Visualization of Code Flow

YURI STANGE
Visualization of Code Flow

Visualisering av kodflöde

YURI STANGE

Email at KTH: yuris@kth.se
Subject Area: Computer Science
Supervisor: Stefan Arnborg
Examiner: Stefan Arnborg
Abstract

Visual representation of Control Flow Graphs (CFG) is a feature available in many tools, such as decompilers. These tools often rely on graph drawing frameworks which implement the Sugiyama hierarchical style graph drawing method, a well known method for drawing directed graphs. The main disadvantage of the Sugiyama framework, is the fact that it does not take into account the nature of the graph to be visualized, specifically loops are treated as second class citizens. The question this paper attempts to answer is; how can we improve the visual representation of loops in the graph? A method based on the Sugiyama framework was developed and implemented in Qt. It was evaluated by informally interviewing test subjects, who were allowed to test the implementation and compare it to the normal Sugiyama. The results show that all test subjects concluded that loops, as well as the overall representation of the graph was improved, although with reservations. The method presented in this paper has problems which need to be addressed, before it can be seen as an optimal solution for drawing Control Flow Graphs.

Sammanfattning

# Contents

1  Introduction ................................................. 1

2  Control Flow Graphs ........................................ 1
   2.1 Directed graphs ........................................ 3

3  Representing code flow in a graphical way .................... 4
   3.1 Desired Layout Attributes ............................... 4

4  Previous works ............................................. 5
   4.1 Graphviz ................................................. 5
   4.2 OGDF .................................................. 5

5  Our goal .................................................. 6
   5.1 Needed Terminology ...................................... 7

6  Sugiyama Layout ............................................ 8
   6.1 Cycle removal ........................................... 8
      6.1.1 Depth-first search approach ....................... 10
   6.2 Hierarchical Layering .................................. 10
      6.2.1 Longest Path ...................................... 12
      6.2.2 Vertex Promotion algorithm ...................... 13
   6.3 Dummy nodes ........................................... 13
   6.4 Cross reduction ........................................ 16
      6.4.1 Layer-by-layer Sweeping ............................. 17
      6.4.2 Transposing ........................................ 19
   6.5 Coordinate Assignment .................................. 20
      6.5.1 Pendulum algorithm with linear segments ....... 20

7  Representing the edges ....................................... 22
   7.1 Ports ................................................. 22
   7.2 Avoiding obstacles ...................................... 23
      7.2.1 Manhattan convention .............................. 23
      7.2.2 Edge tracks ........................................ 24
      7.2.3 Assigning tracks to edges ......................... 25

8  Improving the representation of back edges .................. 26
   8.1 The main idea .......................................... 26
   8.2 Introducing new nodes ................................... 27
   8.3 Prerequisites ........................................... 28
   8.4 Starting off ............................................ 28
   8.5 Alterations to Sugiyama’s crossing reduction step ...... 29
      8.5.1 Downsweeps (receiving layer) ...................... 30
      8.5.2 Upsweeps (receiving layer) ....................... 30
   8.6 Tying up the ends ....................................... 31
      8.6.1 Aesthetical Hungarian .............................. 32
8.7 Alterations to the pendulum algorithm ........................................ 32
    8.7.1 Ibeds .................................................. 32
    8.7.2 Obeds .................................................. 34
8.8 Changes to the edge track partitioning ..................................... 34
    8.8.1 A priority approach .................................... 34
    8.8.2 Distributing tracks to obeds and ibeds ............................ 35

9 Implementation .......................................................... 36
    9.1 Qt ...................................................... 36
    9.2 The Graphics View Framework ................................. 36
        9.2.1 Edges ............................................... 37
        9.2.2 Vertices ............................................ 38
    9.3 About the algorithms ........................................... 38

10 Evaluation .............................................................. 38
    10.1 About the interviews ........................................ 38
    10.2 Results .................................................. 39

11 Conclusions ............................................................ 40

12 Future Work ............................................................ 43
    12.1 Crossings between edge tracks ............................... 43
    12.2 Positioning of edges relative to their obed nodes ............ 43
    12.3 Handling inevitable edge crossings ........................... 44
    12.4 Worst case scenario in the Manhattan layout ................ 45
    12.5 Better port strategies ...................................... 46
1 Introduction

Visual representation of code flow is a great tool available for programmers in order to understand, optimize, debug or reverse engineer programs. It consists of various steps where a visualization tool could be considered as the most important step in the process, also the hardest to implement. Many factors play a role in the quality of the graphical result, often tied up to notions better discussed in a paper about cognitive perception. Evaluation of such a tool is a fuzzy job where “performance” is not always best measured in numbers.

The meaning of this paper is to take a look at the existing solutions and to explore methods better suited for code flow visualization. The code flow in this paper will be represented as a Control Flow Graph (CFG), which can often be seen in the literature dedicated to compiler optimization.

Chapter 2 describes CFGs, how they are represented and their relationship to directed graphs. The requirements set on the layout algorithm are described in Chapter 3, while Chapter 4 presents previous works, their strengths and implementations. Chapter 5 introduces the goal of this paper. Chapter 6 describes in detail the Sugiyama framework, which our work is based on. Chapter 7 is about the way we chose to represent edges, which bind vertices together. Our method will be presented in Chapter 8, where details of its implementation can be seen in Chapter 9. The evaluation of our method, as well as our results are presented in Chapter 10. Our conclusions can be found in Chapter 11, while Chapter 12 will be about what we consider to be future work.

2 Control Flow Graphs

A Control Flow Graph is a representation of all paths a program may have taken during its execution using a graph notation. In this case a directed graph, which we will describe in the following subsection.

In a CFG each node (or vertex) in the graph represents a basic block, this means; a piece of code contiguous in memory without any jumps or jump targets. If the first instruction in the block is executed, then each subsequent instruction will also be executed (Cooper, Keith D. and Linda Torczon, 2004 cited in Saunders, 2007).

Jumps in the code flow are represented as directed edges, where the block being pointed at by the edge is the jump target. An example of possible code, with the corresponding CFG can be see in Figure 1 on the following page.
Figure 1: Example of code and its corresponding CFG
2.1 Directed graphs

A directed graph is a set of vertices connected by edges, where the edges have a direction associated to them. If we traverse a directed graph by using a Depth-first search\(^1\)(Tarjan, 1972), we can categorize the edges into four types; tree edges, forward edges, cross edges and back edges. It should be noted that the classification of edges is tightly coupled to the order in which the Depth-first search visited them. For an example see Figure 2.

- Tree edges describe the relation between a vertex and their direct descendants, they are usually associated with the “natural flow” of the graph.
- Forward edges connect a vertex to some of its descendants, although not their direct ones.
- Back edges describe descendant-to-ancestor relationships. They connect vertices to one of their ancestors in the graph.
- Cross edges are all the other edges in the graph, they connect unrelated nodes in the partial order.

Since CFGs are in fact directed graphs, it is not difficult to drawn parallels between them. Figure 1 on the preceding page demonstrates a if-then clause which gives origin to a cross edge. This paper will focus heavily on back edges, which are usually originated by loops in the code, such as while- and for-clauses.

\(^1\)Algorithm for traversing a graph, where each branch is explored as far as possible before backtracking.
3 Representing code flow in a graphical way

Our goal is to represent the CFG in such way that it becomes trivial to understand the code behind. A graph representing the code should be as powerful for a computer scientist, as a chart pie is for a statistician. This is where two important notions come into play; readability and understandability.

The notions of readability and understandability in a graph are very subjective, and often coupled to the data they are trying to represent. Since we are working with control flow graphs, some of the key points we think should be represented are; How do the blocks relate to each other? Which paths can the code take between two blocks? Where are the loops? Which blocks are involved in loops? How and where does the code branch?

These questions lead to a set of attributes desired in the final representation of the graph.

3.1 Desired Layout Attributes

Saunders (2007) describes in his paper some desired qualities for the representation of the code flow. It is worth to mention, as he puts it, that the “...quality of [the] graph layout [is] based upon human perception, [and] but the desired qualities are often in direct conflict of each other [or] and otherwise not fully obtainable for many graphs.”

Some of the desired qualities Saunders mentions are:

- All edges must point downward except for edges representing backward jumps, which must point upwards.
- Make loops visually distinct.
- Minimize edge crossings.
- Minimize edge bends.
- Minimize edge length.
- Minimize overall graph dimensions, while maintaining sufficient space between edges and nodes for readability.
- Maximize edge verticality.
- Inbound edges should meet at the top of a node, and outbound should meet at the bottom.
4 Previous works

There is a plethora of libraries and tools available which do a fantastic job at visualizing graphs. In this chapter we will look at two which we considered to be the most developed at the time this paper was written. We will now describe them and their strengths.

4.1 Graphviz

Graphviz (www.graphviz.org) is an open source graph visualization software originally developed by AT&T Research labs. It has been incorporated into many projects such as Doxygen (tool for generating documentation from annotated C++ sources) and Scribus (Open source desktop publishing software). Graphviz is not aimed at being an interactive visualization tool, but rather a layout generator for graphs. Graphviz takes as input simple text representation (called .dot files) of graphs and outputs layerings in useful formats such as SVG (Scalable Vector Graphics) or PDF (Portable Document Format).

Graphviz consists of several layout programs which are aimed at different types of graphs. Some examples are; “neato” which is an implementation of an energy-based layout algorithm, and “circo” which generates circular layouts.

The hierarchical (see Sugiyama layout in chapter 6) style layout implementation in Graphviz is called “dot”, which is based on the paper written by Gansner et al. (1993). It utilizes an LP-solver\(^2\) for steps such as the layering and balancing of the graph, which minimizes the length of edges and generates perfect solutions. This does not mean however that the overall solutions are perfect.

One of the main strengths of Graphviz is that it is easy to use. The input files can be easily generated, even by hand, since they are human readable. It is also capable of generating output files in a wide selection of formats.

4.2 OGDF

The Open Graph Drawing Framework (OGDF) is a library (written in C++) engineered for graph drawing. Although OGDF is a great graph layout generator, its main goal is providing a toolset filled with a variety of algorithms and data structures used for graph drawing.

The Open Graph Drawing Framework’s main strength is the modularity of its components, which allow the user to reuse and replace particular algorithms to his preference. It is also a great foundation for engineering and trying new drawing methods without the need to reinvent the wheel.

OGDF has a wide set of layout algorithms, varying from planar graph drawing to multi-level energy-based drawing styles. The implementation of the

\(^2\)Tool for optimization of linear objective functions with linear constraints.
Sugiyama framework in OGDF can be custom tailored by picking the desired algorithms to be used, where several options exist for every step, including methods involving LP-solvers.

5 Our goal

Both OGDF and Graphviz implement the method described by Gansner et al. (1993), which is in fact an implementation of the Sugiyama framework. They deal with back edges (cycles in the graph) by temporarily reversing them. Back edges are handled almost as any normal edge in the graph, with the only difference being the reversed orientation in which they are rendered (see Figure 3). This breaks our desired aesthetical requirement where inbound edges should meet at the top of a node, and outbound at the bottom.

Our goal is to improve the representation of back edges by complying to the aesthetical requirements. The goal is to have back edges meet the targeted vertex only from the top, going around the vertex in question if so needed. We decided to let back edges come in only from the left of the targeted vertex, since we believe this will improve the readability of the graph (based on the reading direction of most western cultures). A picture demonstrating the goal can be seen in Figure 4 on the following page.

We chose to solve this with a procedure that reminds a lot of a reduction\(^3\).

\(^3\)A way of solving a problem we do not know a solution for, by transforming it into another problem we know how to solve.
By adding small variations to the Sugiyama framework we can let the original algorithms solve most of the problems faced.

The next chapter will describe the Sugiyama framework in detail, since the knowledge is needed before diving into our method. We recommend the reader to take a look at the following subsection, it contains useful terminology used across the rest of the paper.

5.1 Needed Terminology

In order to understand the rest of this paper, certain terminology concerning graph theory must be covered.

\( G = (V, E) \) A graph \( G \), consists of a set of vertices \( V \) and a set of edges \( E \).

\( V = \{u, v, v_1 \ldots v_n\} \) The set \( V \) denotes the vertices in the graph \( G \). They are sometimes called nodes. In this case \( V \) consists of the vertices: \( u, v \) and \( v_1 \ldots v_n \).

\( E = \{e(u,v)\} \) The set \( E \) denotes the edges in the graph. Each edge has a source- and a target vertex. For example the edge \( e(u,v) \) has \( u \) as source and the targeted vertex is \( v \).

\( N^+_G(v) \) This denotes the set of edges with the vertex \( v \) as source, in the graph \( G \).

\( N^-_G(v) \) This denotes the set of edges targeting \( v \), in the graph \( G \).
6 Sugiyama Layout

The Sugiyama-style graph drawing, also known as hierarchical layout, is one of the most popular methods for drawing directed graphs which represent a hierarchy. It was presented by Sugiyama et al. in 1981. It generates layouts where the vertices are distributed into layers. The hierarchical layout method consists of several steps which can be solved by different algorithms. A picture showing an example of the algorithm in work can be seen in Figure 5 on the next page.

The Sugiyama framework consists of the following steps, which are run in the presented order;

- **Cycle removal** Possible cycles in the input graphs are removed.
- **Hierarchical layering** Vertices are distributed into layers. (In our case horizontal layers)
- **Dummy nodes insertion** Dummy nodes are inserted into the layering in order to cope with long edges.
- **Cross reduction** The order of the vertices in each layer is rearranged in order to minimize the number of edge crossings.
- **Coordinates assignment** Vertices and edges receive their absolute position.

We will now explain in detail every step of the Sugiyama framework, and the algorithms we have chosen for each step.

6.1 Cycle removal

The first step of the Sugiyama method is to remove, or alter the direction of some edges in order to turn the input into an acyclic graph, also called a DAG (directed acyclic graph). This is done in order to comply with the requirements set by several algorithms in the subsequent steps.

Once the vertices have been assigned to a layer, the original directions of the edges can be restored. It is most desirable to alter the direction of the edges instead of removing them, since the latter would lead to the layering algorithm working on a subset of the graph, which can give undesirable results. (Healy and Nikolov, 2013).

It is also desirable to keep the number of reversed edges as low as possible in order to retain the hierarchical characteristics of the original graph. The minimization problem is known as the maximum acyclic subgraph problem which unfortunately is shown to be NP-hard (Garey and Johnson, 1991 cited in Bastert and Matuszewski, 2001). In order to tackle that, heuristics are put to use.
Figure 5: Sugiyama Algorithm. From Cycle breaking to Coordinate assignment.
Algorithm 1 Depth-first based Remove Cycles

Require: A vertex without incoming edges

1: function breakCycles(graph G)
2:   for each vertex n in G without incoming edges do
3:     DFS(n)
4:   end for
5: end function

6: function DFS(vertex n)
7:   mark n as active
8:   mark n as visited
9:   for each outgoing edge e from n do
10:      v ← vertex targeted by e
11:      if v is active then
12:         reverse edge e
13:      else if v is not visited then
14:         DFS(v)
15:      end if
16:   end for
17:   mark n as not active
18: end function

6.1.1 Depth-first search approach

This is a simple method for removing cycles in a graph, based on a depth-first search. A picture demonstrating the method can be seen in Figure 6 on the following page.

As we remember, the edges in a directed graph can be categorized with a depth-first search. The role of this algorithm is to reverse back edges, turning them into forward edges and thus removing the cycles (Gansner et al., 1993). This method marks the vertices as either visited, active or both. A depth-first search is started at every vertex without incoming edges. The algorithm will crawl through edges and mark the vertices as active if they are in the current chain. Once it has crawled all edges outgoing from that vertex, it will remove the active tag and leave it marked as visited.

If the algorithm finds an edge connecting two vertices marked as active it knows it’s part of a directed cycle (edge pointing to an ancestor, alas a back edge), and will reverse the edge. A pseudocode of the algorithm can be seen in algorithm 1.

6.2 Hierarchical Layering

In this step the main goal is to layer the graph in such way that no edge may connect two vertices sharing the same layer. There are many algorithms that
Figure 6: Cycle Breaking DFS
attack this problem from different angles, the main difference between them being what the algorithms tries to minimize; be it height, width or the number of dummy nodes (see subsection 6.3). It should be noted that minimizing the number of layers with respect to a given width is NP-hard (Bastert and Matuszewski, 2001).

### 6.2.1 Longest Path

The longest path layering algorithm is a list scheduling algorithm which can be used to partition a graph into layers. The main idea is that given an acyclic graph we place the vertices on the $i$:th layer, where $i$ is the length of the longest path to the vertex from a source vertex. A picture demonstrating the idea can be seen in Figure 7.

It is a simple algorithm, easy to implement, has a linear time complexity and produces layerings with the fewest possible number of layers (Healy and Nikolov, 2013). It does however perform very poorly in terms of drawing area and number of dummy nodes (P. Healy and N. S. Nikolov, 2002 cited in Healy and Nikolov, 2013).

The algorithm starts off by assigning all vertices without incoming edges to the first layer. In each iteration it assigns vertices to a new layer if and only if all their ancestors have already been assigned to a layer before. Pseudocode for the algorithm can be seen in Algorithm 2 on the next page.
Algorithm 2 The Longest-Path algorithm, adapted from Healy and Nikolov, 2013

Require: DAG $G = (V, E)$

1: function Longest-Path
2:   $U \leftarrow \emptyset$ \triangleright Set of all vertices already assigned to a layer
3:   $Z \leftarrow \emptyset$ \triangleright Set of all vertices assigned to a layer, below the current layer
4:   $currentLayer \leftarrow 1$
5:   while $U \neq V$ do
6:     Select $v \in V \setminus U$ with $N_G^+(v) \subseteq Z$
7:     if $v$ has been selected then
8:       Assign $v$ to the layer with a number $currentLayer$
9:       $U \leftarrow U \cup \{v\}$
10:   end if
11:   if no vertex has been selected then
12:       $currentLayer \leftarrow currentLayer + 1$
13:       $Z \leftarrow Z \cup U$
14:   end if
15: end while
16: end function

6.2.2 Vertex Promotion algorithm

Vertex promotion is a simple heuristic that can be applied to any layering in order to reduce its dummy vertex count (Nikolov and Tarasov, 2006 cited in Healy and Nikolov, 2013). The algorithm promotes vertices placed on a layer, to the layer above it. This is only done on vertices from the original graph and not those created as dummy vertices. The pseudocode for this algorithm can be seen in Algorithm 3 and 4.

The Vertex-promoting algorithm consists of two functions:

- PromoteVertex; calculates the difference in the count of dummy vertices if the node and its predecessors (recursively), would be promoted to a layer above.

- Vertex-Promoting heuristic; traverses all vertices with at least one predecessor and attempts to promote them with PromoteVertex. If the count of dummy vertices returned is lesser than the current layering’s then the new one is preserved.

A picture demonstrating the algorithm can be seen in Figure 8 on page 15.

6.3 Dummy nodes

In order for the crossing reduction and positioning algorithms to work, the layering of the graph must meet an extra requirement; The layering should be proper. That means all edges should have a span of 1. In order to allow for
Algorithm 3 PromoteVertex, adapted from Healy and Nikolov, 2013

Require: A layered DAG $G = (V, E)$ with the layering information stored in a global vertex array of integers called layering; a vertex $v \in V$.

1: function PromoteVertex($v$)
2:     dummydiff ← 0
3:     for each $u \in N^-_G(v)$ do
4:         if layering[$u$] = layering[$v$] + 1 then
5:             dummydiff ← dummydiff + PromoteVertex($u$)
6:         end if
7:     end for
8:     layering[$v$] ← layering[$v$] + 1
9:     dummydiff ← dummydiff - $N^-_G(v) + N^+_G(v)$
10:    return dummydiff
11: end function

Algorithm 4 Vertex-Promoting Heuristic, adapted from Healy and Nikolov, 2013

Require: $G = (V, E)$ is a layered DAG; a valid layering of $G$ is stored in a global vertex array of integers called layering.

1: function Vertex-Promoting Heuristic
2:     layeringBackUp ← layering
3:     repeat
4:         promotions ← 0
5:         for each $v \in V$ do
6:             if $d^-(v) > 0$ then $\triangleright$ number of predecessors
7:                 if PromoteVertex($v$) < 0 then
8:                     promotions ← promotions + 1
9:                     layeringBackUp ← layering
10:                 else
11:                     layering ← layeringBackUp
12:             end if
13:         end if
14:     end for
15:     until promotions = 0
16: end function
Figure 8: Vertex promotion algorithm
Algorithm 5 Insertion of dummy nodes.

Require: $G = (V, E)$ is a layered DAG

1: function CreateDummyNodes
2:   for each $e(u, v) \in E$ do
3:     $i \leftarrow \text{layer}(u) + 1$
4:     $j \leftarrow \text{layer}(v)$
5:     $\text{last} \leftarrow u$
6:     while $i \neq j$ do
7:       $d \leftarrow$ new dummy
8:       $L_i \leftarrow L_i \cup \{d\}$
9:       $E \leftarrow E \setminus \{e(\text{last}, v)\}$
10:      $E \leftarrow E \cup \{e(\text{last}, d)\}$
11:      $E \leftarrow E \cup \{e(d, v)\}$
12:      $\text{last} \leftarrow d$
13:      $i \leftarrow i + 1$
14:   end while
15: end for
16: end function

edges spanning across several layers dummy nodes are introduced.

A dummy node is a virtual vertex which has neither length nor width. Neither do they have a corresponding node in the input graph. Its major role is to connect vertices whose edges span across several layers in the layering. Pseudo code for the insertion of dummy nodes can be seen in Algorithm 5.

What the algorithm does is crawl the graph looking for edges $e(u, v)$ where the layer of $v$ isn’t the layer of $u$ plus one. In that case a new dummy node in the layer below $u$ is created. The edge between $u$ and $v$ is removed and two new edges; one from $u$ to the new dummy node, and another from the dummy node to $v$, are added to the set of edges. A picture can be seen in Figure 9 on the following page.

6.4 Cross reduction

Edge crossings are one of the crucial parameters in a graph drawing for the sake of readability (H. C. Purchase, 1997 cited in Healy and Nikolov, 2013). The main goal of this step in the Sugiyama framework is to reorder the vertices in all layers, in order to minimize the number of edge crossings between them. Edge crossings make a graph hard to follow and clutters the drawing. Parallel, or close to parallel edges are specially good at making a graph hard to read.

The ideal would be to have a drawing without any edge crossings, this is not always possible since not all graphs are planar. That means not all graphs are able to be drawn in a way where no edges crosses another.
Figure 9: Inserting dummy nodes

One notable observation is that we do not have to look at the absolute $x$-positioning (horizontal positioning) of the nodes, since the number of edge crossings is based on the relative positioning of the vertices in each layer (Bastert and Matuszewski, 2001). Eg. Vertex $v$ is to the right of vertex $u$ in layer $i$.

The general two-layer crossing minimization problem, where permutations for both layers that minimize the number of crossings is required, is NP-hard (M. R. Garey and D. S. Johnson 1983 cited in Healy and Nikolov, 2013). Even if one layer is fixed, the problem still remains NP-hard (P. Eades and N. C. Wormald, 1994 cited in Healy and Nikolov, 2013). In order to tackle this, heuristic algorithms are used. Most of the algorithms do no attempt to reduce the number of crossings on the whole graph at once, but instead they do a layer by layer-sweep.

6.4.1 Layer-by-layer Sweeping

Since trying to minimize the number of crossings in the whole graph is way too hard, a heuristic called layer-by-layer sweep is used instead. What it does is try to minimize the edge crossings between two adjacent layers, one pair of layers at a time. Once the algorithm has minimized the number of crossings between layer $k$ and $k + 1$, it will continue on with layers $k + 1$ and $k + 2$. A picture of the algorithm can be seen in Figure 10 on the next page.
Figure 10: Layer-by-layer sweeping

The layer by layer sweep uses a weight calculated on each of the vertices in the second layer in order to decide their relative positioning. The different ways this weight is calculated give different results. Examples of weight calculating methods are; the barycenter function and the median function. Once the weights have been calculated, the relative order of the vertices can be rearranged. The graph is swept back and forth until the number of crossings is low enough. Pseudocode for the layer-by-layer sweep can be seen in Algorithm 6 on the following page.

The barycenter function calculates the weight of a vertex based on the average of the positions of its neighbours in the previous layer. In other words we calculate \( N(u) \) are all vertices connected to \( u \) from the layer before, \( \pi_1(v) \) is the position of \( v \) in it’s layer):

\[
bary(u) = \frac{1}{\deg(u)} \sum_{v \in N(u)} \pi_1(v)
\]

The median function is much alike the barycenter function. Instead it calculates the median of the positions of the neighbours in the previous layer. Both the barycenter and median function will find a setup without crossings if one exists, however this guarantee is only valid considering two layers (Bastert and Matuszewski, 2001). It should be noted that the barycenter function is known to perform better than the median function in most cases (Healy and Nikolov, 2013).

There is one fact we found out during the implementation of our code, which concerns the weight calculation during upsweeps. In the rare case, when the node to be weighted does not have any successors, instead of setting its weight to zero, we would get better results if we used its downsweep weight.
Algorithm 6 Layer-by-layer sweep, adapted from Saunders, 2007

1: function LAYER-BY-LAYER-Sweep
2: while the crossing number is not satisfactory do
3: for each layer $V_i$ from $i = 0$ to $n$ do
4: for each $v \in V_i$ do
5: Calculate weight $W_p(v)$ \hspace{1em} $\triangleright$ $W_p$ is the chosen weight function
6: end for
7: Sort the nodes of $V_i$ according to the weight $W_p(u)$
8: end for
9: for each $V_i$ from $i = n$ to 1 do
10: \hspace{1em} ... similar with $W_s(v)$ \hspace{1em} $\triangleright$ $W_s$ is $W_p$ seen from bottom to top
11: end for
12: end while
13: end function

Algorithm 7 Transposing algorithm

Require: Layer $layer = \{v_0, v_1, ..., v_n\}$

1: function TRANSPOSE
2: \hspace{1em} $improved = true$
3: while $improved = true$ do
4: \hspace{1em} $improved = false$
5: for each vertex $v_i$ from $i = 0$ to $n - 1$ do
6: \hspace{1em} if $\text{crossings}(v_i, v_{i+1}) > \text{crossings}(v_{i+1}, v_i)$ then
7: \hspace{1em} \hspace{1em} $improved = true$
8: \hspace{1em} \hspace{1em} SWAP($v_i, v_{i+1})$
9: \hspace{1em} end if
10: \hspace{1em} end for
11: end while
12: end function

6.4.2 Transposing

Transposing is a simple heuristic presented by Gansner et al. (1993). It tries to reduce the number of edge crossings by greedily switching vertices. If swapping two adjacent vertices (on the same layer) would produce a better configuration then the algorithm does so.

Gansner et al. suggest the transposing algorithm to be run in combination with other heuristics, such as median or barycenter based layer-by-layer sweep. Pseudocode for the transposing algorithm can be seen in Algorithm 7.

The transposing algorithms needs knowledge on the actual number of crossings in the current configuration. A method for calculating the number of crossings is presented in Sander, 1995.
6.5 Coordinate Assignment

This is the final step in the Sugiyama framework. The main goal is to assign absolute positions to the vertices and edges, in respect to the desired aesthetical qualities.

The layout should be balanced, with vertices centered among their neighbours. Edges should be straight since there is evidence for that being preferable from a perceptual point (Huang, Eades and Hong, 2009 cited in Healy and Nikolov, 2013). This applies specially to long edges, which we have partitioned into dummy nodes in the steps before.

6.5.1 Pendulum algorithm with linear segments

This method was presented in Sander, 1996 and is a combination of the pendulum method and linear segments. Its major goal is to represent long edges (composed of dummy nodes) as straight lines, while keeping the balance in the graph.

Linear segments

In order to describe the method we should start by explaining the notion of linear segments. A linear segment is a sequence of vertices which should be drawn as a straight line, where each vertex belongs to at most one linear segment, for an example see Figure 11.
Pendulum method

The pendulum algorithm (presented in Sander, 1995 and Sander, 1999) is a layer-by-layer sweeping algorithm (much like the crossing reduction algorithms), based on a physical model which can be seen as a set of balls and strings. The vertices are balls and edges strings, with the uppermost vertex hanging from the ceiling. The idea is to let gravity decide their positions, which should result in the most balanced setup. This is done in both directions, with reversed gravity when sweeping from the bottom-up.

Mathematically speaking, the algorithm’s goal is to minimize the force $Z$, defined as:

$$Z = \sum_{v \in V} \left| \sum_{(w,v) \in E} (X(w) - X(v)) \right|$$

which as seen, is the sum of the horizontal distance between all connected vertices. In order to minimize $Z$ we let the vertices move left and right (swinging them back and forth) based on a force which is calculated differently depending on the direction of the sweep.

We define the force $F$ which decides the direction of the swinging as:

$$F_{\text{pendulate}_-\text{down}}(v) = \frac{1}{\text{indeg}(v)} \sum_{(w,v) \in E} (X(w) - X(v))$$

$$F_{\text{pendulate}_-\text{up}}(v) = \frac{1}{\text{outdeg}(v)} \sum_{(v,w) \in E} (X(w) - X(v))$$

If $F$ is negative then we move the vertex to the left, if not, to the right. How far we are to move it depends on; how much room the vertex has to his closest neighbour in that direction, or how high the minimum “swinging movement” is (which is set by the user).

When the balls are too close to each other, or otherwise touching, they influence each other in such way that they should be seen as an entity. We call the set of vertices influencing each other as regions. If any vertex in a region is to be moved in any direction, then the entire region should be moved as a whole.

Regions can be combined or torn asunder depending on how they influence each other. If two regions interact with each other by either coming too close, or “pushing” into each other, then we combine them into one region.

If a region is split, creating an horizontal space between their balls, then we can no longer say they influence each other, and thus, should be divided into two or more regions. A picture describing the birth and death of regions can be seen in Figure 12 on the next page.
Combining methods

By combining the notion of regions with linear segments we can achieve straight long edges, which span across several layers. We start by partitioning the graph into disjoint linear segments and remaining vertices (which are in fact trivial linear segments).

The next step is to give each linear segment a starting positioning. This is done topologically, giving each linear segment the leftmost positioning it is allowed to, by not skewing, or bending any other linear segments. It is important to give each one of the vertices in a linear segment the same $x$-position.

The next step is to apply the pendulum method on the graph with one tiny change; instead of regions consisting of vertices they are now made out of linear segments.

7 Representing the edges

Another aspect to be discussed, which the Sugiyama framework does not touch, is how we represent the edges in the graph. The first topic being; *Ports*, where the edges originate from and their point of destination. The second topic is how we chose to have edges avoid obstacles, such as vertices.

7.1 Ports

One of the aesthetical requirements touches the subject of where and how edges should interact with vertices; incoming edges should always meet at the top of the vertex, while outgoing ones should only be connected to the bottom of a vertex. In order to facilitate this we chose to introduce the notion of *ports*. 
A port is an abstract location on a vertex, where edges both originate from and end their journey. It should be noted that ports are not to be rendered by the application, but are nothing but an abstract notion used in the implementation.

Incoming edges are connected to so called inbound ports while outgoing edges get to be connected to outbound ports. That means every single edge (directed edge) is a path between the outbound port on the source vertex, and the inbound port on the targeted vertex.

In order to comply with the aesthetical requirements we allow inbound ports to be placed only on the topside of a vertex while outbound ports get to be placed only on the bottom side of a node. A picture demonstrating the positioning of ports can be seen in Figure 13.

7.2 Avoiding obstacles

Since we want to allow vertices with different sizes, we need to address the situation when edges (represented as a straight line) would cross vertices, as seen in Figure 14 on the next page.

There are a few ways to solve this problem, such as using polylines that “route” around the vertices. The method we have chosen is to implement a Manhattan layout for the edges. The reasoning behind this choice being; Manhattan is the strategy we consider to be the easiest to adapt to our goals.

7.2.1 Manhattan convention

The Manhattan convention is a strategy where only vertical and horizontal edges are allowed. Our reasoning is the idea to let edges “escape” the dangerous zones (by moving vertically), before taking any horizontal steps. We achieve this by allowing edges to move in the horizontal direction only in the space between layers, where edges could not possibly cross any vertex. An example can be
The choice of the Manhattan strategy leads us to one problem that must be addressed; we cannot allow multiple horizontal edges be drawn on top of each other since that would lead to readability issues. We solve this by dividing the safe zone into tracks.

### 7.2.2 Edge tracks

Edge tracks is the approach we have chosen to tackle the readability problems introduced by the Manhattan convention. The goal is to not allow any horizontal sections from different edges to be drawn on top of each other.

The method is mentioned in Sander (1996), where he describes it as dividing the vertical space between layers into \( k \) horizontal line rows. For an example see Figure 16 on the next page.

The idea is to assign each edge to a track, where they are allowed to draw their horizontal section.
Once an edge has been given a track, it will leap vertically from its outbound port to the \textit{y}-position given by the track. It will then move horizontally until it aligns vertically with the \textit{x}-position of the targeted vertex’s inbound port, where it will finally leap vertically once again, connecting to the inbound port of the targeted vertex.

7.2.3 Assigning tracks to edges

The algorithm for assigning edges is fairly simple. We define the “start” and “end” of an edge as the \textit{x}-positions given by the inbound and outbound ports it connects. Where the leftmost of these two is the start. We can now crawl the horizontal space between the two layers as if it was a time line, from left to right.

The problem reminds a lot of the old time telephone company, where concurrent calls would be assigned to different phone lines.

Once we reach the start of an edge (phone call), a free track (phone line) is assigned to it. The track is now “occupied” by the edge. Once we reach the end of the same edge, we can now free the track and make it available for other edges. The main idea then, is to reduce the number of tracks \( k \) needed by assigning a currently free track (if possible, w/o creating a new one) to each edge that runs concurrently.

We chose to implement the method described in Sander (1996); All edges are sorted in ascending order by their \textit{x}-positioning and crawled by a sweeping line from left to right. Once we reach the start of an edge; it is assigned a track and added to the unfinished set \( U \). When we reach the end of an edge it is removed from the set and the track released. Every new edge added to \( U \) must then be assigned a track not currently in use by the edges residing in the set.
Algorithm 8 Method for assigning tracks. Adapted from Sander (1996)

1: function ASSIGN-LAYERS
2:    U = \emptyset; \quad k_i = 0;
3:   for each \((e, x) \in C\) in increasing order do
4:       if \(x_s(e) \neq x_e(e)\) then \(\triangleright x_s\) start, \(x_e\) end
5:           if \(x = \min\{x_s(e), x_e(e)\}\) then
6:               \(r(e) = 1 + \max\{r(e')|e' \in U\}\) \(\triangleright r(e) = \) track assigned to \(e\)
7:                   if \(k_i < r(e)\) then
8:                       \(k_i = r(e)\)
9:                   end if
10:              U = U + e
11:           else /*\(x = \max\{x_s(e), x_e(e)\}\)*/
12:               U = U - e
13:           end if
14:       end if
15:   end for
16: end function

The pseudocode for this method can be seen in Algorithm 8.

8 Improving the representation of back edges

In this chapter we will describe the changes made to the Sugiyama framework and their implications.

The first step of the Sugiyama method is to remove back edges. This is done by simply reversing the direction of the edge in question. When it is time to render the resulting layout, the only attempt made in order to restore the nature of the original graph is to visually reverse the back edges once again. Back edges become then a path from the inbound port in the original source vertex, to the outbound port in the original targeted vertex.

In order to comply with the aesthetical requirements we need to improve the representation of back edges. The goal is to have back edges become a path from the outbound port (on the original source vertex), go around the target vertex and connect its end to the corresponding inbound port.

8.1 The main idea

The main idea reminds a lot of a reduction problem, if we manage to place an edge (representing a back edge) to the left of the original targeted vertex, we can easily tie it to both the original source and target vertices.

In order to achieve this we need to introduce new nodes, which will remind a lot of dummy nodes, and let them be handled by the Sugiyama framework in
a similar fashion. There are some differences and special cases to be taken into account, which will be discussed in detail.

8.2 Introducing new nodes

The two nodes to be introduced are what we decided to call *outbound back edge dummy* (obed) and *inbound back edge dummy* (ibed) nodes. To easily understand the meaning of these, we recommend the reader to take a look at Figure 17. It should be noted that these nodes, as with other dummy nodes, will not be rendered by the rendering algorithm.

The role of the obed nodes, is to represent the turning point where the back edges start their journey leading to the layer housing the targeted node. At one point the layout algorithm will create a small edge, connecting the outbound port of the source vertex to an obed node, which will be picked later in the algorithm.

Ibed nodes have a similar job. They will act as anchors, defining the turning point where back edges end their journey. These are closely tied to the targeted node and will always reside closest to the left of it. Once the layout algorithm is done, an almost trivial edge will be drawn from the ibed node to the corresponding inbound port on the targeted vertex.

![Diagram showing obed and ibed nodes](example)

Figure 17: Introducing obed and ibeds.
8.3 Prerequisites

Before we introduce these new nodes to the graph there is one certain prerequisite that must be met; The layout algorithm must not forget which edges were reversed in the first step of the Sugiyama method.

8.4 Starting off

We assume the Sugiyama framework has taken the following steps up to this point;

- Cycles have been removed by reversing a few edges. These edges have been noted as “reversed”.
- Vertices have been distributed into layers.
- Dummy nodes have been added to the layering.

The first step is to remove the reversed edges from the graph, although leaving the notation which points out which edges have been reversed. The decision to remove the original reversed edges from the graph comes from the following; By adding ibed and obed nodes we have already reserved space for the reversed edges. If we do not remove them, they will skew the balancing of the graph and make logical alterations, such as creating unnecessary dummy nodes.

Each reversed edge will need at least two new nodes; an obed and an ibed. The obed node will be placed right to the left of the original source vertex. The pairing ibed node will be placed strictly to the left of the original targeted vertex, on the same layer as the vertex itself.

The next step is to tie these two together with an edge leading from the ibed node to the newly placed obed. If the back edge spans across several layers then normal dummy nodes are to be placed in between, in order to keep the layering proper. For an example see Figure 18 on the following page.

This leads to one pair of matching ibed and obed nodes for each reversed edge in the graph.

In the case a vertex is involved in several reversed edges, we proceed to locate the ibed or obed nodes directly to the left of the node in question. How close to the vertex they should be placed depends on a set of criteria;

- In case we have several ibed nodes spawned by a vertex; we will place them in order of the span of the back edge, with shorter jumps being closer to the vertex that spawned them. If two ibeds belong to jumps of the same length then we just place them in order of arrival.
- In case we need to place both ibed and obed nodes next to a vertex, we will give the ibed nodes a higher priority, thus placing them closest to the node in question.
• If the vertex gave birth to several obed nodes then we proceed to place them in order of arrival.

These criterion are followed for a number of reasons; We want the incoming edges (based in ibed nodes) closely tied to the vertex, ordered by the length of the back edge they represent. By doing this we instantiate a notion of “the further the jump, the further from the vertex” which we believe will increase the readability of the graph, again based on the reading direction in western cultures.

In contrast to ibed nodes, obed nodes are not closely tied to the vertex which they spawned next to, at this stage of the algorithm they are not owned by a particular node, but are part of a chain which ends at a particular layer.

### 8.5 Alterations to Sugiyama’s crossing reduction step

We would like to start by reminding the reader that our goal is having ibed nodes closest to the left of the vertex that spawned them, as well as straight back edges. In order to preserve this a few changes must be done to the crossing reduction algorithm.

As we recall, the layer-by-layer sweeping algorithm works with two adjacent layers in pairs, where one of them can be seen as the “receiving” one. Which one is deemed as the receiving layer depends if we are doing a downswEEP or
8.5.1 Downsweeps (receiving layer)

The major change in the downsweeps is how we handle ibed nodes; When the time comes to calculate the weight of the ibed nodes we just skip them, due to their lack of incoming edges. They are also ignored by the reordering phase.

In order to keep them closest to the vertex that “fathered” them, we will always make sure that; In case the crossing reduction algorithm decides to change the relative position of a vertex that spawned ibed nodes, it will always carry it’s “children” along, keeping them to the left.

An example can be seen in Figure 19.

Obeds on the other hand will be handled as any other node in the graph. This is done in order to allow the algorithm to find a setup which gives us straight back edges.

8.5.2 Upsweeps (receiving layer)

During upsweeps, ibed nodes are treated in the exact same way as in the downsweeps (no weights, move along with their father). The big difference now resides in how we handle obed nodes.

Since obed nodes do not have any outgoing edges (which are base for the weight calculation during upsweeps) they have no weight whatsoever. We have

Figure 19: Ibeds moving along on reordering.
no means to decide where we should place them once it’s time to reorder the vertices in the layer.

As stated before in the paper, we found out during the implementation phase, that a good alternative is to use the downsweep weight. This is possible since we know for sure obeds will never lack predecessors.

### 8.6 Tying up the ends

Up to this point we have:

- Added ibed and obed nodes to the graph, as well as normal dummy nodes between them (when needed).
- Reduced the number of crossings in the graph and (hopefully) straightened up all back edges.

As mentioned earlier obed nodes are not tightly coupled to the vertex that spawned them. It is now time to pair obeds to vertices in the layer where they spawned. It is noteworthy to mention that we will not attempt to render these edges at this step in the algorithm, but only pair them up which is needed for the next step.

There are two aspects to be discussed when we decide to pair obeds to vertices; The total relative distance the pairings cover, and if it is possible to draw edges representing those them without crossings.

The total relative distance these pairings cover is of importance, since these edges will be horizontal. The longer these edges become, the higher the chance they will cross the vertical edges between layers. This fact leads us to our first goal in this step; To minimize the total relative distance these pairings cover.

The second aspect to be mentioned is the aesthetical, we want pairings that do not generate edges with crossings between them, since these are more pleasant to the eye. This gives us the second goal of this step; To minimize the number of crossings between obeds and their paired up vertices. For an example see Figure 20 on the following page, where two different setups are shown. Although both are perfect solutions mathematically, the second one is preferable from an aesthetical point.

Minimizing the relative distance is not a hard task. It is easy to see how this problem relates to the assignment problem\(^4\), where the vertices and obed nodes constitute two disjoint sets, with edges between each other with a cost equal to the distance between them.

There are several algorithms that solves this problem, such as the Hungarian algorithm, although they do not take into account the aesthetical requirement we have set.

---

\(^4\)Combinatorial optimization problem where we try to find the matching which yields the maximum / minimum weight in a weighted bipartite graph.
8.6.1 Aesthetical Hungarian

We can easily modify the Hungarian algorithm in order to generate solutions that are also pleasant to the eye. The key is to pick the pair (obed and vertex) with minimum distance, which would not interfere with another perfect pairing.

The first step is to find all the vertices (on the same layer) with a reversed back edge pointing to the same goal, and the obeds that were generated by those. Next step is to put them in a list, in order of appearance, from left to right. Now we can proceed to calculate the relative distance from every obed to every vertex and put these values in a table. We can now pick a minimal pairing (only entry in its column with value 1) which would not break another perfect pairing. The next step is to remove the selected pair from the list, allowing the rest of the nodes to take their position. An example of this method can be seen in Figure 21 on the next page.

8.7 Alterations to the pendulum algorithm

Due to the peculiarity of ibeds and obeds, we need to make a few changes to the pendulum algorithm in order to make it work.

8.7.1 Ibeds

As we remember, ibeds were created without any incoming edges from the layer above them. This could be a bit problematic when trying to balance the graph since they are the starting point for long linear segments.

The solution chosen was to temporarily give them the same incoming edges as the vertex they are bound to. These changes give us the desired secondary effect of keeping back edges close to the targeted vertex, as well as drawing them in a balanced way.
Figure 21: Modified Hungarian
8.7.2 Obeds

Regarding obeds; we have chosen not to make any alterations, such as giving them additional edges pointing to the next layer.

The reasoning behind this is that we believe this would lead to unbalanced graphs. The pendulum algorithm is based on the distance between connected vertices, which should be relatively close to each other, due to the crossing reduction algorithm.

Since obeds relative positionings are fathered from the ibeds above them, we have no means to decide good candidates in the layer below. This leads to obeds (and the linear segment they belong to) having a force of 0 when the balancing algorithm pendulates up.

8.8 Changes to the edge track partitioning

In the normal Sugiyama approach, the partitioning of edge tracks needs to take care of only inter-layer edges, if they are not part of a linear segment. By introducing horizontal edges, bound to the ibed and obed nodes, we need to make some changes.

The goal is to distribute the edge tracks in a way that feels “natural” for the user, allowing him to quickly recognize the edge type by looking at its positioning.

With our approach we now have three sets of edges to take into account:

- Edges going from outbound ports to obeds.
- Inter-layer edges.
- Edges going from ibeds to inbound ports.

We have chosen to deal with this by introducing a priority method.

8.8.1 A priority approach

Since we are dealing with three distinct sets of edges we chose to partition the space between layers into three partitions (see Figure 22 on the following page);

- The space closest to the outbound ports on the top layer, which we chose to call the outbound partition.
- The space closest to the inbound ports on the bottom layer. The inbound partition.
- The space in between the inbound segment and the outbound segment, the middle partition.

Assigning a partition to each of the edge types was almost trivial. We chose to assign the outbound partition to edges going from outbound ports to obeds.
The middle partition was assigned to inter-layer edges and the inbound partition to edges going from inbound ports to ibeds.

Assigning the inbound partition to ibed edges was an easy pick, mostly based on the fact that we want their horizontal edges closest to the inbound ports. A similar reasoning was behind the assignment of the outbound partition to the obeds and their edges. An example of a possible result can be seen in Figure 23 on the next page.

It should be noted that the edge track reduction and assignment should be done w/o mixing edges from different partitions.

8.8.2 Distributing tracks to obeds and ibeds

In order to improve the readability of the graph, we should distribute the tracks within the inbound and the outbound partitions in a smart way. The main goal being to reduce the number of crossings between the edges occupying these tracks.

Distributing the tracks within the inbound partition is more or less trivial. Since the origin of the edges are always from the left of their destination, we can assume the same order as given by the normal track assignment algorithm. This is done since we want to preserve the notion of “further away from the node, the bigger the jump”.

Tracks residing in the outbound partition are a bit trickier. As we remember, we have already paired them up in such way that they could be drawn w/o crossings, although only withing their owns groups (all obeds bound to the
same targeted vertex). The main difference now is that we have to take into account the mixing of different groups in the same layer.

Our testings showed us the best way to tackle this problem, is to handle them as we would with any other vertices. Once the distribution of tracks is done in the outbound partition, all we do is invert the order. Those edges assigned to the last track become now assigned to the first one, and so on.

Results of these two methods can be seen in Figure 24 on the following page.

9 Implementation

The implementation of our method was done in C++, using the Qt Framework (http://qt-project.org/) provided by Digia. The reasoning behind this choice was mainly based on our experiences with the Qt Framework, and the ease of use of their rendering engine. We will now describe Qt in short and some details of our implementation.

9.1 Qt

Qt is an application and UI framework used for development across multiple platforms, including mobile devices. It consists of a set of modular cross-platform C++ classes called the Qt libraries. Digia also provides an Integrated Development Environment called Qt Creator IDE, which in fact, was the IDE of choice during the implementation of our code.

9.2 The Graphics View Framework

QGraphicsView, together with QGraphicsScene (both part of the Graphics View Framework) was our rendering engine of choice. The decision was based mainly
on our familiarity with the framework. QGraphicsScene is a class which provides a surface for managing a large number of 2D graphical items. It serves as a container for QGraphicsItems, which can be visualized together with QGraphicsView.

QGraphicsItems are the base class for all graphical items in a QGraphicsScene. It provides a lightweight foundation for implementing custom items. That includes methods for altering their geometry, collision detection and more. The shape can be altered by re-implementing their paint function.

The main method of use of the rendering engine is creating an item (which inherits from a QGraphicsItem) and adding it to the scene with the QGraphicsScene.addItem(QGraphicsItem*) method. Once the scene is in possession of the item, all the rendering will be handled by the scene. That includes repainting the item when necessary. The item in question must have a paint method (usually overridden by the user) which will be used by the scene when a repainting is scheduled.

If the user wants to alter the visual representation of the item, one must call the QGraphicsItem.update() method. This will schedule the item for repainting. In case the shape of the item is about to be altered, the method called QGraphicsItem.prepareGeometryChange() must be called. This is to ensure the whole item is repainted, since QGraphicsView does not repaint the whole area the items cover, if its not necessary.

### 9.2.1 Edges

Since the edges were to be represented as polylines we decided to implement those as a combination of two basic QGraphicsItems.

The arrow pointing to the destination vertex was implemented with the help
of a QGraphicsPolygonItem. The three points which the arrow consists of were recalculated every time the positioning of either the source, or the targeted vertex changed. The line itself was implemented with a QGraphicsPathItem, the points where the polyline would be bent at were stored in a list.

In order to facilitate for the algorithms each edge would hold knowledge of its source and targeted vertex. Edges would also mark if they had been reversed, or if they were to render their arrow or not.

9.2.2 Vertices

Vertices were represented graphically as boxes, thus their item of choice was the QGraphicsRectItem.

Vertices needed to provide a lot of data about themselves in order to work together with the algorithms. Vertices stored all their predecessors, successors, the type of vertex they were (ibed, obed, dummy or normal), which layer they were assigned to as well as their position in such layer. Vertices would also keep track of which ibed nodes they owned.

Obeds needed to keep track of additional data, specifically the vertex they spawned next to and the vertex which gave birth to their corresponding ibed.

Data related to the graphical aspects of the algorithm, such as their absolute position in the plane, was provided by the Qt Graphics View Framework, as well as their width and height.

9.3 About the algorithms

In order to ensure the portability of the method we opted to not use any Qt-specific functions (besides basic rendering ones). If we introduced steps in the algorithms which used engine-dependent functions, such as collision detection, we would be tying all our user to the Qt framework (or similar engines with these functions).

10 Evaluation

Since we believe the performance of our method cannot really be measured in a quantitative way, we decided to opt for informal qualitative interviews, where test subjects were allowed to test our implementation and speak their thoughts. This chapter will present our findings, which are based on the interviews.

10.1 About the interviews

Our group of test subjects consisted of five people, who all accepted to be interviewed. The test subjects were selected based on a single criteria; a background in computer science. This criteria was chosen based on the belief that our test
subjects would be able to give us better feedback, since we could easily explain to them what our test data consisted of, and what exactly was being represented on the graphs.

The interviews were conducted in a way where the users could compare the normal Sugiyama method to our method. We did not give information to our test subjects on which method was the normal, or the altered one. This was done in order to not impose the notion of one of these being “alternative” or “superior”.

Input graphs of different sizes and characteristics were tested. The majority of these were actual graphs generated by a compiler. Some of the input graphs were handcrafted by us, since the graphs generated by the compiler lacked some traits we wanted to expose our test subjects to. Specifically shorter, nested loops.

The questions given during the interviews were focused on how the test subjects perceived the drawing methods. The points we tried to bring into light were mostly the readability and understandability of the graphs, the differences between the methods, as well as what the test users would like to improve in both.

10.2 Results

The following is an overview of the answers and comments given to us by our test subjects. We will begin by presenting the overall view on both methods followed by how the users compared them to each other.

The normal Sugiyama method wasn’t liked all that much by our test subjects, the main reason being its perceived lack of “natural order”, as well as how cluttered it felt for them. In comparison, our method was considered as a better choice, although with some reservations. The most common reaction was how easy it was to spot loops and follow them, compared to the normal method. Many users commented on how natural the representation of loops felt for them, one aspect where the test users considered the normal method to be lacking.

One comment that was shared among the majority of our test users, is how much easier it became to read the same graphs in the altered method, compared to the normal. Some test subjects related this to the structure generated by our method, which separates backedges from the rest of the graph. Test subjects pointed out how our method gave less cluttered graphs, which required less concentration, as well as less backtracking while reading. All of our users remarked how much easier it was to get an overview on the graphs created by the altered method, while a few also pointed out how this could lead to easier memorization of the graphs.
Even though our method was considered superior by our test users, they would not see it as a perfect solution, mostly due to some peculiarities concerning the representation of normal edges. The main culprit was the way we assigned edge tracks. Sometimes, the algorithm would give us problems as described in subsection 12.1. This was pointed out by our users as the most fearful pitfall in our method, since it could easily lead to confusion and make the graph impossible to read for the users.

Another aspect pointed out by our test subjects, was how easily they could identify small changes which would have improved the representation of the graph. The two most recurring points were; unnecessary crossings involving back edges, due to unfortunate placements of nodes (discussed in subsection 12.2) and the port strategy used in our implementation (discussed in subsection 12.5). One interesting aspect is how our test users didn’t feel the same about the normal method, they had no doubts about the need for changes in order to improve the representation of the graph, although they were unable to suggest alterations, as in the alternative method.

Pictures demonstrating both methods can be seen in Figure 25 on the next page and Figure 26 on page 42.

11 Conclusions

In this thesis an alternative interpretation of the Sugiyama framework was presented. It was specifically aimed at the visualization of code flow. The main improvements offered by our method would be the enhanced representation of back edges, which represent loops in the code flow, as well as a less cluttered graph which gives a better overview, as stated by our test subjects.

If we were asked to summarize what we learned while writing this report, our answer would be; Graph drawing is no easy feat. What looks really good on the drawing board, becomes often a horrible idea when put into work. We experienced this first hand, many alterations had to be done to our method before it would present decent results. Tiny changes to the algorithm would often yield huge variations in the quality of the graphs, as well as new unforseen problems.

Another big challenge was the fact that we decided to not use any Qt-specific functionality, this would show up to be a huge crippling factor in our implementation. Many problems we faced while engineering the code would have been easily solved, with for instance, collision detection provided by Qt.

The testing phase gave us two important pieces of knowledge; the first being, our method is by no means perfect and requires a lot of further work before being shipped to users. The second, the positive comments received from our
Figure 25: A small graph.
Figure 26: A bit bigger graph.
test users makes us believe our work is worth exploring further on.

We believe the method presented in this paper would become a really good alternative, if the problems left as future work were to be solved.

12 Future Work

There are a few matters which we did not address, mostly due to time constraints. This chapter will present the problems we decided to leave as future work, as well as their possible solutions.

12.1 Crossings between edge tracks

The Manhattan layout utilized in our method gives us an easy way to avoid edges colliding into vertices. It does however have a downside.

When our method picks a track where an edge should reside, it does not take into account the possible crossings this choice could create. This problem is better described by looking at Figure 27 on the next page. The problem arises when two or more edges with the same orientation, get assigned tracks which creates an unnecessary crossing.

We attempted to reduce this problem to a set of popular graph problems with no positive results. There is however a way of solving this, although it would require the need of the Qt rendering engine. This would be to try all possible permutations of edge tracks (with as many tracks as the number of edges between the two layers), and ask the rendering engine how many crossing there are between edges. The permutation which gives us the least number of crossings, and tracks used, would be the solution to the problem. The exploration of all permutations may become problematic, in terms of time performance, when there is a large number of edges between layers. In that case a greedy trade transposition could become a feasible option, although not optimal.

12.2 Positioning of edges relative to their obed nodes

One room for improvement found during the testing phase, is the way nodes get placed in the graph relative to their obed nodes. The situation can be easily described by looking at Figure 28 on page 45. As the reader may remember, obed are not tied to the node that spawned it during the crossing reduction step. Instead, obed and normal nodes are paired up later in the algorithm. Obeds' relative positioning in their layer (decided during the cross reduction phase), is based entirely on the relative positioning of their matching ibed. As seen in the picture, this leads sometimes to layouts where one simple switch would remove an unnecessary crossing.

We believe the key to this problem relies in the way we calculate weights during the crossing reduction step. The final goal, bringing obed closer to their
paired up node, implies we need to somehow tie them already while reducing crossings. One possible approach could be to temporarily add to normal nodes, the incoming edges their corresponding obeds have. This should in theory, affect the weight calculation of normal nodes in such way, that the algorithm attempts to bring them closer to their obeds. Once the crossing reduction step is done, these new connections should be discarded from the graph.

12.3 Handling inevitable edge crossings

There are times when the crossing reduction algorithm will not find a setting without edges crossing. Our implementation does not handle these cases in any particular way, this was shown to us (by our test subjects), to be a source of confusion when reading the graph. A clear improvement would be to somehow alter the appearance of these edges, making them visually distinct to the user.

One of the peculiarities of the Manhattan layout utilized in our implementation, is the fact that edges can only cross each other in a perpendicular way (besides in the worst case scenario). We can easily exploit this fact in order to improve the visualization of edge crossings.

A possible approach would be one similar to what is seen in Figure 29 on the next page. This method requires knowledge about where the edges meet, information which could be easily extracted from the Qt-rendering engine. If no engine, or similar tool, is available a more archaic method could be used.

Based on the positioning of the vertices on each layer, together with their corresponding edges, a list of all crossing edges could be calculated. The next step would be to, together with the edge track assigned to each edge, decide...
which edge crosses at its vertical or horizontal section. The $x$-positioning of the meeting point is given by the vertical section of the edge being crossed, while the $y$-positioning is given by the track assigned to the edge, who's horizontal section is crossing the other edge.

12.4 Worst case scenario in the Manhattan layout

There is one situation, encountered during the testing phase, which could be deemed to be the absolutely worst case scenario for our method. This can be better explained with a picture, seen in Figure 30 on the following page. The problem occurs when two crossing edges share the same $x$-coordinate for the outbound, respectively inbound ports. Our method will assign them tracks in a way it becomes almost impossible to tell them apart, creating confusion.
They key to the problem resides in the way we balance the graph. An adjustment would need to be done at the balancing stage, it would need to take into account the positioning of edges and their endpoints, as well as how much these adjustments would skew the balance of the graph. Unfortunately, we do not have any suggested methods for solving this problem.

12.5 Better port strategies

One aspect we did not cover in our report, is the different strategies that can be applied to port distribution. The strategy we adopted was to have all outgoing edges share the same outbound port. We decided to let incoming back edges have their own port, placed a bit closer to the left edge of the vertex, while normal incoming edges would all share the normal inbound port. This was done since we believe it would bring more clarity to the graph.

One approach we would have liked to explore, is having a flexible amount of outbound and inbound ports. Factors like, how many outgoing edges a vertex has, the direction of these edges, as well as other, could decide the number of ports a vertex gets. This could be taken even further as to deciding which port each edge gets, mostly based on their direction and type of edge.

**List of Algorithms**

1. Depth-first based Remove Cycles .............................................. 10
2. The Longest-Path algorithm, adapted from Healy and Nikolov, 2013 ................................................................. 13
3. PromoteVertex, adapted from Healy and Nikolov, 2013 ..... 14
4. Vertex-Promoting Heuristic, adapted from Healy and Nikolov, 2013 14
5. Insertion of dummy nodes ......................................................... 16
7. Transposing algorithm .............................................................. 19
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example of code and its corresponding CFG</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Directed graph traversed with a DFS (visiting order specified by vertex numbering)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Reversal of back edges after Sugiyama</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Improved Sugiyama</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Sugiyama Algorithm. From Cycle breaking to Coordinate assignment</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Cycle Breaking DFS</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Assigning layers</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Vertex promotion algorithm</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>Inserting dummy nodes</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>Layer-by-layer sweeping</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>Graph divided into linear segments</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Birth and death of regions</td>
<td>22</td>
</tr>
<tr>
<td>13</td>
<td>Ports and their positioning</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>Avoiding obstacles</td>
<td>24</td>
</tr>
<tr>
<td>15</td>
<td>Safe zone between layers. Horizontal edges allowed</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>Edge tracks</td>
<td>25</td>
</tr>
<tr>
<td>17</td>
<td>Introducing obeds and ibeds</td>
<td>27</td>
</tr>
<tr>
<td>18</td>
<td>Inserting obeds and ibeds</td>
<td>29</td>
</tr>
<tr>
<td>19</td>
<td>Ibeds moving along on reordering</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>Two possible pairings with the same total cost</td>
<td>32</td>
</tr>
<tr>
<td>21</td>
<td>Modified Hungarian</td>
<td>33</td>
</tr>
<tr>
<td>22</td>
<td>Partitioning the edge tracks</td>
<td>35</td>
</tr>
<tr>
<td>23</td>
<td>Edge tracks in their partitions</td>
<td>36</td>
</tr>
<tr>
<td>24</td>
<td>Assigning tracks to obeds and ibeds</td>
<td>37</td>
</tr>
<tr>
<td>25</td>
<td>A small graph</td>
<td>41</td>
</tr>
<tr>
<td>26</td>
<td>A bit bigger graph</td>
<td>42</td>
</tr>
<tr>
<td>27</td>
<td>Edge tracks creating unnecessary crossings</td>
<td>44</td>
</tr>
<tr>
<td>28</td>
<td>Alternative obed positioning</td>
<td>45</td>
</tr>
<tr>
<td>29</td>
<td>Handling edge crossings</td>
<td>45</td>
</tr>
<tr>
<td>30</td>
<td>Worst case scenario</td>
<td>46</td>
</tr>
</tbody>
</table>

## References


