coordinates, dimensions, visibility and by being associated to some cluster. The node also has ports, these are places where edges begin or end. The layout algorithm representation of node then adds informations about assigned layer, relative position of this node in its layer. The layout node doesn't use ports, but maintains lists of child and parent connections to other nodes. Also layout nodes themselves don't maintain 1:1 ratio with nodes of the input graph as the algorithm can create some dummy nodes during layout creation.
- Port represents endpoints for connectors. Each port has assigned its node and relative position from upper left corner of its node. The layout algorithm uses ports only in final phase, when routing of edges is being done.
- Edge - Each edge leads from one port to another one and has set of points through which resulting polyline should be drawn. Edges, just like nodes, have an indicator of visibility. The layout algorithm encapsulates this edge to connection which has parent and child node directly without ports. Some of these connections may also have an indicator of being reversed edge. This indication is used only during layout creation and doesn't affect the input graph. Connections are also not mapped 1:1 to input graph edges, but one connection can represent only part of an edge.

3.3 Outer layout algorithm

The outer layout algorithm takes clusters of the graph and their nodes. For each cluster, it creates data structure which contains all nodes of the cluster and all its edges between nodes of the cluster. All edges which are between nodes of different clusters have to be split. Splitting of such edge going from cluster A to cluster B is done by splitting it to three parts.

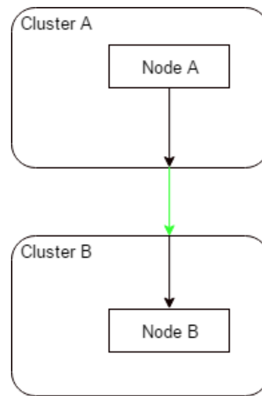


Figure 11 Splitting of edge by the outer layout

As shown in figure 11, the first part is the edge from the Node A in the cluster A to a border of the cluster A, the second part indicated by the green line is edge between the cluster A and the cluster B and the third part is edge from a border of the cluster B to Node B.

After the preparation of the clusters is done, the inner layout algorithm is called to create layout of each cluster. After this is done, the inner layout algorithm is called also on the clusters themselves. At the end, the algorithm joins edges which have been split back together.

3.4 Inner layout algorithm

The inner layout algorithm creates the layout of nodes itself using Sugiyama's framework. In this section there is a description of selected implementations of its steps.

3.4.1 Removing cycles

Removing loops can be done directly. These edges are removed from the graph data structure and maintained separately since they wouldn't affect the resulting layout but only increase the running time.

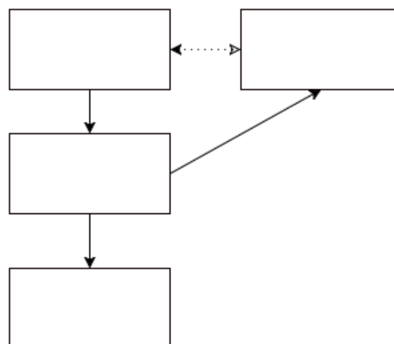


Figure 12 Removing cycles from graph

Then removal of longer cycles begins. Since the input graph has oriented edges, only oriented cycles have to be removed. And these cycles can be broken by changing direction of any edge in that cycle. As shown in figure 12, the edge causing cycle has its direction reversed. But every change of direction of an edge results in back edge

in final layout. To maintain the aesthetic criteria of hierarchical layout, the number of back edges should be minimized. Finding minimal number of edges that have to be reversed in order to break all cycles in directed graph is called minimum feedback arc set and is proven to be NP-hard task[7]. To be able to create layouts in real time, a heuristic algorithm has to be used. The possible heuristics have two parameters by which they can be compared. The first is time complexity and the second is upper bound guarantee specified by multiplying constant. This constant shows how many times more edges will be reversed than in ideal solution in the worst case. One of the possible heuristics is using a simple depth first search, which can be easily modified to detect cycles and break them. The main advantage of this heuristic is linear time complexity. The disadvantage is that this method doesn't guarantee any upper bound on number of reversed edges. These problems can be expected especially when a lot of edges are part of multiple cycles. This heuristic is selected primarily due to its speed and also since expected number of cycles is relatively small compared to size of the instances. In case of using the algorithm on, for example, tournaments, this part of the algorithm should be replaced with some other heuristic which, while being slower, will create less back edges.

3.4.2 Assigning layers to nodes

When assigning layers, there are several aesthetic criteria to consider. The first is minimizing number of back edges. This essentially means that each child node should be in a higher layer than all of its parent nodes. The only exception to this rule are edges which have been reversed in previous step of the algorithm. The second criterion is a compromise between having minimal height of the layout and minimizing width of the layout. If width of node is significantly bigger than width needed for edge, which is usually the case, then overall width of the layout can be decreased by spreading nodes across several layers. This criterion is also not purely aesthetic. Having more layers can lead to more edge crossings when using heuristic algorithms. This is due to each layer having a chance to create suboptimal relative ordering of nodes. It also means that changes in layout have to be propagated through more layers.

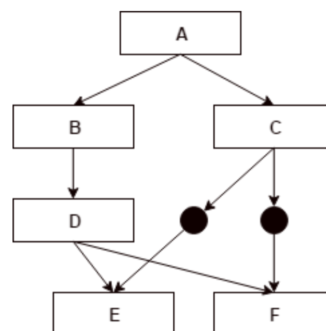


Figure 13 Assigning minimum layer to nodes

Assigning minimal possible layer to each node doesn't necessarily have to lead to minimum of dummy nodes as shown in figure 13. By moving node C into higher layer, number of dummy nodes could be decreased. The optimal layering can be found by using integer linear programming or by static scheduling [8]. There are also many heuristics that try to find reasonable compromise between width, height, number of dummy nodes and performance. The limiting factor in this use case is real time computation. Running time of some of possible algorithms can reach seconds even for

instances smaller than one hundred of nodes [9]. On the other side, the longest-path algorithm creates layering in linear time while maintaining reasonable properties. This algorithm also guarantees that resulting layering will have minimal number of layers which can lead to wider layout.

The longest-path can be implemented by assigning balance to each node and using a queue. The initial balance is equal to indegree of node. Edges of nodes, which have initial zero balance, are added into queue and nodes themselves have assigned first layer. Then, edges are iteratively picked from queue and if child node has smaller layer than parent node, then layer of child node is updated and balance decreased by one. If balance reaches zero, then it indicates that all parent nodes of the current child node have been visited and the child node itself has its final layer assigned. Child connectors of this node are then added into the queue. After the queue is emptied, all the nodes have its layers assigned.

3.4.3 Proper layering

Another problem that exists in context of assigning layers is called proper layering [1]. The layering is called proper if edges are only between nodes in neighboring layers.

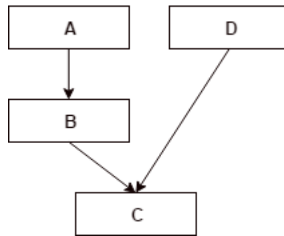


Figure 14 Layout without proper layering

Figure 14 shows layout that is not properly layered. If the layering is not proper after assigning layers to nodes, it has to be transformed into one. This is done by transforming any long edge, meaning an edge which connects nodes that are not in neighbouring layers, into path of dummy nodes and edges.

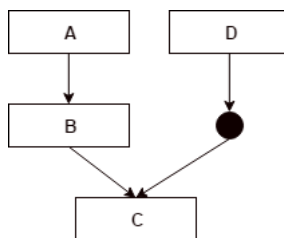


Figure 15 Layout after proper layering

Figure 15 shows layout after being transformed to be properly layered with black node indicating dummy node. While this transformation is easy to do, long edges bring two disadvantages. The first disadvantage is simply a performance issue. Since transformation of long edge increases size of the instance for which layout is computed by adding more nodes and edges. The second disadvantage is increasing number of bends of visualized edge, this is problematic especially if there are many edges, as it makes edge tracking harder.

This step can be easily implemented by iterating over edges of nodes and checking if the edge connects nodes in neighbouring layers. If that is not the case, then this edge is transformed into path of edges and dummy nodes. In this case, this transformation is done together with aggregation of edges based on output rules. This is done to avoid creating unnecessary dummy nodes, which would be removed again in the next step.

3.4.4 Aggregate edges

To reduce amount of dummy nodes, it's possible for edges to share them. There are several strategies which can be used. Easiest solution is skipping this step and simply leaving path of edges and dummy nodes for each individual long edge. Or it's possible to create only one dummy node per-node, or even per-port. This allows to decrease number of dummy nodes which speeds up the algorithm.

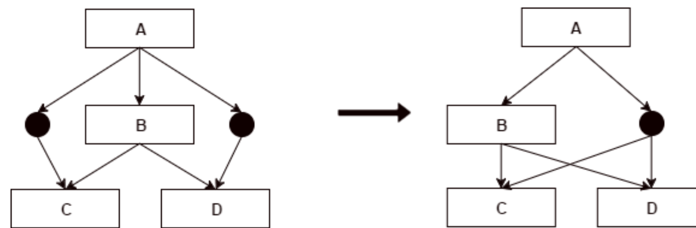


Figure 16 Aggregation decreases number of dummy nodes

Another advantage is that since dummy nodes are shared as long as possible by multiple edges, it decreases number of visual edges user sees and has to keep track of, as shown in figure 16. Both per-port and per-node reductions can also be done based on both input and output part of the edges and all approaches can be combined at the same time. But, when combining, it's important to always combine only those edges which won't create situations where it's impossible to see from which, or to which node an edge goes.

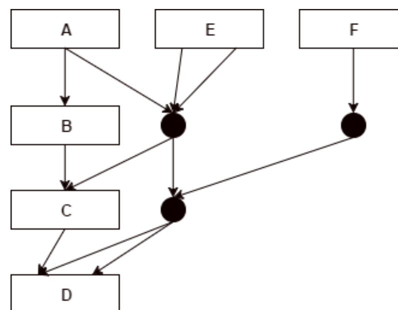


Figure 17 Aggregation per-node from inputs and outputs

Figure 17 shows situation where edges are reduced by per-node reduction based on both input and output. In this case, it's still possible to track which nodes are connected by edges. For example, an edge from the node F to node D joins aggregated edges from nodes A and E only after edges going to node C split from the aggregation. But it's no longer possible to say exactly which port of node E has edge leading to the node C or which edges lead to first port of node D.

While these reductions have advantages, they are not suitable in every use case. Sometimes it may be critical to be able to track exact port to port connections. In

case of the Ideal Graph Visualizer, the ideal setting is aggregating outputs per-port and no aggregation for inputs, since the IGV provides tools to work with this type of aggregation. But other tools may not have this support, or may support some other combination. To provide support for different tools and use cases, the algorithm implements all three aggregation strategies per-node, per-port and no aggregation based on both input and output side of edges. Due to the output aggregation strategy being applied already during transformation into proper layering, the input strategy effect is limited as it will be used only when it doesn't interfere with the output strategy.

3.4.5 Crossing reduction

Edge crossing reduction is problem of finding such ordering of vertices which leads to minimal number of edge crossings. Optimal solution of this problem is NP-complete as it requires to try all node permutations in each layer. The original solution uses barycentric 2-layer algorithm[8].

Implemented algorithm uses similar concept of iterative improvements. There can be between two and ten iterations. The number of iterations is based on a number of real edges, i.e. edges created by transformation into proper layering don't count. Each 200 edges decreases iteration count by one. This allows to create smaller layouts with better quality and also bigger layouts can be created in better time, even though the quality may suffer.

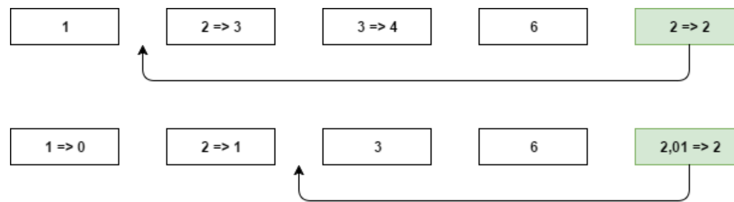


Figure 18 Solving collisions

The crossing reduction is done by placing nodes to initial positions, which represent their relative positions. The algorithm then iteratively goes over the layers several times. During these iterations, each node has its position recalculated. If resulting ideal position is different, node is moved to its ideal position. If the position was already occupied, this state is called collision. Collision is solved by moving node, which was previously at position of collision, one position to the right or to the left based on exact positions each node wants. This shift can result in new collision, so this shift has to be repeated recursively if needed. Figure 18 shows both cases of possible collisions with green nodes being currently placed. The numbers represent wanted position and position after they are placed. For the other nodes, the numbers represent assigned positions before and after placing. While these shifts may be more performance demanding than placing node on closest non-colliding position, they are important. Using more aggressive strategy of placing nodes on required positions and then solving collisions helps to overcome a poor quality of initial placement of nodes and it also prevents layout from getting locked in some permutation.

These steps are similar to the original algorithm, but there are some important differences. Unlike the original concept which uses combination of downward and upward passes through layers. This implementation uses only downward passes, which generally provide better results. This is combined with another important difference. The original algorithm compares always only two layers, but this implementation computes

the ideal position of each node based on both parent and child nodes at the same time. This allows the algorithm to propagate relative positions in both ways, which would be normally done by switching between downward and upward passes.

After the main crossing reduction algorithm finishes, there is an auxiliary algorithm. It goes iteratively over the nodes same as the main algorithm. But this time it checks if each two neighbouring nodes wouldn't be closer to their ideal position if they swapped their positions. Unlike the main algorithm, this swapping is not aggressive, meaning that if both nodes want the exact same position, they won't swap their positions. Another difference is that the node can swap its position only with its neighbour, it cannot jump to its ideal position as in the main algorithm.

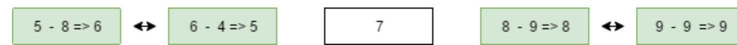


Figure 19 Position swapping

Figure 19 shows these two cases. The numbers represent current position, wanted position and final position after swap. In the left case, the swap is done, but even though the left node would be placed even after the white node if the main algorithm would be used, in this case that doesn't happen. The other case in the figure shows two nodes wanting same positions, no swapping happens in this case. Due to this different approach the layout won't change as drastically as in the main algorithm, but rather if there is possibility to make some final beneficial adjustments, they are done.

While these relative positions created by the ordering algorithm could be used as basis for real positions of nodes, it leads to poor layouts in cases where there are several nodes which share same group of parent and child nodes. Since all these nodes want same ideal position, the algorithm will always do default right shift. This then leads to parent node not being centered over its child nodes, but rather being left-aligned. Another thing to consider is that there may be several subgraphs which are well separated and create big gap in positions. This then leads to unnecessary gaps and resulting layout is too wide.

3.4.6 Assigning coordinates

Due to selected implementation of previous step, assigning coordinates based on positions would lead to poor results. To prevent this, another step is added to assign x coordinate. There are several possible ways to assign coordinates. A very trivial solution is simply placing all the nodes as close to each other as possible and centering them around the middle of previous layer. This solution will create a layout with minimal width, not considering other possible layer assignment, but it will also lead to very significant edge cluttering. Other possible solution is using some uniform form of grid which adds more rigid structure to the layout. This is problematic if differences in width of nodes are big, as it leads to a lot of wasted space. The implemented algorithm uses more complex system to assign coordinates which is combination of these approaches.

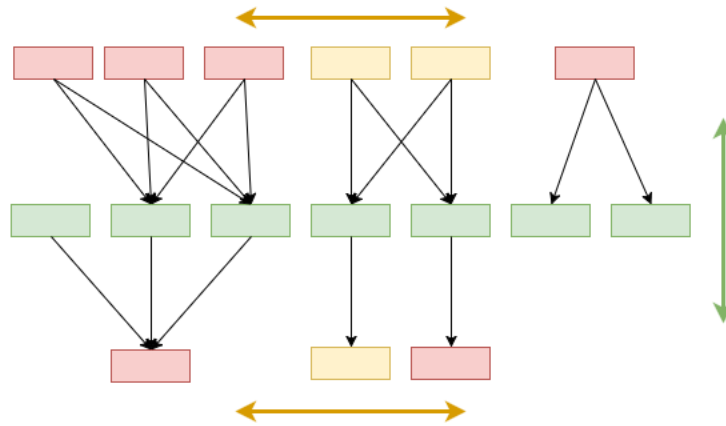


Figure 20 Assigning coordinates to nodes

As you can see in figure 20, the assigning coordinates begins by finding the widest layer (green nodes). Nodes of this layer are then placed using the relative order gained in the previous step and are placed as close together as possible, having only the minimal spacing between themselves. Starting from this layer, the rest of the algorithm is split into two parts. Both parts are identical, only the first is going upward and the second downward as indicated by green arrows.

In each processed layer, groups are created at first. Group contains nodes which want the same position relative to previously processed layer. From these groups, group which wants to be positioned as close to the center of already processed layer is placed first (yellow nodes). Then going to the left and to the right from the placed middle group (yellow arrows), other groups are placed either to their ideal position or if the position is already occupied, the group is placed at the first possible position after that. By assigning coordinates from the widest layer and then having every layer built from the middle to the sides, it is possible to create good layouts.

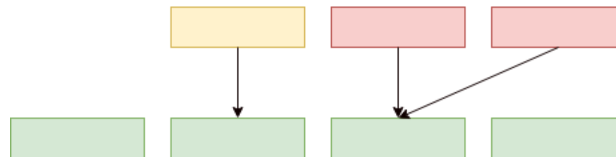


Figure 21 Unbalanced graph

But problems can appear if the graph is heavily unbalanced on one side as shown in figure 21. Then this approach can lead to small middle group (yellow node) being placed first and then big unbalanced group (red nodes) having to be pushed more to the side. To prevent this, it is possible to iteratively sweep over the layers and their groups of nodes, adjusting coordinates of the groups. When doing this, it is good idea to have some threshold to prevent small changes to positions which lead to staircase effect over many layers. This optimization iterates three times, but it's done only if a number of input graph edges is smaller than 300. This limitation was added as it can be rather time consuming to recalculate coordinates.

3.4.7 Edge routing

After previous step, creation of the layout is nearly complete. There are only two problems to be solved. The nodes still need to get y coordinates and also routing of

the edges has to be done. These two problems are connected. While the edges will be routed either directly between nodes, or through dummy nodes, beginnings and endings of back edges may need extra space. Otherwise there is a risk that the back edges will overlap with nodes. At the same time, this required space should be minimized. To do this, for each layer and both directions, a mapping is created. This mapping maps offsets of the back edges. When a new back edge is added, it's checked against already added back edges and it gets assigned the lowest offset where there is no collision. This approach is very fast, but there is no guarantee that this mapping will result in minimal extra height.

After assigning the offsets to the back edges, it's possible to assign y coordinate to the nodes as needed gaps are now known. After this, routing of the edges is simple. Lines, representing a normal edge, are created by navigating through the dummy nodes. The back edges differ only in the first and the last step of path, where the previously assigned offset is used.

4 Implementation

In the previous chapter, the description of implemented algorithm was made. This chapter contains description of tool IGV which is used to visualize results of layout algorithm. This description is focused on features that allow working with graph itself. The second half of chapter then contains code structure and layout algorithm features that are not part of Sugyiama's framework.

4.1 Ideal Graph Visualizer

The Ideal Graph Visualizer [11], or the IGV, is tool created by Thomas Wuerthinger and is part of OpenJDK platform. This tool is used to visualize bytecode flow charts in different steps of optimization. The IGV allows to load bytecode graphs in XML or BGL format. These graphs can then be visualized. Users can manipulate with these visualizations, for example zoom or change visibility of the nodes. They can also use special language to write filters, these filters can change style of both the nodes and the edges.

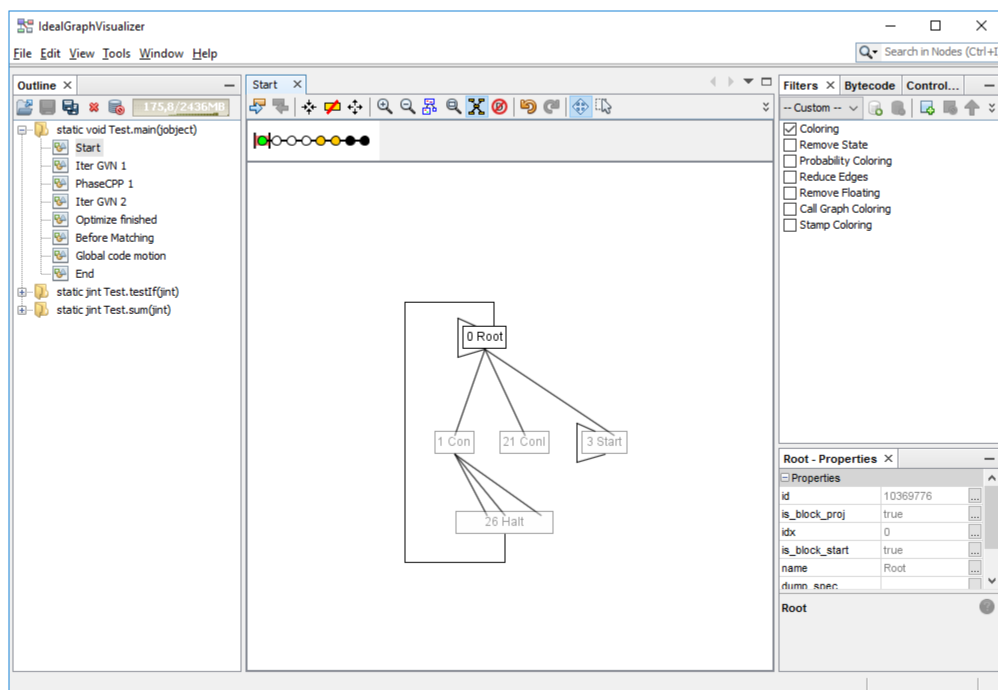


Figure 22 IGV user interface

This means that IGV provides loading of graph to graph structure inside the application and it also provides renderer of the final graph layout. The original application has its own layout algorithm. This algorithm doesn't support changes in visibility of nodes and edges. To simulate these changes, the layout is used only with nodes and edges that are visible in current view. This leads to strange anomalies. For example

node which has been at last layer in the original graph layout moves into first layer upon making some other node as invisible.

The IGV itself is built as modular application using Netbeans 7.1 platform [12]. The code is written and compiled using compliance level of Java 1.8. This means that to create a build of the application, user has to have this platform available. Netbeans platform 7.1 is a part of a standard Netbeans IDE of version 7.1, but it is possible to use newer versions of Netbeans and required dependencies of IGV will be automatically downloaded. In some cases, Netbeans fail to download some of the dependencies, but it doesn't generally lead to build failure. If the missing dependencies are not satisfied in runtime, some of the modules will be disabled. This should not prevent from visualizing the graphs, but it may affect some other features like filters and manipulation with the graph.

4.1.1 Features of IGV

The IGV implements several features that make it a good tool to showcase the new layout algorithm. There are also features which help users to navigate through the layout. While the new algorithm can be used even without these features, they do significantly increase readability of the graph layouts.

It's also important to note that the IGV doesn't provide full support of these features for all of the possible combinations of edge aggregation strategies of the new algorithm.

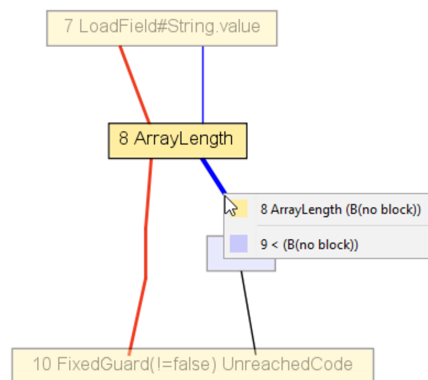


Figure 23 IGV - Edge context menu

Users can open a context menu by clicking on an edge by right mouse button as shown in figure 23. Context menu then shows parent node and all child nodes if connector is aggregation of edges. Users can also click on any of these nodes to immediately center view on selected node. This feature is supported only if the edges are aggregated based on having the same output slot, or there is no aggregation at all. This is due to this setting being the default behavior of the original layout algorithm. If the new layout algorithm is used with different settings, this feature won't work correctly.

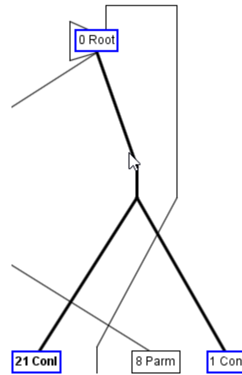


Figure 24 IGV - Connectors highlighting

When a user points at some edge, it will get highlighted as shown in figure 24. This feature allows to track edges even in layouts with many edges and edge crossings. Same as in the previous case, this feature is supported with same aggregation rules as in the original layout algorithm, that is edge aggregation based on output port, or no aggregation.

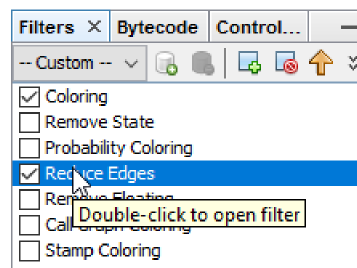


Figure 25 IGV - Filters

IGV allows users to use and even create their own filters. These filters can be either purely visual, changing only colors of nodes and edges, or they can be used to remove some type of edges from graph. This second type of filters will force recreating graph layout, while first type doesn't.

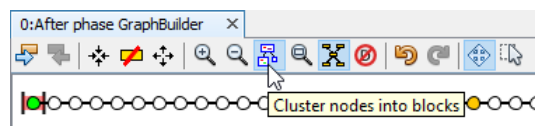


Figure 26 IGV - Splitting graph into clusters

IGV supports hierarchy of nodes. By swithing button on toolbar as shown in figure 26, IGV will force recreating layout with consideration of clusters.

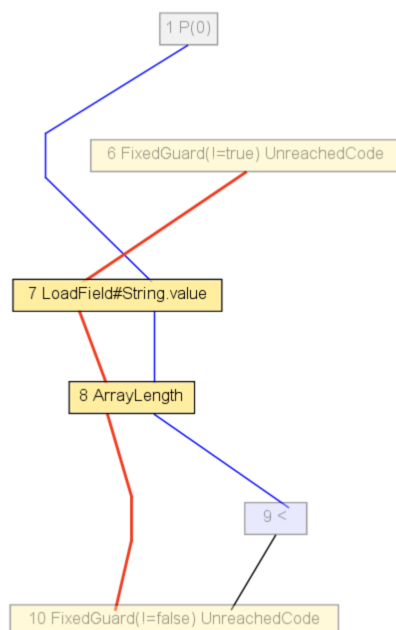


Figure 27 IGV - Visibility of nodes

Users can select which nodes they want to focus on. They can either use context menu to remove focus on some node, or use double click with mouse to switch focus on and off. Figure 27 shows situation where there are two opaque nodes which are focused and four are semi-transparent. Semi-transparent nodes are neighbours of at least one focused node. From perspective of layout algorithm, both focused nodes and their neighbours are considered visible nodes. The layout algorithm doesn't have information about which nodes are part of focused group.

4.2 Code structure

IGV is created as modular software with 17 modules. Since there is already layout algorithm, there are several relevant modules.

- Layout module contains interfaces of graph elements, namely node, port, edge and cluster. Next, there is interface for layout algorithm. This interface is very lightweight and describes only how graph structure is passed to layout algorithm. The graph structure is passed using class implemented in same module.
- HierarchicalLayout module contains the original layout algorithm, same as the new algorithm.
- Graph module contains implementations of interfaces defined in Layout module. While layout algorithm uses interfaces from Layout module, these are real implementations of those interfaces.
- View module, among many other classes, contains class DiagramScene, this class handles manipulation with work area, where the graph is visualized. It is also place where layout algorithm is used to recreate the view.

Since there was already layout algorithm in the application before, some of the code is reused in the new layout algorithm, especially interfaces. This doesn't prevent reusing the layout algorithm since interfaces are very minimalistic.

4.2.1 Implemented code

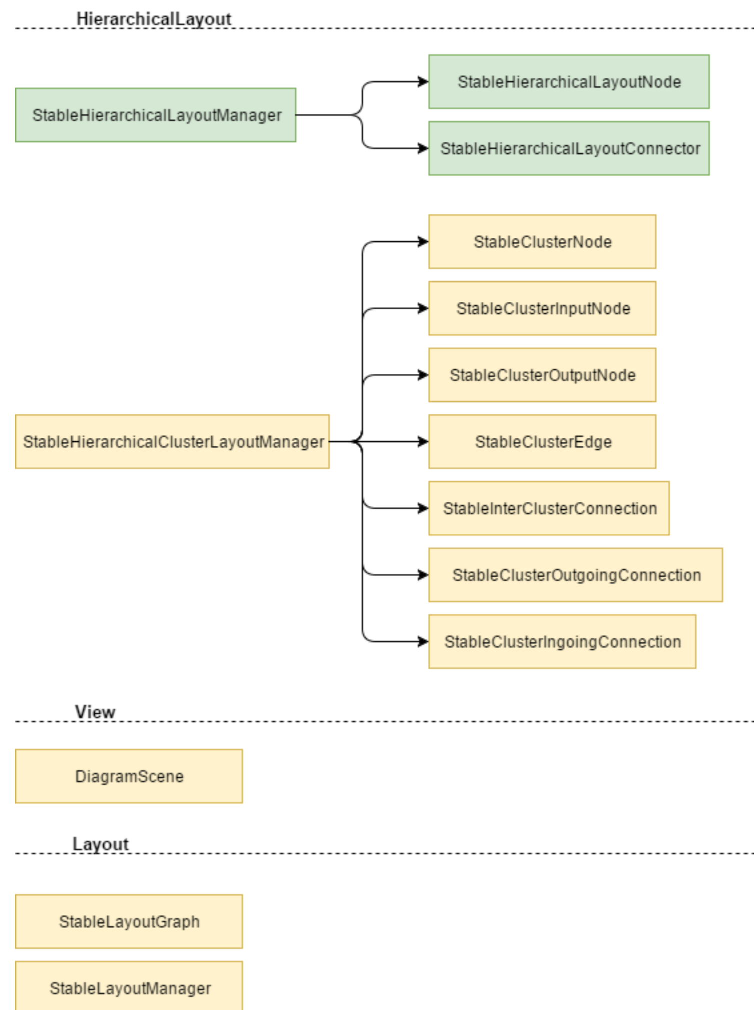


Figure 28 Direction of usage

As shown in figure 28. The yellow classes represent classes which were created based on classes in the original solution. These classes contain varying levels of modifications. In case of DiagramScene, the original one is used with modification of used layout algorithm. It also contains a change in way the graph structure is passed to layout algorithm to add support of visibility. The StableHierarchicalClusterLayoutManager class represents the outer layout algorithm presented in previous chapter. This reused class has some modifications compared to the original one and it also uses many other different classes from the original solution. These classes are used purely to store a structure of an input graph and have minimal modifications to add support of layout stability. The green classes are the newly created classes of the solution created in this thesis. The StableHierarchicalLayoutManager class is the implementation of the Sugiyama's framework as described in previous chapter. This implementation uses its own representation of nodes and edges (connectors). Layout module then contains StableLayoutGraph which is modification of original class used to pass graph structure to layout algorithm. StableLayoutManager is an interface used for new LayoutManager classes.

4.3 Implementation features

The new algorithm supports several features that may affect visual result of layout algorithm.

4.3.1 Dynamic gap between layers

The layout algorithm uses dynamic gap between layers. This gap is based on angles of edges between those two layers and it's possible to set minimum and maximum gap. This technique is introduced to reduce edge cluttering which happened in the original layout.

Figure 29 Original layout edge cluttering

Figure 29 shows situation in the original layout. Since gap was static, it was set to low value, to reduce height of graph. But in some cases it can lead to having many edges with very similar angles at the same place. This then reduces edge traceability.

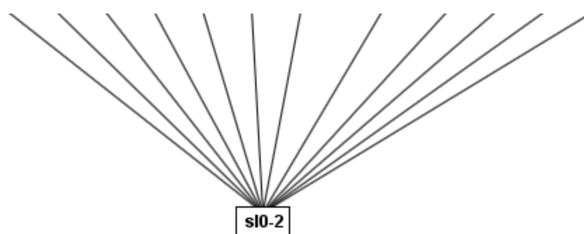


Figure 30 New layout with dynamic gap

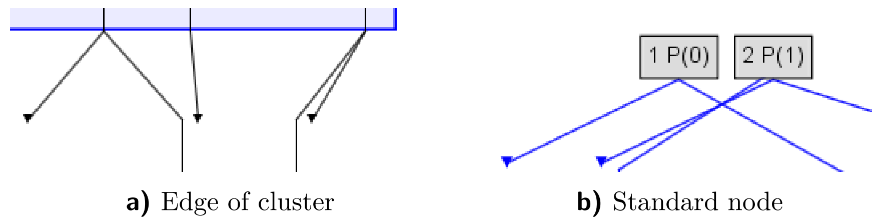
Figure 29 shows same case with the new layout. Since very flat edges have been detected, gap between layers was increased to help users with edge tracking.

While this technique can help with edge traceability, it also increases overall height of graph. Due to performance issues the algorithm doesn't set height of gap by searching for edges that are very close and then finding minimal height needed to get proper angle between those two edges, which would be more optimal solution. Instead heuristic algorithm is used. This algorithm uses differences between horizontal positions of edge beginning and ending to find proper height. Due to heuristic nature of whole layout algorithm, this can possibly lead to unnecessary big gaps between layers. By setting maximum for this gap, it's possible to diminish worst cases. It also allows more easily to shift between better edge traceability and graph layout compactness.

4.3.2 Interrupting too long connectors

Same as the original layout algorithm, it's possible to set maximum difference in layers, for which edges will be still drawn. Reasoning behind this technique is that too long edges are hard to trace and in case of IGV, it is possible to use edge navigation (see 4.1.1). In case of using the algorithm in tool, which doesn't support any other suitable navigation technique, it's possible to turn this feature off and always have all edges drawn. Due to the need of maintaining stable layout, changes in this setting have

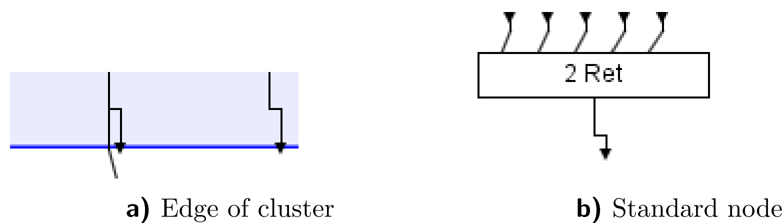
rather minimal impact on performance of the algorithm itself. Implementation of this technique also changed.



a) Edge of cluster **b)** Standard node

Figure 31 Interrupting long edges in old layout

Figure 31 shows how were long edges interrupted in the old layout algorithm. The interrupted edges were led to auxiliary node placed in next layer. This has advantage that edge will be more visible. On the other side if created ordering places this auxiliary node too far from the original node, then resulting edge will be very long and may add unnecessary edge crossings. It may also worsen problems with edge cluttering.



a) Edge of cluster **b)** Standard node

Figure 32 Interrupting long edges in new layout

The new layout algorithm uses different approach. As figure 32 shows. In case of clusters, edge is interrupted on the border of cluster with distinctive bend. In case of normal nodes, outgoing edges act similarly as in case of clusters. Incoming edges are placed in such a way that prevents their crossings. Both incoming and outgoing edges do not lead to previous or next layer, but instead they are placed in gap between layers. Edges are also bent in such a way that prevents them from being overlapped by other edges. This helps to reduce edge crossings and edge cluttering, even though it can make it harder to work with these edges if node has many edges in the first place.

5 Testing

Automatic testing of the layout algorithm is quite complicated because it's hard to evaluate the results algorithmically. Testing of a quality of the layout has to be done manually. It's also problematic to create specific graphs for testing since the IGV uses mainly a binary format BGV to store graphs. It's also possible to use XML format, but this leads to problems with loading clusters. Fortunately Oracle, which uses the IGV, provided huge number of graphs. These graphs have anywhere from a few to several hundred thousands nodes and edges.

The visualized graphs will be discussed in the following chapter. In this chapter, the performance of the algorithm is presented on several graphs with sizes from hundreds of nodes and edges to tens of thousands. The time is measured using Netbeans profiler.

5.1 Performance overview

ID	Nodes	Edges	Nodes[1]	Edges[2]	Time (ms)	Time[3] (ms)
1	272	436	1,706	3,534	**15.4**	9.98
2	650	1,027	6,553	15,483	**10**	20.1
3	949	2,121	19,763	42,188	**30.1**	30.1
4	2,017	3,367	16,242	49,774	**10**	10
5	11,918	27,877	326,900	1,479,396	**1,258**	575
6	2,278	10,081	191,713	1,524,173	**2,352**	746
7	7,220	15,479	32,179	3,530,520	**859**	390
8	17,286	33,701	212,932	9,557,330	**2,514**	512
9	19,997	36,610	290,932	29,532,825	**8,767**	705

[1] Node count including dummy nodes
[2] Edge count with split edges to achieve proper layering
[3] Time of the old algorithm

Table 1 Time needed to create layouts

Table 1 shows time needed to create a layout for some graphs. First thing that can be noticed is that the new algorithm is slower than the original one. The main reason behind this is that the new layout algorithm has to maintain stability. Because of this, the new algorithm has to truly create paths from dummy nodes between all nodes. The original algorithm can do optimization. In the previous chapter, a feature that allows to avoid drawing edges that are too long was presented. This feature allows the old algorithm to actually convert too long edge into only two edges with two dummy nodes, one at the beginning and the other at the end. This can drastically decrease size of instance which is processed by the algorithm. The new algorithm cannot do this, as it would break the connection between these nodes, which would cause instability of layout. This is due to changes in visibility of layers that can change which edges are too long and won't be drawn. As the table shows, the number of needed nodes and

edges by the algorithm can grow very fast. For example graphs 4 and 6 are very similar in number of nodes, but a size of the instances is very different.

Another very significant result is graph 7. Even though the number of edges needed by the algorithm is very high, the overall time needed is smaller than in case of a instance with less than half of the edges. The main difference in this case is much smaller size in terms of nodes. This makes especially the crossing reduction step of the algorithm much faster, because while the ideal position of a node is computed using all the edges, the lower number of nodes in this step simply means that there is less possible collisions and also less time is needed to maintain a data structure.

Considering the results, the algorithm can be used to create layouts in real time. But unlike the original algorithm, it doesn't scale as well. This is due to the need to maintain all dummy nodes. Still, the algorithm can be used to create layouts with several thousands nodes, but an edge-node ratio has to be considered. A graph with relatively few nodes and many edges is a good indicator that there may be also a lot of long edges, which will have to be transformed into paths with dummy nodes.

5.2 Running time of individual steps

Measuring running time of individual steps is important in order to find the places, that need to be optimized.

ID	Time (ms)	Dummy nodes	Edge crossings	Coordinates	Edge routing
5	1,258	200	462.2	161	301
6	2,352	153	213	52.1	1,879
7	859	240	260	141	146
8	2,514	1,111	706	261	304
9	8,767	4,427	875	395	242

Table 2 Time needed to perform individual steps of the algorithm

The table 2 shows time needed to perform individual steps of the algorithm. This table is missing some steps, particularly cycle removal and assignment of layers. This is due to these steps being very fast and borderline unmeasurable with times about 10ms. Also first four graphs were omitted because of the same reason.

As table shows, most of the time is needed to create dummy nodes and redirect the edges. Unfortunately, these steps are mandatory to provide layout stability. But considering linear time complexity of layer assignment which was below threshold of the profiler. It might be possible to optimize dummy node creation time by using more complex layer assignment algorithm which would generate a better layer assignment. This would then require less edges to be transformed into dummy node paths and thus decrease running time of this step.

Another more time consuming step is reduction of edge crossings, although this step seems to scale well with a size of an instance. Tested graphs also contain one anomaly, the graph 6 has much longer time needed to route edges. This is due to high number of interrupted edges connected to one node. To prevent having unnecessary edge crossings, these edges are sorted, which is very time consuming action, when considering sizes of the instances. While this sorting doesn't commonly take much time, it can become a problem in some edge cases.

5.3 Running time with and without clusters

ID	Time With Clusters (ms)	Time Without Clusters (ms)
5	1,063	1,258
6	2,098	2,352
7	1,013	859
8	2,369	2,514
9	6,739	8,767

Table 3 Running time with and without grouping nodes to clusters

The table 3 shows comparison of time needed to create layout with consideration of clusters and without it. With exception of the graph 7, layouts with clusters generate faster. While this may seem counterintuitive, it is important to understand, that layout with clusters is created by splitting the instance into smaller instances of layouts, which are created separately. Having more smaller tasks then helps with non-linear algorithms. Also smaller instances means less dummy nodes and less chances for collisions, when creating relative ordering of nodes. At the same time splitting the graph into these subgraphs is done in linear time. The only downside is, when there are many edges connecting nodes in different clusters, or there are many clusters with only a few nodes. Then the running time can increase.

5.4 Conclusion

Tests show that it's possible to create layouts for even few thousands of nodes and edges quickly. But even graphs with similar count of nodes and edges may have very different running time. This makes it hard to find some maximum size of graph that can still be worked with interactively. The main problem is creation of dummy nodes, which can be partly solved by using layouts with clusters.

6 Results

This chapter contains a samples of different layouts created using the new layout algorithm. Sample layouts shown here belong among the smaller ones due to space constraints. The chapter is divided into four parts. In the first part, a layout created when some of nodes are not visible is presented. In the second part, there is a comparison of the new layout algorithm and the old layout algorithm from IGV. The third part shows results when using a different edge aggregation strategies. In the last part, there are shown some flaws of the new layout algorithm, which could be fixed to achieve better resulting layouts.

6.1 Node filtering

This section contains two figures of the same graph, first figure shows the graph with all nodes visible, the other figure contains the graph with only a few nodes visible.

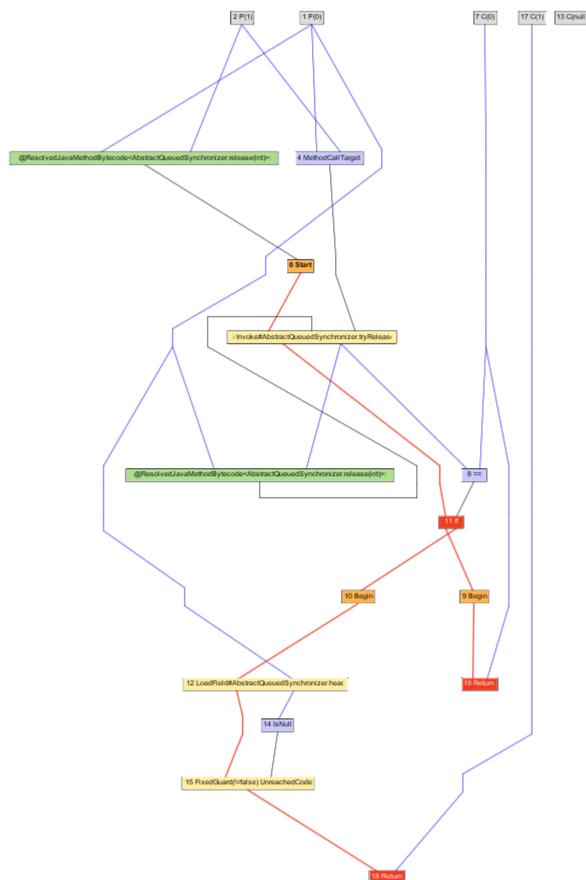


Figure 33 Full layout of graph

Figure 33 shows whole layout of sample graph. This graph contains one back edge and also two cases of edge aggregation. It also shows usage of dynamic height of layers.

The gap between layers changes dynamically to avoid low angles of the connectors. This is done to make graphs more readable, the downside of this is making the layout taller. Since dynamic gap is computed by heuristic algorithm, it can sometimes create unneeded gap. In case of this layout, the gap between second and third layer could be smaller. On the other side the bigger gap between fourth and fifth helps, as there are several nearly parallel edges nearby. Generally, there are more problems if graph contains very wide nodes like the two green nodes in this graph.

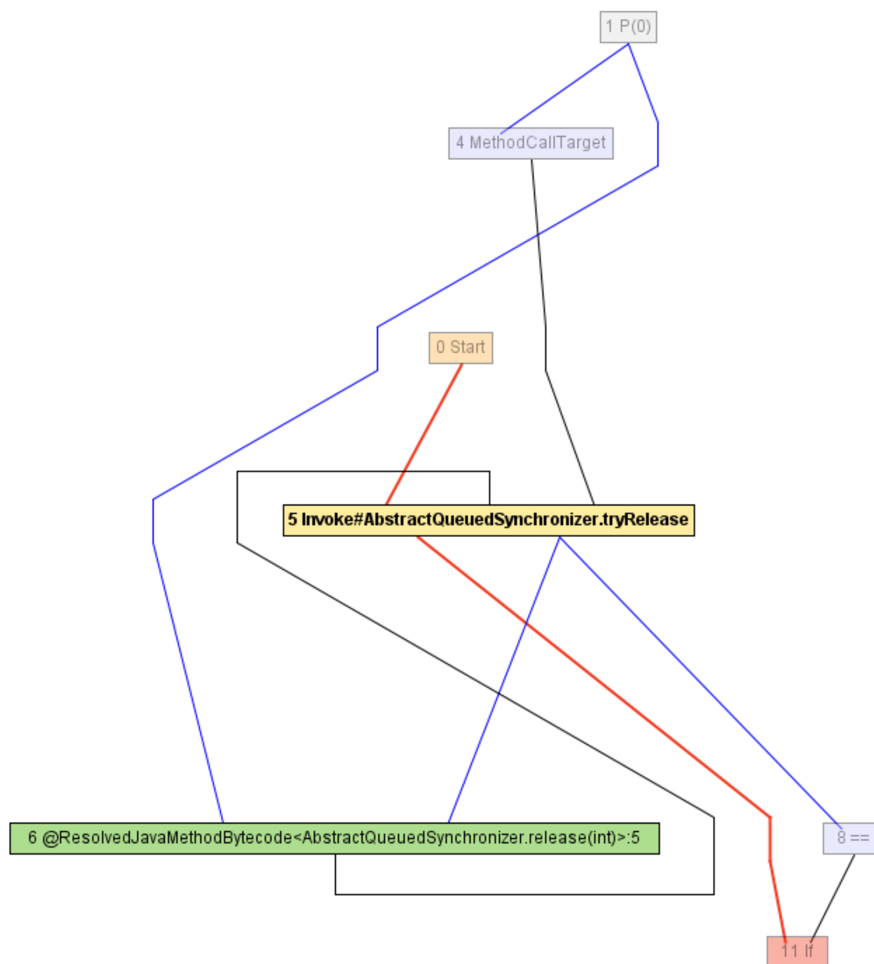


Figure 34 Changing visibility of graph nodes

The Figure 34 shows the same graph as in previous figure 33. But this time with focus on two nodes, so only their neighbouring nodes are shown. In this case, the layout looks very similar to the original one. This may not always be the case. When assigning coordinates, the algorithm takes into consideration even nodes which are not visible, but these have zero width. This changes width of whole layer and assigned coordinates then differ. In some cases, these differences can add up and have relatively big impact, yet again the very wide nodes are more problematic.

This layout also shows case where horizontal relative positions of nodes in different layers change. The two nodes at the top of the graph changed their relative positions. This is due to wide green node no longer being visible. In this case the green node is still placed on the left of the blue node, but has zero width and so it doesn't drag the

gray node in the first layer to the left. The original relative positions wouldn't make sense in this layout, but by allowing this change it's possible to create layout which looks better and it also doesn't disrupt users mental map.

6.2 Comparison with the original algorithm

In this section there are presented again one graph and two figures. In the first part, there is the graph layout created by new algorithm, in the second part is the original layout.

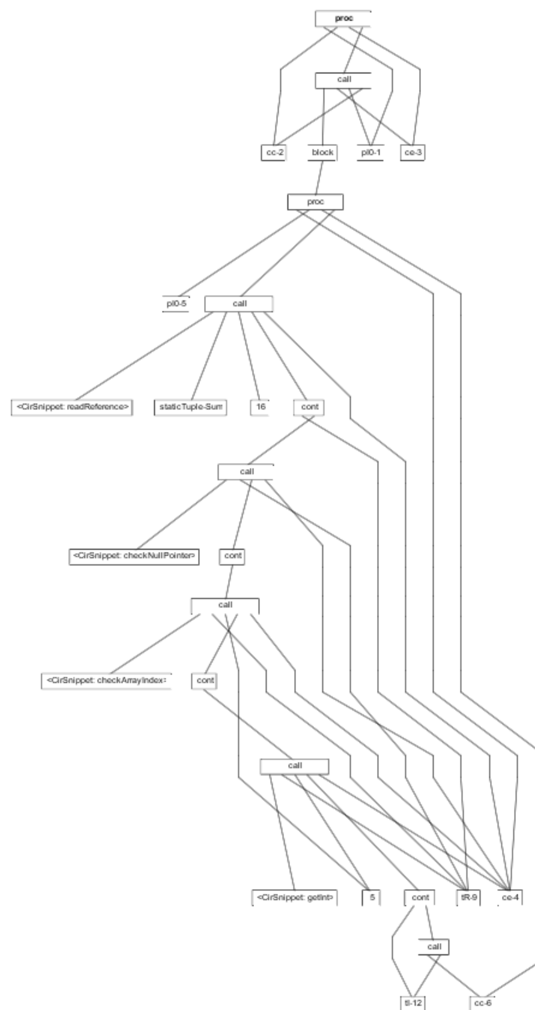


Figure 35 Layout of graph using the new algorithm

The Figure 35 shows one of the more complex layouts. It is obvious that some edges can be really long, going over many layers. The overall shape of layout is pretty good. Even by looking at the overall shape of the layout it's still possible to recognize which nodes are connected. The dynamic height of layers make the layout taller, but it also helps with edge traceability. For example even though there is many connectors crossings in the bottom part of the layout, it is still possible to follow where each of the connectors is heading.

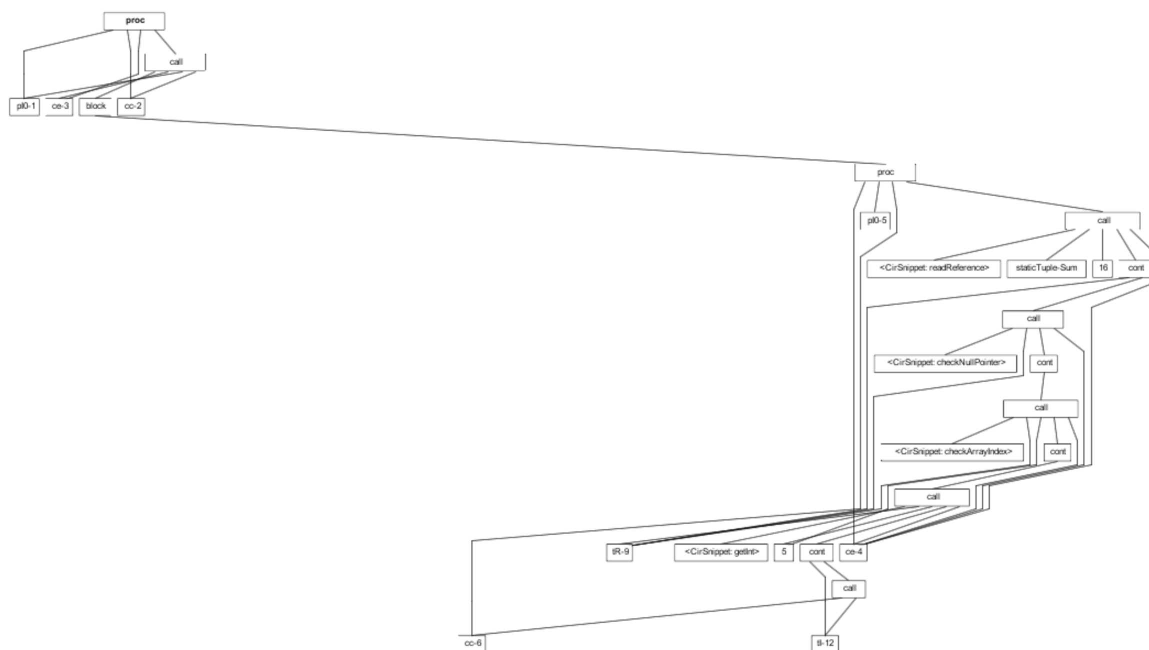


Figure 36 Layout of graph using the old algorithm

Since the original application had its own algorithm, it is good to compare the results of both algorithms. Figure 36 is same graph as in Figure 35. The first thing anyone will notice is that the original layout is wider. In this case the extra width is also illogical as there is no reason for the gap between the upper part of the layout and rest of the graph. Another problem is at the lower part of the layout. The original layout uses static gap between layers. This gap is quite small, since with static gap, it's better to try to minimize overall height of graph. Trying to find some sort of ideal gap size that would fit all possible graphs would be impossible. But in this case, this leads to edges in the lower part of the graph to be completely untraceable.

The old algorithm also tries to have spaces between objects as small as possible, which makes situation even worse. Although it helps to minimize the overall width of the layout. It also increases edge cluttering, especially when there are long edges as in this graph.

This example shows that by trying too much to achieve some aesthetic criteria, which is typically considered to be positive, it is possible to actually make the resulting layout worse.

6.3 Edge aggregation strategies

The new algorithm also supports different aggregation strategies which were described in chapter Design (3.4.4).

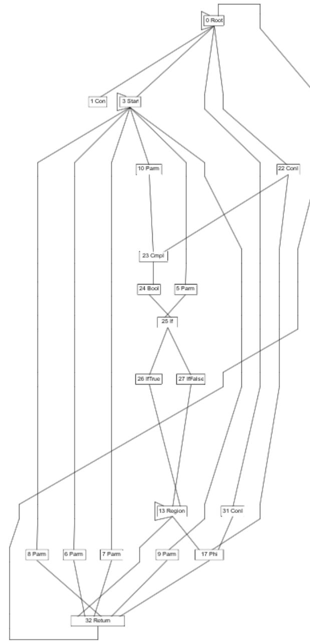


Figure 37 Layout with no aggregation

The figure 37 shows a layout with no edge aggregation. Quality of this layout is quite good. The only problem is the back edge crossing all the edges, even though a solution where this doesn't happen is trivial. As for the edges, while this layout is simple enough, it may be quite hard to track for example all the edges coming from the node in the second layer. Especially since one of these edges, the one on the right, splits off the main group. This could be problematic in bigger layouts.

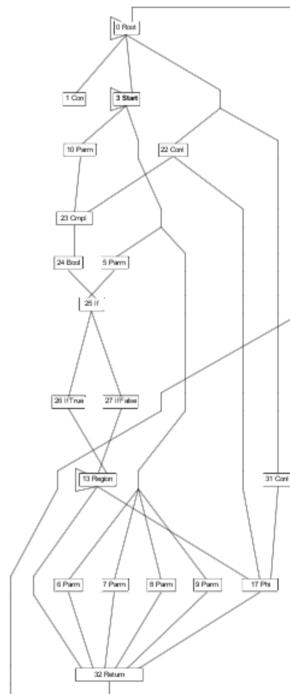


Figure 38 Layout with port-based aggregation

The figure 38 shows a layout of the same graph as was in figure 37, but this time with port-based edge aggregation for both inputs and outputs. The first thing that needs to be mentioned is that layout stability is not guaranteed across different aggregation rules. Another thing that can be easily noticed is that the back edge crossing over the layout remained. But this time, the number of edges is smaller. This also allows the layout to be thinner, which is additional benefit, as it means that users have to scroll less, which reduces problems with navigation.

6.4 Layout with clusters

One of the most important properties of the new layout algorithm is support for grouping nodes into clusters. Since creation of a layout with clusters is very similar to creation of a standard layout, results will be generally quite similar. But there are still some differences.

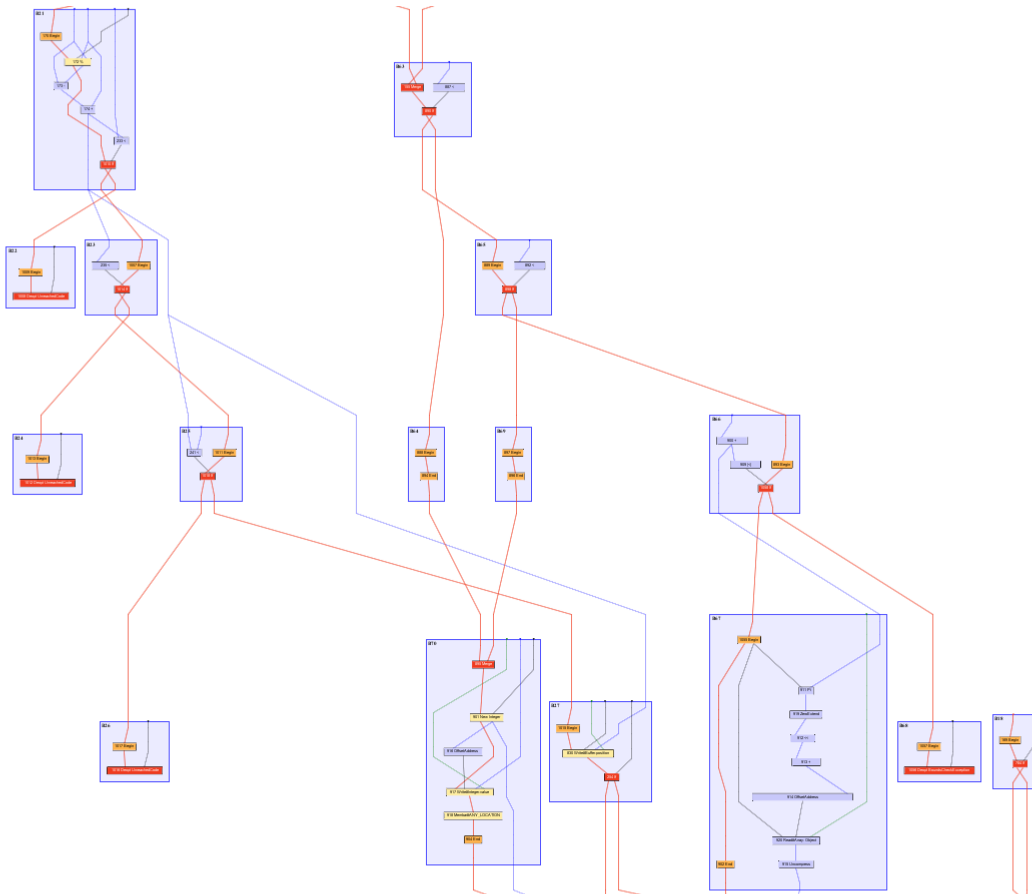


Figure 39 Part of layout with many clusters

The figure 39 shows part of a layout with nodes placed into different clusters. Since this figure shows only a part of much bigger layout, many of the edges are actually not drawn outside of the area of the clusters due to interruption of too long edges. Another thing that can be noticed is that the clusters are vertically centered and edges leading to these clusters are routed in such a way that they won't overlap with neighbouring clusters. This feature is not exclusive to clusters, but these do generally have different heights.

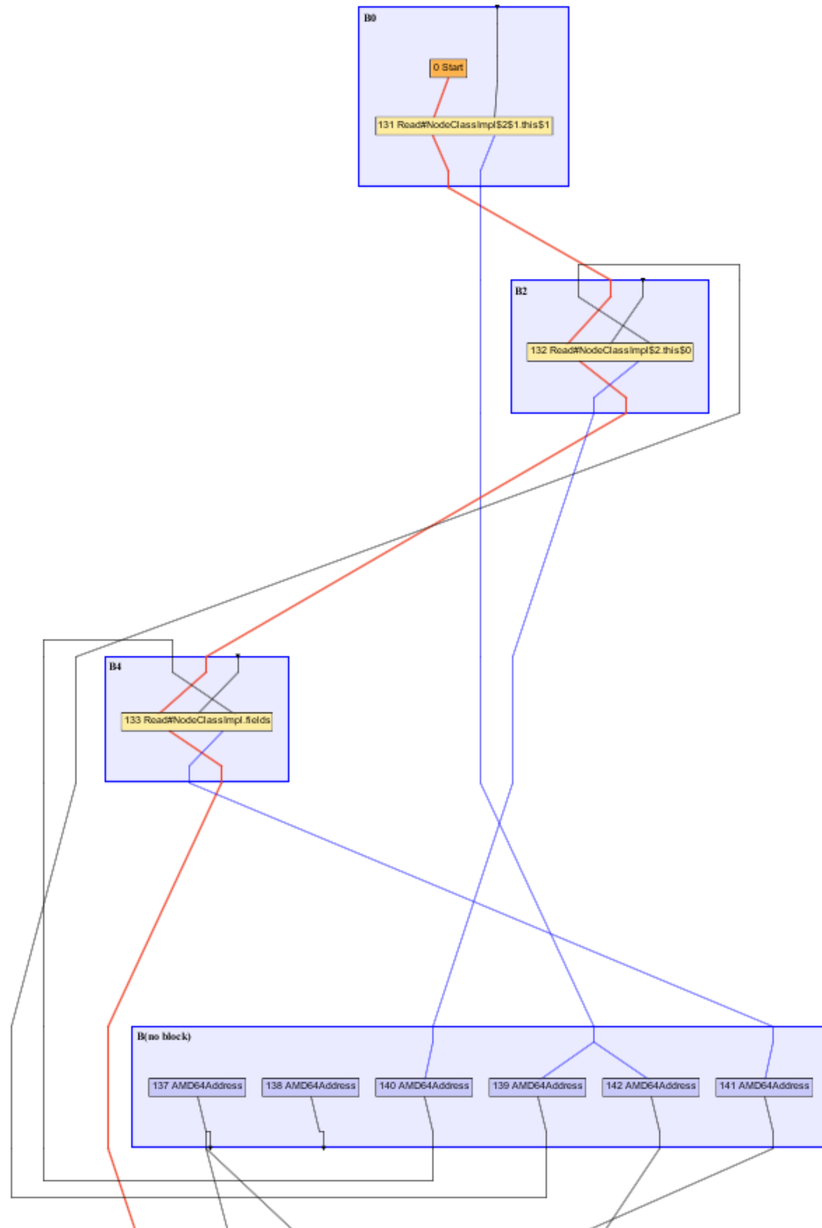


Figure 40 Problematic layout with clusters

Clusters also bring some problems, as shown in figure 40, due to splitting creation of layout into two parts. The inner layout algorithm, which knows nothing about graph outside the cluster, can place outputs from the cluster in wrong order, when considering to which other clusters the edges lead. This then brings additional unnecessary edge crossings. This is also connected to other problem. At the time, when relative ordering of nodes is created, it's done based on positions of nodes with no consideration of ports. This is apparent from clusters in second and third layer. In both cases, the outputs from cluster are placed in wrong order, but from a point of view of the algorithm, this placement was good, because both nodes wanted to be under their parent node and they got it. But later, when the edge routing is done and the ports are considered, it leads to edge crossing. It also cannot be really solved by swapping positions of ports on the side of nodes, as this position may have some additional meaning.

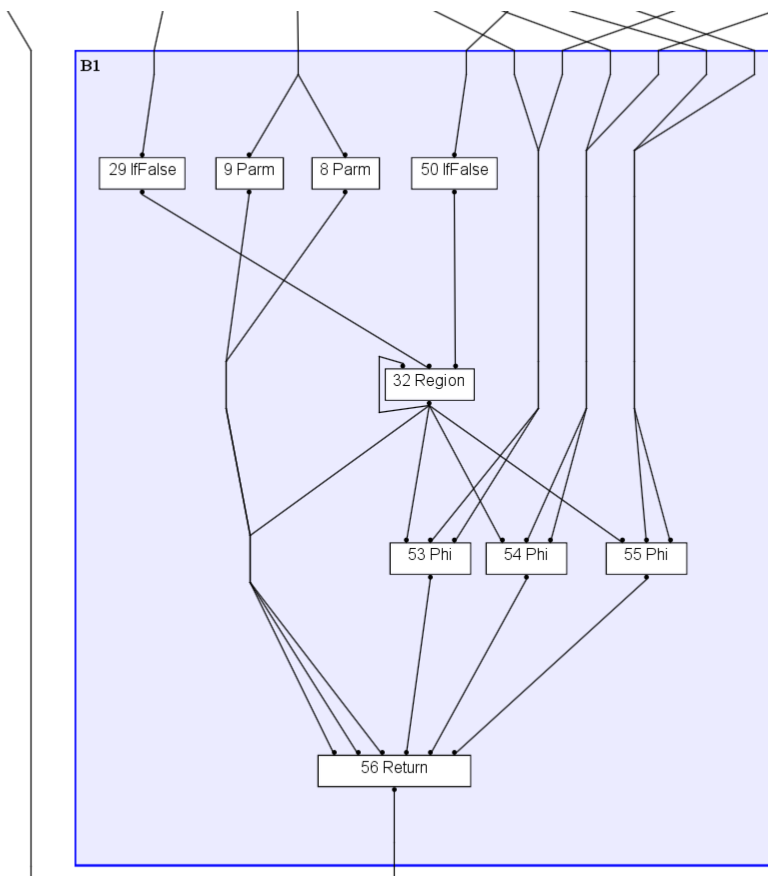


Figure 41 Cluster layout detail

Figure 41 contains cluster with many incoming connectors. It has also both aggregation rules set to aggregation per-node, which can significantly reduce number of visible edges. From this figure, it is evident it would be better if the connectors were joined even sooner, either during entering the cluster, or even before entering the cluster. But this is problematic, since the outer layout doesn't know content of the cluster and so it cannot aggregate the edges.

7 Coneclusion

The goal was to create a layout algorithm, which would allow to create layouts of hierarchical flow charts with about a few thousands nodes. This algorithm was created using Sugiyama's framework, which is heuristic hierarchical layout algorithm. Some of the aspect of the algorithm were modified to allow having nodes with subnodes. The algorithm also maintains stability of layout when focusing on only part of whole graph. This adds additional benefit, when it's used together with tools supporting node visibility and filtering. Functionality of this algorithm was presented when using it as part of the Ideal Graph Visualizer tool.

When comparing the new layout algorithm to the old layout algorithm implemented in the original IGV, it can be noticed that the new layout algorithm creates layouts which are easier to read. This was achieved by suppressing some aesthetic rules. When comparing these two solutions, it can be seen that balancing aesthetic criteria is more important than trying to maximize some of them at the expense of other.

7.1 Future improvements

The resulting layout algorithm manages to create good layouts, yet there are still places which could be improved. Created layouts sometimes contain more edge crossings than is needed and number of connectors could be decreased even more when there are edges between nodes in different clusters.

Also back edges could be reworked to be more aware of their surroundings in some cases, when a connector of back edge bends it does so inconveniently in wrong location. While it is always better to prevent having back edges, it would be better if in case the back edge exists, the algorithm could handle them with more awareness to their surrounding to make the resulting layout more comfortable for the user.

While this algorithm creates layouts in real time for small and medium sized graphs with up to few thousands nodes, it starts to fall behind when instances have several thousands nodes and edges. Maybe further optimizations of dummy nodes would allow to solve bigger instances, even though it's impossible to reach same performance as with layout algorithms that disregard layout stability.

7.1.1 Two step algorithm

Another possible improvement would be splitting the algorithm into two parts. In the first part, data structure of whole graph would be created together with removing cycles, assigning layers and creating relative positioning of nodes. Then the second part would handle assigning coordinates to nodes and routing edges. This split is possible since the first part of the algorithm is shared and always same, no matter which nodes and edges are visible in the current view. Splitting the algorithm would then allow to run the first part of the algorithm only when opening the graph for the first time. Subsequent changes in visibility of nodes and edges would invoke only the second part of the algorithm. This would increase performance when working with bigger graphs. Unfortunately, the architecture of IGV doesn't support this split at this time, but the

algorithm is created in such a way that splitting the algorithm into these two steps would be possible with relatively minimal changes in the algorithm itself. To split the algorithm properly, procedure that removes volatile nodes from invisible layer in the second part of algorithm would have to be removed and edge routing rewritten to jump over volatile nodes in these invisible layers. These changes shouldn't be done unless the algorithm will be truly used for two step layout creation, since current solution is faster.

Bibliography

[1] S. NIKOLOV, Nikola and Patrick HEALY. Hierarchical Drawing Algorithms. In: R. Tamassia, Editor.Handbook of graph drawing and visualization. CRC Press, 2013, pp. 409-453.ISBN 978-158-4884-125. Available from: https://cs.brown.edu/ rt/gdhandbook/chapters/hierarchical.pdf

[2] K. SUGIYAMA, S. TAGAWA and M. TODA. Methods for Visual Understanding of Hierarchical System Structures in IEEE Transactions on Systems, Man, and Cybernetics, vol. 11, no. 2, pp. 109-125, Feb. 1981. doi: 10.1109/TSMC.1981.4308636

[3] REYNOLDS, Jason. A Hierarchical Layout Algorithm for Drawing Directed Graphs. Canada, Ontario, Queen's University Kingston: 1997. Available from: http://www.collectionscanada.gc.ca/obj/s4/f2/dsk2/ftp04/mq20694.pdf. Thesis. Queen's University Kingston, Department of Computer and Intormation Science. Supervisor Dr. D. Rappaport

[4] G. KOBOUROV, Stephen. Spring Embedders and Force Directed Graph Drawing Algorithms [online]. CoRR, 2012, abs/1201.3011 University of Arizona. Seen on 19.1.2017 Available from: http://arxiv.org/pdf/1201.3011.pdf

[5] T. WÜRTHINGER. Visualization of Program Dependence Graphs. Austria, Linz, Institute for System Software Johannes Kepler University Linz: 2007. Available from: http://ssw.jku.at/Research/Papers/Wuerthinger07Master/Wuerthinger07Master.pdf. Master Thesis. Institute for System Software Johannes Kepler University Linz. Supervisor O.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck.

[6] T. REINHARD. Complexity Management in Graphical Models. Switzerland, Zurich, University of Zurich: 2010. Doctoral Thesis. University of Zurich, Faculty of Economics, Business Administration and Information Technology. Supervisors Prof. Dr. M. Glinz, Prof. Dr. H. C. Gall

[7] R. M. KARP. Reducibility among Combinatorial Problems. In: R. Miller, Editor. Complexity of Computer Computations. Springer US, 1972, pp. 85-103.ISBN 978-1-4684-2003-6.

[8] K. SUGIYAMA. Graph Drawing and Applications for Software and Knowledge Engineers. World Scientific, 2002. ISBN 981-02-4879-2.

[9] N. S. NIKOLOV, A. TARASSOV and J. BRANKE. In Search for Efficient Heuristics for Minimum-width Graph Layering with Consideration of Dummy Nodes. In: Journal of Experimental Algorithmics. ACM, 2005. doi:10.1145/1064546.1180618

[10] A. QUIGLEY, P. EADES. FADE: Graph Drawing, Clustering, and Visual Abstraction. In: J. Marks, Editor. Graph Drawing. Springer, 2000, pp. 197-210. DOI: 10.1007/3-540-44541-2

[11] *Ideal Graph Visualizer* [online]. Johannes Kepler University Linz. Seen on 19.1.2017. Available from: http://ssw.jku.at/General/Staff/TW/igv.html

[12] *Netbeans* [online]. Oracle Corporation. Seen on 19.1.2017. Available from: http://www.netbeans.org