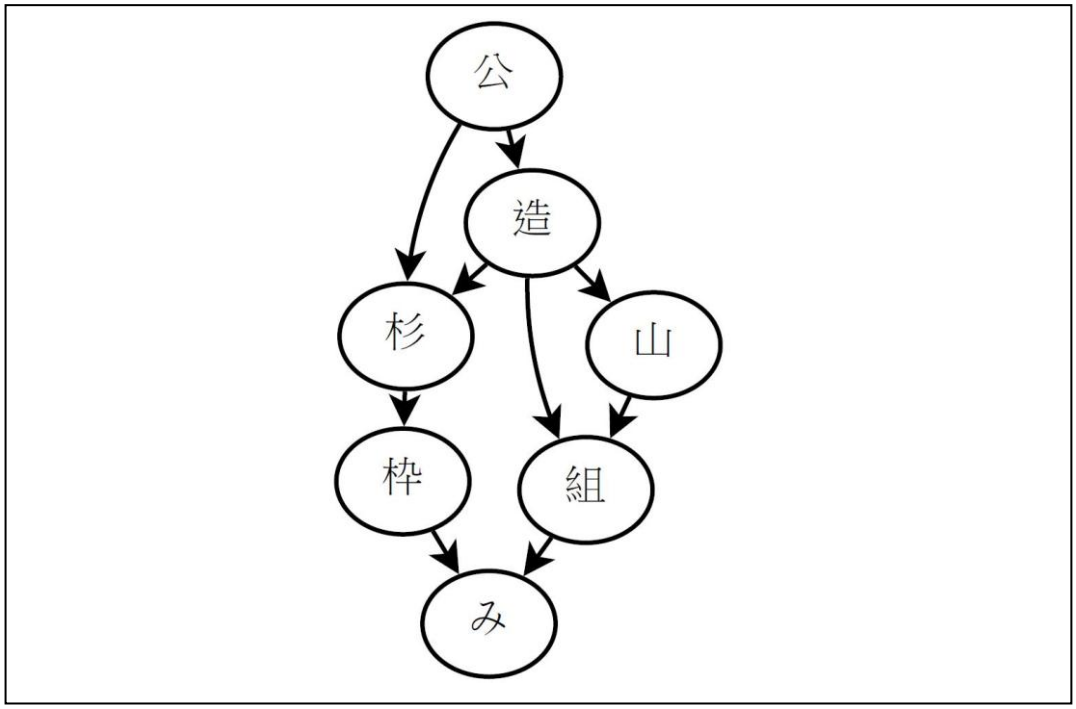
CHALMERS



Visualisation of state machines using the Sugiyama framework

Master of Science Thesis in Computer Science

VIKTOR MAZETTI
HANNES SÖRENSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2012

Visualisation of state machines using the Sugiyama framework

VIKTOR MAZETTI
HANNES SÖRENSSON

Examiner: Peter Damaschke

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
A layered graph where the vertices spell “Kozo Sugiyama Framework”.

Department of Computer Science and Engineering
Göteborg, Sweden June 2012

Abstract

In this thesis a tool was developed to help Ericsson visualise state machines. In this process an evaluation of graph layout algorithms within the Sugiyama framework, a suitable method for drawing state machine diagrams, was conducted. The framework consists of four steps which are implemented and tested according to some chosen criteria for graph layouts. The steps are also tested in relation to each other so as to see if an algorithm used in one step can impact the quality of the solution for subsequent steps.
Interestingly enough we saw that a seemingly worse performing algorithm in one step could actually improve the quality of the next step. We discuss these results and suggest that future work is needed to investigate if there are some properties of a graph that can indicate how well subsequent steps can perform.

Acknowledgements

We would like to thank our supervisor at Ericsson, Tomas Nilsson, for his help and guidance throughout the project as well as our manager Gordon Sun for making our work there possible.

We would also like to thank our supervisor and examiner at Chalmers, Peter Damaschke, for theoretical guidance and for providing feedback on the report as it developed.

Contents

List of Figures

List of Tables

List of Algorithms

This page intentionally left blank.

1 Introduction

1.1 Background

EPG (Evolved Packet Gateway) is a network node developed by Ericsson that connects a mobile network to the Internet. A lot of functionality exists in this node, such as various forms of charging, packet inspection, communication with different external servers, and so on. So the EPG software application contains a lot of sometimes complex logic that deals with these things. Also high performance is important, so the code is written in C and C++.

One way of reducing the complexity of the logic is to use state machines to model it. A state machine is a well defined model that consists of a number of logical states and the possible transitions between these states. A variant of this is used in many places in the EPG application. A state machine framework written in C is used to define the states and transitions of the state machine in the code. While powerful, it is quite difficult to use since the states and transitions are defined by manually creating C arrays containing function pointers and index values. This makes it hard to see how the states and transitions actually interact.

To solve this problem Ericsson wants a tool that can read these state machines and visualise them as state diagrams. The tool should be in the form of a plugin for the Eclipse IDE (integrated development environment) used at Ericsson. Since the code is updated frequently the diagrams need to be generated automatically and quickly. For the tool to satisfy these criteria while maintaining a good drawing quality it needs a good graph layout algorithm.

The field of graph layout algorithms is vast and involves a wide array of different solutions and approaches to the graph drawing problem. There is no final solution that fits all types of graphs; graphs like state diagrams, however, have certain properties, primarily directionality, for which the Sugiyama framework [1], or layered graph drawing, is most suitable [2].

1.2 Purpose

Ericsson needs a way of visualising their state machines in an easy way to facilitate development. Therefore we aim to develop a plugin application for the Eclipse IDE that can parse Ericsson's state machine framework and draw the corresponding state diagram.

An integral part of drawing the graph is to have a good layout algorithm. To find this, we intend to explore the Sugiyama framework [1] for drawing graphs, such as state diagrams. We will evaluate different approaches to implementing this framework with Ericsson's state machines using a set of criteria commonly used in the area of graph layouts.

The Sugiyama framework divides the graph layout problem into several smaller problems. Most research already done on the subject tends to review and compare algorithms only within these smaller subproblems and not compare the whole combination. We aim to do both in our study. We also aim to investigate further within the framework how exact solutions compare to the different heuristics that can be used.

1.3 Limitations

Our work, though applicable to graphs in general, mainly focuses on graphs defined by Ericsson's state machines or those similar to them, whether in size or structure. We hope that this will help limit the scope of the thesis and allow us to draw more specific conclusions.

Our implementations of the various algorithms will not necessarily be optimised in terms of efficient data structures. The focus is on replicating the behaviour of the the algorithms and to discuss their theoretical properties, not to ensure the fastest possible running time. This limitation is due to time as well as the chosen platform (the Java programming language) being relatively high-level.

1.4 Method

At first a preliminary study of graph drawing was conducted which led to the chosen focus on the Sugiyama framework for hierarchical graphs. The reason for this focus was that the Sugiyama framework seemed to be the most popular method for drawing directed graphs.

The main work consisted of two parts: development of the plugin for Ericsson and finding a good layout algorithm, which in turn consisted of a literature study, implementation and testing. These were done concurrently throughout the project.

We have implemented a visualisation tool as a plugin for Eclipse, using Eclipse itself and the Java programming language. In the beginning a first prototype of the plugin was developed with basic functionality such as the ability to parse Ericsson's state machines and render graphs of them using an existing graph layout program to provide layout data, such as vertex coordinates, for the drawing. The plugin was then released as a beta-version to some chosen developers at Ericsson for testing who provided feedback which guided the continuing development.

We also conducted a literature study of the field of graph drawing, with a focus on the Sugiyama framework. This led to discovering different approaches to solving the various sub-problems defined in the framework. Some of the most common and effective approaches were chosen for implementation and testing.

Implementation of different graph layout algorithms was done within the plugin since it provided a suitable test environment, having the basic functionality of rendering the graph already in place.

In order to evaluate the chosen algorithms a test set consisting of the Ericsson graphs, as well as others, was assembled and several performance measures were chosen. Comparison between algorithms also centered around their theoretical properties, such as time complexity.

The following tools and pieces of software were used in the project:

- **Graphviz** - A collection of graph drawing tools, of which *dot* is one. Maintained by AT&T.

  `http://www.graphviz.org/`

- **GLPK (GNU Linear Programming Kit)** - A free software LP solver.

`http://www.gnu.org/software/glpk/`

- **GLPK for Java** - A Java binding for GLPK.

  `http://glpk-java.sourceforge.net/`

- **Eclipse** - An IDE (Integrated Development Environment). Used for Java and C/C++ amongst others. Also contains Eclipse CDT (C/C++ Development Tools) which can be used for parsing C/C++.

  `http://eclipse.org/`

1.5 Outline

In this report we will start by going through some of the theory necessary to understand the thesis. This theory includes a definition of Ericsson's state machine framework, a quick overview of the graph layout field with the Sugiyama framework in focus, and finally a short introduction to linear programming. After that we will cover the implementation stage which is presented in two chapters about the developed plugin and the implemented layout algorithms. Finally we will walk you through and discuss the benchmark results and draw up a conclusion.

2 Theory

2.1 The state machine framework

Formally, a state machine as defined by the framework used at Ericsson is an 8-tuple $M = (S, \Sigma_R, F_s, \Sigma_E, F_t, T, G, H)$ where

- S a finite set of states.
- Σ_R a finite set of raw events - these are the external input that the machine receives.
- F_s a finite set of state functions - for each current state and raw event, at most one of these is run.
- Σ_E a finite set of events - these are generated by the state functions and can be considered "internal" input.
- F_t a finite set of transition function - these are run during a specific transition between two states. These may also generate an event, which can then lead to an immediate transition when reaching the next state.
- $T : S \times \Sigma_R \to F_s$ mapping a current state and a raw event to a state function name.
- $G : S \times \Sigma_E \to S$ mapping a current state and an event to the next state.
- $H : S \times S \to F_t$ a partial function mapping an existing transition to a transition function name.

T and G may be thought of as partial functions, since many of the combinations are considered invalid in some way. H is necessarily partial since there is no requirement that G defines a transition between each (ordered) pair of states.

One may note that the above does not properly describe a traditional state machine, since there does not exist a direct connection between on the one hand input (raw events) and current state and on the other hand transitions and output. The actual flow is hidden in the functionality of the state functions, which may or may not, possibly depending on external factors, generate an event leading to a transition.

In the code, S, Σ_R and Σ_E are represented by `enum` declarations, F_s and F_t are simple arrays with pointers to C functions and T, G and H are matrices (in the form of arrays) containing an index into F_s, S or F_t respectively. Invalid combinations (which should not be shown in the state diagram) are marked by a special value.

During our work at Ericsson, the representation described above was replaced in some code by a simpler one where the information was stored in a few lists of tuples, with invalid combinations simply omitted. The logical components are still the same as described.

Our visual representation is a, slightly modified, state diagram. States are vertices, whose labels contain the name of the state and one row for each (valid) raw event name/state function name pair. Transitions are edges with labels containing one or more names of events that cause it and one transition function name. See figure 1 for a simple example.

A possible execution for the state machine represented by figure 1 is the following:

- The machine starts in state A and receives raw event 1.

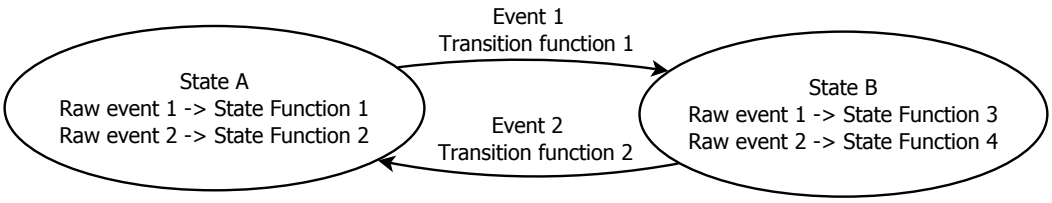


Figure 1: Example of state machine

- This causes state function 1 to run. Here it is, as previously mentioned, not clear from the diagram what might happen. Let's assume that the state function generates event 1.
- This leads to a transition to state B as well as the running of transition function 1.
- Assume that raw event 1 is received again. Let's say that this does not lead to any transition, since no event is generated by state function 3.
- Raw event 2 is received.
- State function 4 is run, which generates event 2.
- The machine transitions to state A and transition function 2 is run.

There are nine state machines in use at Ericsson which we aim to visualise. They are not very large; they are on the order of 10-20 states and 20-50 transitions.

2.2 Graph definitions

A *graph* $G = (V, E)$ is a pair of a set of *vertices* and a set of *edges*. An edge $e = (v, w) \in E$ is an ordered pair of vertices from the set V. In our terminology, the word graph is synonymous with *directed graph*, or *digraph*. Usually there is a distinction made between these and *undirected* graphs where the edge pairs are not ordered. State diagrams are graphs according to this definition.

Additionally, we mention *acyclic* graphs. These are graphs that contain no cycles, i.e. there is no directed path $((v_1, v_2), (v_2, v_3) \dots (v_n, v_1))$ (sequence of edges in E) starting from a vertex v_1 and returning to itself. Directed acyclic graphs are often called simply DAGs.

When describing algorithms for graphs one often works with a vertex, v, and its connected edges $E(v)$ and neighbouring vertices $V(v)$. Since we are working with directed graphs however we often separate these two sets into incoming and outgoing. For these we use the notation $E^+(v)$ for edges with v as destination and $V^+(v)$ for the vertices which are the sources of those edges. Respectively for outgoing we use $E^-(v)$ and $V^-(v)$.

2.3 Criteria for good graph layouts

Measuring the quality of graph layouts is a complex and most subjective field. There is no known optimal measure for deciding how good or bad a graph layout is, but there are some criteria that are commonly used in graph research. A few of these are as follows: [2,3]

Edge crossings A drawing with few edge crossings is easier to read and follow than one with many.

Bounding box of graph A drawing should be as small as possible while maintaining readability.

Symmetry A drawing should ideally show the underlying symmetry of the graph. However, there are no easily deduced automatic measures for symmetry.

Simple edges Angles of edge bends should be minimised to improve readability. Angles of more than 90 degrees should be avoided.

Length of edges The length of edges should be minimised. A drawing with a lot of long edges can be hard to follow.

Graph flow The flow of the graph should be preserved meaning edges should point in a uniform direction.

We should note that many of these criteria can conflict with each other - to find the minimum number of edge crossings, for example, you may have to increase the size of your layout by an unreasonable amount. There may also be additional or alternative important criteria; for example, recent studies have shown that maximising the angles of the remaining edge crossings after crossing minimisation improves readability [4].

2.4 The Sugiyama framework

The Sugiyama method, introduced by Sugiyama et al. in 1981 [1], is one of the most popular graph drawing schemes. Terms like "layered graph layout" are often used synonymously with "Sugiyama method" or "Sugiyama framework". A layered layout places the vertices in layers where edges only go between different layers.

The idea of the framework is to divide the task of drawing a graph into several subproblems, most of which closely resemble other well known problems within computer science. That way one can use algorithms for the similar problems to solve the Sugiyama subproblems and thereby simplify the graph layout process. The different steps of the method are illustrated in figure 2. They are the following:

(a-b) Cycle removal First the possibly cyclic graph must be made acyclic by removing cycles, done by reversing some edges.

(b-c) Layer assignment Second, the vertices are assigned to layers and dummy vertices and dummy edges are introduced for every edge that spans over more than two layers so as to create a *proper layering* [2], i.e. one where every edge has its endpoints in adjacent layers.

(c-d) Vertex ordering Third, the vertices are ordered within their layers to minimise edge crossings.

(d-e) Coordinate assignment Fourth and last, the vertices are assigned coordinates to create a balanced graph.

All these four steps will be described further in this chapter.

This division of the problem makes it possible to use different approaches to the different subproblems and thus achieve different results. However, solving each subproblem to (some

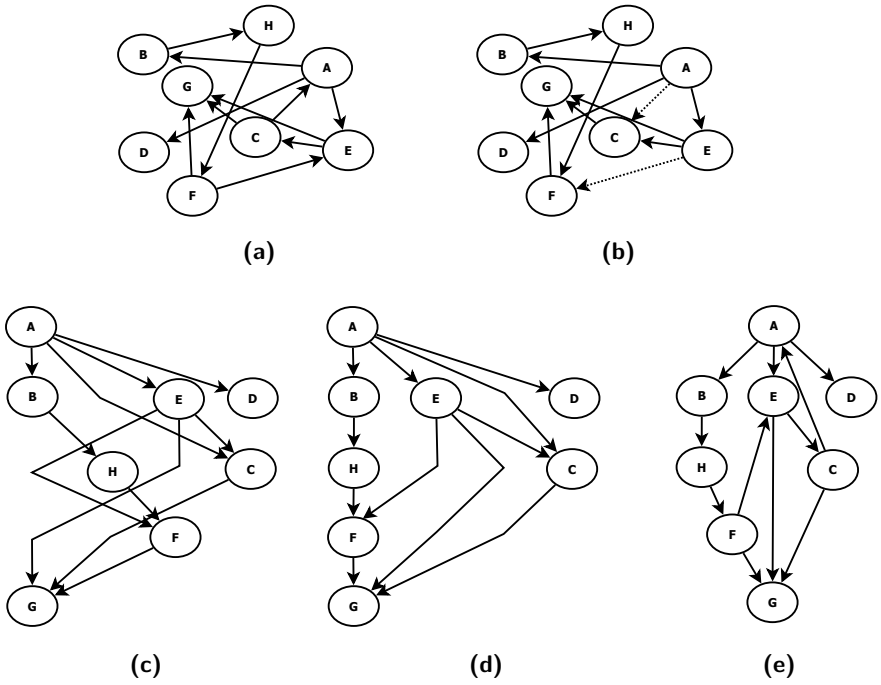


Figure 2: The steps of the Sugiyama framework.

measure of) optimality does not necessarily mean that the main problem (that of graph layout) is also solved optimally. This is especially so since each subproblem objective may clash and interfere with the others.

The framework has proven to be very popular and many implementations exist, including *dot* [3], *Microsoft Automatic Graph Layout* [5] and *Tulip* [6].

There exist also several variations and additions to this framework. The coordinate assignment step, though now considered standard, was handled as a post-processing part in the ordering step in Sugiyama et al.'s original paper [1]. Often there is also a fifth step [3]; that of converting edges into splines, a spline being a smooth polynomial interpolation of a list of points (an edge) making the drawing more aesthetically pleasing. A slight variation necessary in order to handle edge labels is to create dummy vertices for each label next to its edge. The framework can also be used to create radial drawings where each new layer is placed in a surrounding circle instead of a row below [7]. These are but a few examples and many more exist [2].

2.4.1 Cycle removal

Objective: Make the graph acyclic by reversing as few edges as possible.

Subsequent steps in the Sugiyama method require that the graph is acyclic. To achieve this, edges are reversed temporarily during the layout process and, of course, restored at the end. Here it is desirable to keep the number of reversed edges small so that the hierarchical nature of the graph is expressed as clearly as possible in the final drawing.

The maximum acyclic subgraph problem is formulated thus: given a digraph $G = (V, E)$

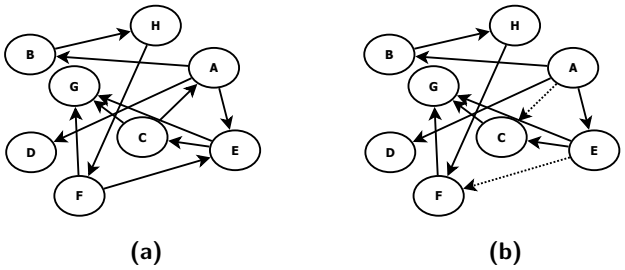


Figure 3: Before and after cycle removal.

find a set $E_a \subset E$ of maximum size such that the graph (V, E_a) contains no cycles. This is often known alternatively as the *minimum feedback arc set* (FAS) problem, formulated thus: For a graph $G = (V, E)$ find a minimum set $E_f \subset E$ such that the graph $(V, E \setminus E_f)$ contains no cycles. This was one of Karp's original NP-hard problems [8].

A recent result has shown the feedback arc set problem to be fixed-parameter tractable in the number of cycles [9]. What this means is that there exists an algorithm for which the running time is bounded by $f(k)|x|$, where f is some computable function (typically a single exponential), $|x|$ the size of the input and k the parameter (in this case, the number of cycles in the graph). Intuitively, one can say that the exponentiality is "contained" in the parameter instead of applying to the whole input.

The aforementioned intractability of the problem is not a big obstacle in this context since an optimal solution does not necessarily translate into a noticeably better drawing. Therefore polynomial-time heuristics are commonly used. Many such heuristics exist for this problem with different approximation bounds for the size of the set and some of these will be explored in the following chapters.

2.4.2 Layer assignment

Objective: Assign each vertex a layer so that each edge stretches between two different layers.

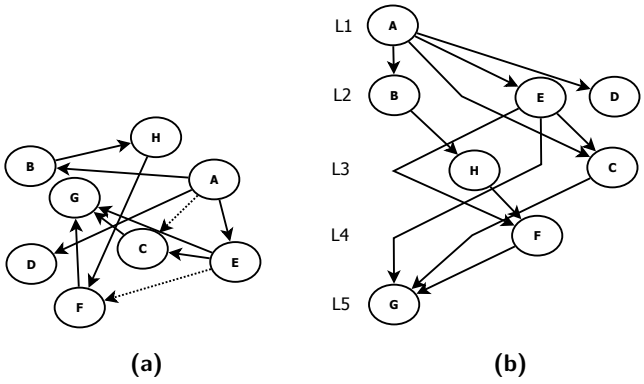


Figure 4: Before and after layer assignment.

A *layering* of a graph $G = (V, E)$ is a partitioning $L = \{L_1, L_2, \ldots, L_h\}$ of V such that

for each edge (v, w), with $v \in L_i$ and $w \in L_j$, $j > i$. That is, what we wish to do is assign exactly one layer to each vertex such that there are no edges between vertices in the same layer. Additionally, we want our layering to be *proper*, i.e. one where every edge has its endpoints in adjacent layers - for this we must introduce dummy vertices and edges.

Dummy vertices and edges are introduced for those edges that span over multiple layers so as to make the layering proper. This is a necessary assumption in subsequent steps. These dummy vertices are merged together again in the coordinate assignment stage.

It is desirable to keep the number of dummy vertices created small. This is because the running times of subsequent steps in the Sugiyama framework often depend on the total number of vertices (including dummies). Also, a small number of dummy vertices means that the edges are short and should thus be easier to follow. This is therefore a useful criteria in judging the performance of a specific layer assignment algorithm.

The problem of layer assignment when trying to minimise the height is very similar to the NP-hard *job shop scheduling* problem with job dependencies. That is where you are given n jobs that may be dependent on other jobs, and m machines on which to divide these jobs in order to minimise the makespan. This makes it possible to use scheduling algorithms also for the layer assignment problem with a maximum width of vertices corresponding to the m machines.

The layer assignment problem can be solved by using heuristics to try to minimise different properties of the final graph such as height, width and edge-length among others.

2.4.3 Vertex ordering

Objective: Order the vertices in every layer so as to minimise the number of edge crossings between all layers.

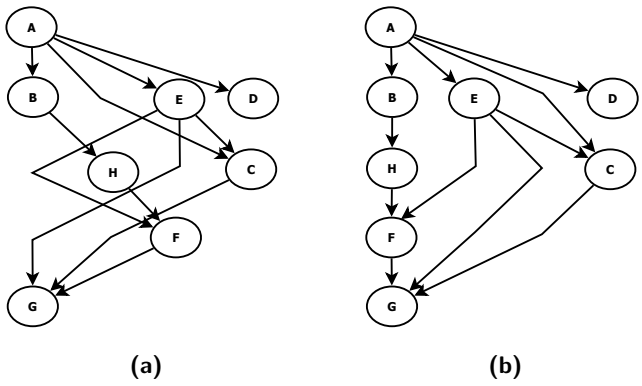


Figure 5: Before and after vertex ordering.

The main criteria for vertex ordering is the number of *crossings* as mentioned in section 2.3. Minimising crossings in the graph can improve its readability since each crossing clutters the graph and increases the risk of misinterpretation. In particular, crossing edges that are close to parallel are very hard to follow.

Ideally, we want to have no crossings; however, not all graphs are *planar*, i.e. there does not always exist a drawing of a graph without crossings. The equation in 1, given by

Pach and Tóth [10], shows a relationship between the size of a graph and its minimum number of crossings. Additionally, the layering constraint provided by the previous step in the Sugiyama framework may not admit a drawing with as few crossings as is otherwise possible for the graph. Despite this, we try to draw with as few crossings as possible.

$$crossings(G) \geq \frac{e^3}{64n^2}, \ \ if \ \ e > 4n \tag{1}$$

Vertex ordering, or crossing minimisation, is often referred to as *multi-layer crossing minimisation* (MLCM). Regrettably, the MLCM problem is NP-hard even in the simple case of only two layers (TLCM) [11] and when one layer is fixed (OLCM) [12]. Therefore, heuristics are often used.

Most heuristics treat the problem as several *one-sided layer minimisation* (OLCM) subproblems by considering each pair of adjacent layers on its own and sweeping through the layers from top to bottom. An exact solution on the other hand considers the whole problem as MLCM.

2.4.4 Coordinate assignment

Objective: Assign vertices x- and y-coordinates so as to straighten edges and minimise graph size.

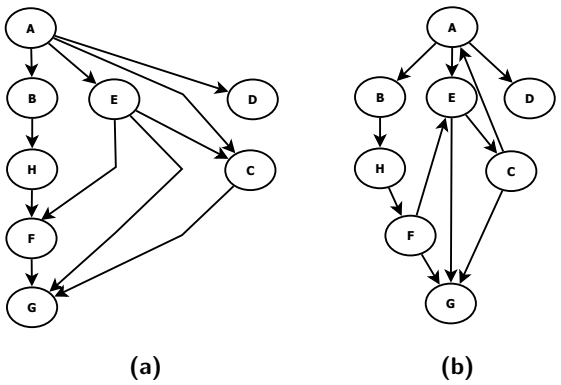


Figure 6: Before and after coordinate assignment. Original edge orientations are restored.

This is the final step where the actual positions for the vertices are computed. Assigning y-coordinates for each vertex is trivial since it is simply based on which layer the vertex is in. Assigning x-coordinates however is more complicated since one wants to both minimise the size of the graph and the number of zig-zagging edges through the layers, by ensuring that long edges are as straight as possible. This is done while also maintaining the ordering of the vertices so as to keep the crossings minimised from the previous step. The solution also takes into account a minimum separation width between vertices and edges. The last steps in this subproblem consist of merging together connected dummy vertices into their corresponding long edge while saving the computed coordinates as points in the edge as well as restoring the orientation of edges that were reversed during cycle removal.

The heuristic solutions for coordinate assignment are often similar to those of the vertex ordering problem since both problems can be approximately solved by considering the problem layer by layer. Solutions for this problem often focus on criteria such as the total size of the drawing and the complexity of its edges, represented by the number of bends and their angles.

2.5 Alternative solutions

There exist several other approaches to the problem of graph drawing. These are mentioned here in order to give an overview of the area and a background to the chosen focus of this thesis on the Sugiyama framework.

2.5.1 Force-directed algorithms

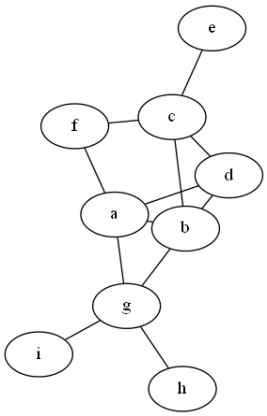


Figure 7: Output from the *neato* tool in Graphviz which uses the approach of Kamada and Kawai [13].

One of the more popular [2] kinds of layout algorithms is those based on a mechanical system of forces. The main principle behind these algorithms is the use of attractive and repellent forces between vertices, with the vertices being iteratively moved until the system reaches or is close to an equilibrium. Usually vertices are replaced with electrons, making them repel other vertices, and edges with springs, making them attract connected vertices. Now we can use the physical laws of electromagnetism and springs (Hooke's law) to calculate the forces directed at each vertex. If we iteratively calculate these forces and move the vertices respectively then we will eventually be close to an equilibrium, meaning that the vertices have now been placed in such a way that the attractive and repellent forces cancel each other out.

The different algorithms within this area of force-based algorithms primarily differ in the way this system is calculated. Three of these algorithms are those of Eades, Kamada-Kawai and Fruchterman-Reingold which we will describe further.

Eades was the first to present an algorithm with this force-based approach, often called the spring embedder model [14]. Eades uses springs for both attractive and repellent forces. The springs between connected vertices are of unit length while springs between unconnected vertices are of infinite length. To calculate attractive and repulsive forces he uses his own formula instead of Hooke's law.

The key points with the algorithm of **Kamada & Kawai** [13] is that it uses an entirely spring-based system and that it only moves one vertex at a time. In this system there is a spring between every vertex even if they are not connected with the length of the spring being based on the shortest path (graph theoretical distance) between the two vertices. The basic steps of the algorithm are as follows:

1. Pick the vertex with the highest energy.
2. Use Newton-Raphson's method [15] to find its position closest to equilibrium.
3. Repeat

Fruchterman & Reingold [16] use a model where vertices are electrons, exerting repellent forces against each other, and edges are springs as in the examples previously mentioned. The key difference of this algorithm is that:

1. Displacement is calculated for all vertices every iteration as in Eades.
2. A temperature that cools for every iteration is used to slow down the process towards the final iterations.

Force-directed approaches are especially useful when visualising large amounts of data, such as social networks. Other advantages include a relatively simple implementation and fast running time. They excel at showing graph symmetry, which is a very important criteria and hard to achieve otherwise. A disadvantage is that the layout produced can be far from optimal depending on the initial layout, which is often randomly generated. The methods do not consider the directionality of the edges so hierarchical properties of the graph are not clearly expressed.

2.5.2 Orthogonal layout

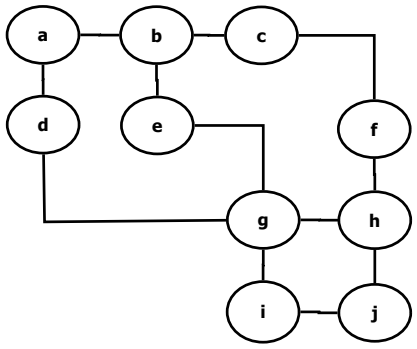


Figure 8: An example of an orthogonal layout.

Vertices are arranged in a regular pattern, where each edge consists of alternating horizontal and vertical segments. Dummy vertices are inserted at crossing points in order to make the graph planar. In representations where a vertex is a point, the maximum degree of a graph that can be drawn is four; this can be solved if rectangular areas are allowed.

This type of layout, also called Manhattan layout, originated in *Very Large Scale Interaction* (VLSI) contexts in the 1970s [2]. In this application, the vertices are fixed and it is the edges (or wires) that need to be placed while minimising their length.

2.6 Linear Programming

A *linear program* (LP) is a mathematical formulation of an optimisation problem. They can always be expressed on the following canonical form [17]:

$$\begin{aligned} \text{maximise} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ \text{and} \quad & x \geq 0 \end{aligned} \tag{2}$$

where the first line is the objective function, consisting of a vector c of coefficients and a vector x of variables, the second line contains a matrix A and a vector b who together define the linear constraints and the third line which constrains the variables to be non-negative. For minimisation problems, values can easily be adjusted to fit into the definition.

Fast algorithms (in both the theoretical and practical sense) exist for solving this kind of problem, when the variables are allowed to take real values. When an integer constraint is added the problem becomes computationally hard, however, and no general polynomial algorithm exists. Problems with this constraint are called *Integer Linear Programs* (ILP).

3 State Machine Visualisation plugin

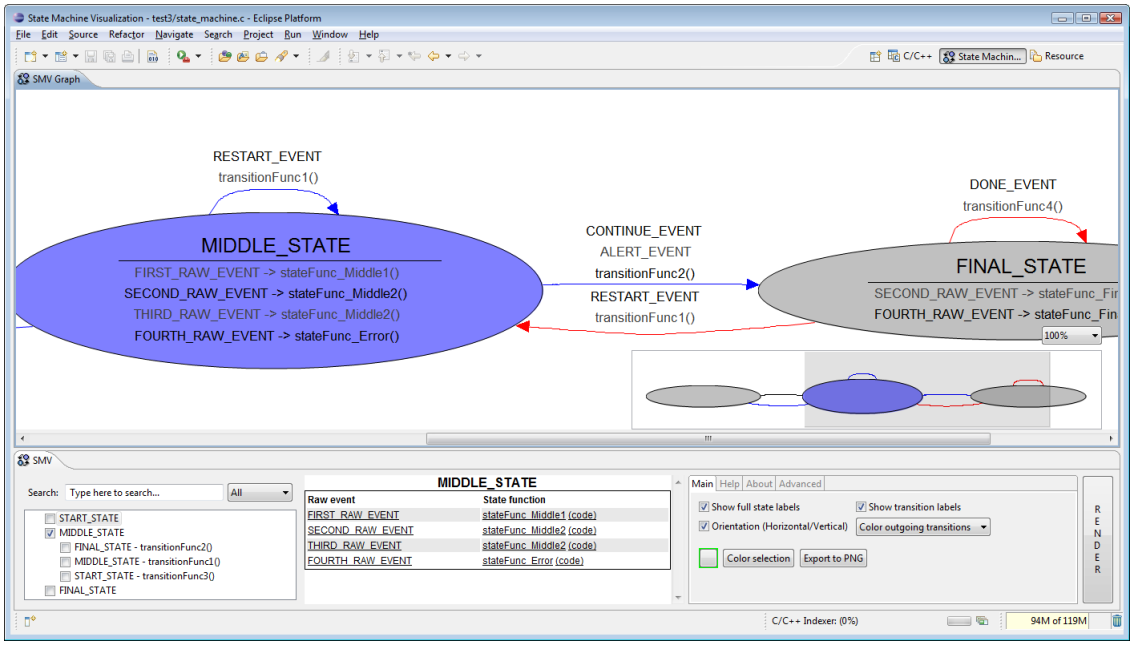


Figure 9: An instance of Eclipse running the plugin.

Previously, Ericsson had used a script written in C that parsed through some of the state machines and output DOT language (a description language for graphs) files that one then manually invoked *dot* (the layout tool which uses the Sugiyama framework) on in order to get state diagram images. This limited the ease of interaction with the development environment (Eclipse) and allowed none of the functionality possible when integrating the state diagram into an application.

The diagram produced by this script was used as a stylistic starting point for our program. The same format for e.g. displaying labels remained in our implementation, as seen in figure 9 and described in section 2.1.

Some of the notable features of the plugin include:

- Rendering of diagram from state machine defined in code.
- Clickable functions in the diagram redirect to definition in code
- Search function for text in graph labels, like events and functions.
- Colouring the diagram and exporting to image (for documentation purposes).

Some of these are elaborated further below. Together they make the plugin more dynamic and helpful than the old static script.

3.1 Parsing

Some parsing of C files is done by the plugin to retrieve the different constructs of the state machine (detailed in section 2.1). The built-in Eclipse CDT (C/C++ Development Tools) parser (which ordinarily points out syntax errors etc. to a developer) is here used to provide an abstract syntax tree of the file which is traversed using a visitor pattern.

The actual code at Ericsson that uses the state machine framework consists of relatively few files which are old and unlikely to change in any major way within the nearest future. This made feasible, in the short term, the approach of hard-coding several elements of the parser. The aim, however, was to construct a parser that, as far as possible, made no assumptions about the code other than that it adhered to the actual framework. A few concessions were made in this regard where necessary, such as assuming a specific name of a construct or that values were given in a specific way.

One challenge encountered in the parsing of the C source was the extraordinary amount of time it sometimes took to build the abstract syntax tree when deployed in Ericsson's Linux environment. This problem arose from two factors: that the size of the tree was bloated by extra (for our purposes) unnecessary included header files and that Ericsson uses a network file system which tended to be slow. This was not a problem in our local Windows development environment where the header files did not exist.

Since some parts of the state machine are usually located in the header file, it was initially considered a positive thing that all header files were automatically included and only one syntax tree produced. However, with the parsing taking upwards of one minute, the usability of the plugin would have been impaired. This was solved when we discovered the option of telling CDT's parser to skip including any headers - this then required the construction of a second syntax tree for the necessary header file and two subsequent traversals for the parser.

Another feature also uses built-in CDT functionality. Clicking on a name of a state or transition function in the state diagram opens its definition in code. This works regardless of whether the function definition is contained in the same file as the state machine definition. This is achieved using CDT's source file indexer.

The plugin can also parse files written in GraphML[1], a file format for graphs. The format is here used as an easy way to store input to the benchmarking procedure. It is based on XML, which made the parsing implementation easier by enabling the use of existing XML libraries.

3.2 User interface

For drawing graphs in the plugin we use *dot* to generate layout data - that is, x and y positions and spline control points - and then use this to draw the graph using Java components with the Draw2d framework[2]. Using Java components, as opposed to a static image, allowed more dynamic GUI functionality.

The main reason for using *dot* instead of our implementations of the Sugiyama framework, which are described in the coming sections, is that in order to make our implementations look good we would have to implement the additional step of smooth edge drawing using splines as well as clear edge labels using additional dummy vertices (see section 2.4). The former would have required a non-trivial amount of effort while the latter could have been attempted; none of them were a focus in this project, however, since they are not considered traditional parts of the Sugiyama framework. Also, *dot* already produces adequate layouts including splines and edge labels while being easy to interface to. Another reason is that

[1] GraphML homepage: http://graphml.graphdrawing.org/
[2] Draw2d homepage: http://www.eclipse.org/gef/draw2d/

the implemented algorithms in *dot* turned out to perform well in our benchmarks as seen in section 5.

The GUI supports selecting and highlighting vertices and edges which in turn triggers an information box that shows detailed information about the selected object. Also integrated is a search field which searches through all vertices and edges and selects those that match the search. These features can be seen in our screenshot of the program (figure 9).

3.3 Benchmarking

To ease the development of the layout implementations we constructed the plugin to be able to switch between normal mode (parsing Ericsson's state machines and using *dot* for layout) and "debug" mode (parsing either Ericsson's state machines or other graphs from GraphML, and using either *dot* or our layout implementations for layout). This way we could easily use the plugin to test our layout implementations on all graphs.

To produce the results as seen in section 5 we have written an automated benchmark program that iterates over test data in a folder and tests it with every possible layout combination while saving the results to an SQLite3 database. From this database we can then easily fetch the results we want.

4 Layout implementations

In the previous section we talked about the implementation of the plugin. Here we will describe the algorithms we have implemented for our study, grouped by the steps from the Sugiyama framework. All time complexity analyses here assume efficient data structures. Our own implementations may or may not actually adhere to this principle - this limitation is the reason that actual running time is not a main factor in our discussion of an algorithm's results.

4.1 Cycle removal

4.1.1 A simple heuristic

This simple algorithm first described by Berger and Shor [18] works by reversing, for each vertex, either the set of incoming ($E^+(v)$) or outgoing edges ($E^-(v)$) - whichever one is smaller. Edges and vertices that are visited are removed temporarily in the running of the algorithm but restored afterwards. The graph produced by this procedure is acyclic, proven by the following argument:

Suppose the graph has a cycle $(v_1, v_2, \dots v_n)$. Then there is a vertex v_i that was the first to be processed by the algorithm. But then either edge (v_{i-1}, v_i) or $(v_i, v_{i+1}$ would have been reversed and there can be no cycle. We have a contradiction and can conclude that the graph must be acyclic.

This algorithm guarantees that the set E_r of edges to be reversed is no more than half as large as the total number of edges E. The proof is by the following argument: all vertices are traversed and for each one, all its edges are either reversed and removed or just removed. In each step, at most as many edges are reversed as those that aren't.

Time complexity is $O(|V| + |E|)$. For each vertex, all its edges (except for those that have already been removed when visiting their other respective endpoints) are counted and then removed.

Algorithm 4.1.1 A simple heuristic by Berger and Shor [18].

```
1:  function REMOVE_CYCLES(G)
2:      E_r ← ∅
3:      for all v ∈ G do
4:          if |E+(v)| ≥ |E−(v)| then
5:              E_r ← E_r ∪ E−(v)
6:          else
7:              E_r ← E_r ∪ E+(v)
8:          end if
9:          delete E(v) from G;
10:     end for
11: end function
```

4.1.2 An enhanced greedy heuristic

Eades et al. [19] describe a new algorithm with an improved performance bound. In this solution, vertices are processed in a special order. It makes use of the observation that edges incident to sources and sinks (vertices with no incoming and outgoing edges, respectively) cannot be part of a cycle.

We can relate this algorithm (see pseudocode 4.1.2) with the previous one (algorithm 4.1.1) by noting that they are essentially the same; only this one processes the vertices in a special order, starting with sinks (vertices with no outgoing edges) and progressing to sources (with no incoming edges). To find this ordering while keeping the running time linear in the size of the graph one is slightly more dependant on an efficient implementation. Eades et al. advice using a bucket sort algorithm.

Because of its similarity to the previous algorithm, the same correctness argument applies here. For performance, however, it can be shown that the set of reversed edges is no bigger than $\frac{|E|}{2} - \frac{|V|}{6}$. The full proof of this can be seen in the original paper [19].

Algorithm 4.1.2 An enhanced greedy heuristic by Eades et al. [19]. Pseudocode based on Bastert & Matuszewski [20].

```
1: function REMOVE_CYCLES(G)
2:     E_r ← ∅
3:     while G not empty do
4:         while G contains a sink v do
5:             delete v and E⁻(v) from G
6:         end while
7:         delete from G all vertices v with |E(v)| = 0
8:         while G contains a source v do
9:             delete v and E⁺(v) from G
10:        end while
11:        if G is not empty then
12:            let v be a vertex in G with maximum value |E⁺(v)| − |E⁻(v)|
13:            add E⁻(v) to E_r and delete v and E(v) from G
14:        end if
15:    end while
16: end function
```

4.1.3 Depth-first search

The *dot* tool in Graphviz uses a depth-first search type algorithm (pseudocode shown in 4.1.3). The output of the algorithm is the set E_r of edges to be reversed.

The algorithm works by using the `on_stack` set to keep track of what vertices have already been encountered during the current iteration. When the same vertex is seen again, the edge to it is added to E_r and removed (temporarily during the running of the algorithm) from the graph. The `marked` set contains all vertices that have been visited over the complete run. Time complexity is $O(|V|+|E|)$, since all vertices and all edges are traversed in the worst case.

The algorithm is reasonably fast but does not give any performance guarantees by itself.

Adding another step could solve this, however: after the running of the algorithm, reverse either edges in E_r or its complement, i.e. edges that are in E (the set of all edges in the graph) but not in E_r - whichever is smaller. This would ensure that no more than half of the edges are reversed while still resulting in an acyclic graph. The implementation in *dot* does not use this step however, and so neither do we. As we shall see in section 5 this seems to not be an issue.

Algorithm 4.1.3 Cycle removal using depth-first search. Based on *dot* source code.

```
1: function REMOVE_CYCLES(G)
2:     E_r ← ∅
3:     marked = ∅
4:     on_stack = ∅
5:     for v ∈ G do
6:         depth_first_search(v)
7:     end for
8: end function
9: function DEPTH_FIRST_SEARCH(v)
10:    if v ∈ marked then
11:        return
12:    end if
13:    marked = marked ∪ {v}
14:    on_stack = on_stack ∪ {v}
15:    for (v, w) ∈ E⁺(v) do
16:        if w ∈ on_stack then
17:            E_r = E_r ∪ {(v,w)}
18:            Remove (v,w) from G
19:        else
20:            if w ∉ marked then
21:                depth_first_search(w)
22:            end if
23:        end if
24:    end for
25:    on_stack = on_stack \ {v}
26: end function
```

4.1.4 LP formulation

The problem of cycle removal can be formulated as an integer linear program in the following way:

$$\begin{aligned} \min \quad & \sum_{i \neq j} x_{ij} \\ \text{subject to} \quad & x_{i_1 i_2} + x_{i_2 i_3} + ... + x_{i_{k-1} i_k} \geq 1 \\ & \quad \text{if } (i_1, i_2, ..., i_k) \text{ is a cycle in } V, k \geq 2 \\ & 0 \leq x_{ij} \leq 1 \\ & x_{ij} \text{ integer} \end{aligned} \tag{3}$$

This and many other formulations are described in [21]. We chose the one above mainly

for its simplicity. There is one variable x_{ij} for each edge between vertices i and j, which is set to one if that edge is reversed.

The idea is simply to break all cycles by reversing one edge in each cycle. However, this requires that all cycles are known; this can be achieved by using Johnson's algorithm for finding cycles. For pseudo-code, we refer to Johnson's paper [22]. There he shows that, for a graph $G = (V, E)$ with c cycles, his algorithm has a running time bounded by $O((|V| + |E|)(c + 1))$.

4.2 Layer assignment

4.2.1 Longest path

Trademark: Layers vertex to layer i where i is the length of the longest path from a source

Longest path layering is a list scheduling algorithm which is frequently used in the Sugiyama framework since it produces drawings with the smallest possible height [2] and is simple to implement, for example as a variation of Dijkstra's shortest path algorithm with negative weights or as described by Tamassia et al. [2], seen in algorithm 4.2.1. It can be implemented with time complexity linear in the size of the graph.

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. The algorithm in 4.2.1 picks vertices whose incoming edges only come from past layers (Z) and assigns them to the current layer U. When there is no vertex left that fits that condition the layer is incremented and it starts over again until all vertices have been assigned a layer.

Algorithm 4.2.1 Longest path algorithm from Tamassia et al. [2].

Require: A DAG $G = (V, E)$
1: $U \leftarrow \emptyset$
2: $Z \leftarrow \emptyset$
3: $currentLayer \leftarrow 1$
4: **while** $U \neq V$ **do**
5: Select vertex $v \in V \backslash U$ with $V^+(v) \subseteq Z$
6: **if** v has been selected **then**
7: Assign v to the layer with number $currentLayer$
8: $U \leftarrow U \cup \{v\}$
9: **end if**
10: **if** no vertex has been selected **then**
11: $currentLayer \leftarrow currentLayer + 1$
12: $Z \leftarrow Z \cup U$
13: **end if**
14: **end while**

4.2.2 Coffman-Graham

Trademark: Layers with a given maximum width W, i.e. number of vertices per layer

The Coffman-Graham algorithm [23], like the longest path method, is a list scheduling algorithm that can also be used for layer assignment. It produces a layering with a maximum width of a given parameter W, which corresponds to the number of machines m in the job shop scheduling definition mentioned earlier (section 2.4.2).

As seen in algorithm 4.2.2, it first numbers all vertices in order of their number of incoming edges. After that it loops until all vertices have been assigned and in every iteration it picks a vertex such that its outgoing edges only go to unassigned vertices and which maximises the numbering from the first step. After a vertex has been picked we check if we should add it to the current layer or a new layer by checking against the parameter W and also make sure that any edges aren't connected with the current layer so as to avoid horizontal edges.

Algorithm 4.2.2 Coffman-Graham layering algorithm [2].

Require: A DAG $G = (V, E)$ without transitive edges and an integer $W > 0$
1: **for all** $v \in V$ **do**
2: $\lambda(v) \leftarrow \infty$
3: **end for**
4: **for** $i \leftarrow 1$ to $|V|$ **do**
5: Choose $v \in V$ with $\lambda(v) = \infty$ such that $|V^-(v)|$ is minimised
6: $\lambda(v) \leftarrow i$
7: **end for**
8: $k \leftarrow 1$, $L_1 \leftarrow \emptyset$, $U \leftarrow \emptyset$
9: **while** $U \neq V$ **do**
10: Choose $v \in V \backslash U$ such that $V^+(v) \subseteq U$ and $\lambda(v)$ is maximised
11: **if** $|L_k| \leq W$ and $V^+(v) \subseteq L_1 \cup L_2 \cup \cdots \cup L_{k-1}$ **then**
12: $L_k \leftarrow L_k \cup \{v\}$
13: **else**
14: $k \leftarrow k + 1; L_k \leftarrow \{v\}$
15: **end if**
16: $U \leftarrow U \cup \{v\}$
17: **end while**

In order for this to work properly a preprocessing step is required to transform the graph into its *transitive reduction*. What this means is that all edges (u, w) for which the edges (u, v) and (v, w) exist (with v distinct from u and w) are removed. For an example of a transitive reduction, see figure 10.

The worst-case time complexity of the algorithm itself is $O(|V|^2)$ [23]. However, when computing of the transitive reduction is accounted for we get a complexity of $O(|V|^3)$ [24].

For $W = 2$, the algorithm produces the optimal layering [23], i.e. the one with the fewest possible layers. Because of this we chose to use $W = 2$ when running our benchmarks. Additionally, in order to be better able to compare it with other algorithms in our benchmarks we decided to include a run using a value of $W = \sqrt{|V|}$. The motivation for this

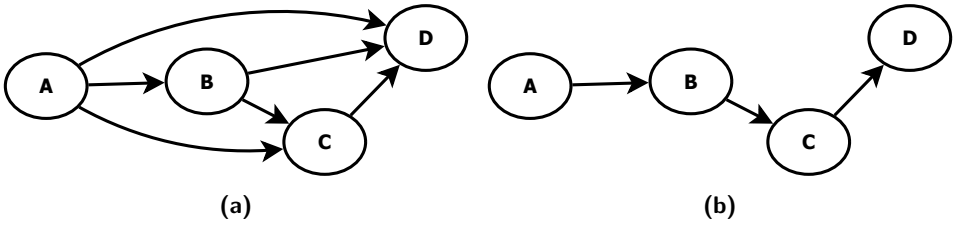


Figure 10: A graph and its transitive reduction.

is that it should also be able to produce layerings where the width is not dependent on a fixed constant but on the size of the graph itself.

4.2.3 LP formulation

Trademark: Minimise the total edge length.

The problem can be formulated as the following integer linear program [3,25] when trying to minimise the lengths of edges in the graph, where the length of an edge is the number of layers it stretches over:

$$\begin{aligned} \min \quad & \sum_{(v,w)\in E} \omega(v,w)(\lambda(w)-\lambda(v)) \\ \text{subject to} \quad & \lambda(w)-\lambda(v) \geq \delta(v,w), \forall (v,w) \in E \end{aligned} \tag{4}$$

Where $\lambda(v)$ is the layer vertex v is assigned to and $\omega(v,w)$ and $\delta(v,w)$ define a weight for and a minimum length of edge (v,w) respectively (these functions usually have the value 1).

An interesting note: the constraint matrix for this linear program is *totally unimodular* [17], which means that there always exists an integer solution if any solution exists at all. Because of this there is no need for this to be formulated as an ILP problem and the problem can be efficiently solved with the simplex method, commonly used for solving ordinary LP instances.

4.3 Vertex ordering

4.3.1 Barycenter and Median

Trademark: Layer-by-layer: Place vertex at barycenter (average) or median of connected vertices in last layer

The vertex ordering problem is usually solved by using the barycenter or the median function [3, 12]. Both these solutions sweep through the graph layer by layer such that the previous layer is already ordered and finished when the current layer's ordering is calculated. In these sweeps they calculate a value based on the edges between the previous layer and the current vertex which decides where in the layer the current vertex is to

be positioned. The barycenter function uses the average of the vertex positions of the connected vertices on the previous layer to position the current vertex while the median function uses the median of the same positions to do the positioning. As an example, a vertex with connected vertices in the previous layer at {1,2,5} gets a position of 4 with barycenter (average) and 2 with median. The sweeps are done three times, first top to bottom, then bottom to top and last top to bottom again. This is done to make sure that all vertices are considered in the ordering.

In the case that there exists an ordering within two layers that gives zero crossings both the median and barycenter will find it [12]. However, this guarantee is only valid when considering two layers (TLCM). Because the algorithms are sweeping through the graph layer-by-layer there is no guarantee that such an ordering is possible.

Gansner et al. [3] use the median function with mainly one improvement: they add an additional heuristic to reduce obvious crossings after the median function has ordered the layers. The heuristic iterates over adjacent vertex pairs in the same layer and swaps their positions if this improves (decreases) the number of crossings. By adding this simple heuristic, called transpose, the number of crossings can decrease even further. Also, they run the entire algorithms multiple times based on the previous result and then picks the best solution (one with least crossings). The following pseudocode in algorithms 4.3.1, 4.3.2 and 4.3.3 describe the median ordering used by Gansner et al. [3].

In our implementations and benchmarks we have four different but very similar algorithms:

Median with transpose is the algorithm described in pseudocode (4.3.1), that is, it uses the median and the transpose functions.

Barycenter with transpose uses the barycenter (average) instead of the median, and uses transpose.

Median uses the median but not the transpose function.

Barycenter uses the barycenter (average) but not the transpose function.

Algorithm 4.3.1 Main ordering function from Gansner et al. [3].

```
1:  function ORDERING
2:      order = init_order();
3:      best = order;
4:      for i = 0 → Max_iterations do
5:          wmedian(order,i);
6:          transpose(order);
7:          if crossing(order) > crossing(best) then
8:              best = order;
9:          end if
10:     end for
11:     return best;
12: end function
```

Algorithm 4.3.2 Median from Gansner et al. [3]. Barycenter works similarly but uses the average instead of the median.

```
1: function WMEDIAN(order,iter)
2:     if iter % 2 == 0 then
3:         for r = 1 → Max_layer do
4:             for v ∈ order[r] do
5:                 median[v] = median value of vertices in rank r-1 connected to v
6:             end for
7:             sort(order[r],median);
8:         end for
9:     else
10:         ...
11:    end if
12: end function
```

Algorithm 4.3.3 Transpose from Gansner et al. [3]. Loops through all neighbouring vertices and tests if transposing them decreases the number of crossings.

```
1: function TRANSPOSE(layer)
2:     improved = True;
3:     while improved do
4:         improved = False;
5:         for r=0 → Max_layer do
6:             for r = 0 → |layer[r]|-2 do
7:                 v = layer[r][i];
8:                 w = layer[r][i+1];
9:                 if crossing(v,w) > crossing(w,v) then
10:                    improved = True;
11:                    exchange(v,w);
12:                end if
13:            end for
14:        end for
15:    end while
16: end function
```

4.3.2 LP formulation

Trademark: Minimise the number of crossings using pair-wise comparisons.

When formulating the problem we need a way to model the ordering of the vertices within layers. To do this we use the notation x_{ij} which tells us which order i and j is in: x_{ij} is 1 if vertex i lies before j and 0 otherwise.

The following equation (5) is a simplification which we have derived from Jünger et al. [26].

$$\begin{aligned} \min \quad & \sum_{(i,j),(k,l)\in E} c_{ijkl} \\ \text{subject to} \quad & x_{ik} + x_{lj} - c_{ijkl} \leq 1, (i,j),(k,l) \in E \\ & x_{ki} + x_{jl} - c_{ijkl} \leq 1, (i,j),(k,l) \in E \\ & 0 \leq x_{ij} + x_{jk} - x_{ik} \leq 1, 0 \leq i < j < k < n \\ & 1 \leq x_{ab} + x_{ba} \leq 1 \\ & c_{ijkl}, x_{ij} \text{ integer} \end{aligned} \tag{5}$$

As seen in equation 5 we want to minimise crossings between edges (i, j) and (k, l)(first row), and by comparing all edge pairs coming from the same layer we can easily come up with the two possible crossing scenarios as seen in figure 11. These two types of crossings are modelled by the constraints in rows two and three respectively. We also need a global rule, that of transitivity seen in the fourth row, to hold the smaller subproblems together. The fifth row is necessary to keep x_{ab} and x_{ba} as inverses of each other.

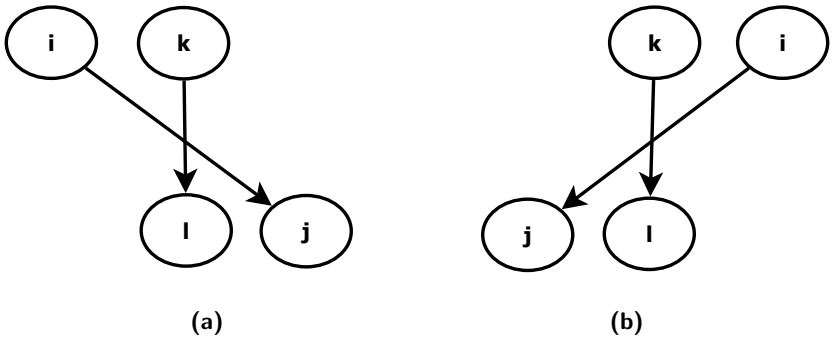


Figure 11: The two types of crossings.

Unfortunately, this formulation showed to be very slow to compute when the number of vertices and edges grew. For some combinations on the bigger graphs in our benchmark it could take up to three days to complete, making us have to skip it in our data section for a few of our test graphs. This could be because the formulation is not properly optimised for an LP solver. However, that kind of optimisation was outside of the scope of this thesis and was therefore not investigated further here.

4.4 Coordinate assignment

4.4.1 Priority method

Trademark: Layer-by-layer: In a prioritised order, place each vertex at the barycenter of its neighbours.

The priority method [27] is very similar to the barycenter ordering solution since it, for each vertex v, tries to place v based on its neighbours' average positions. The difference between them is that the priority method assigns every vertex a priority based on two criteria:

Total degree of edges connected to the previous layer. A high degree means the vertex is highly connected and should be prioritised over less connected vertices.

Dummy vertex means that the vertex is a part of an edge. By giving dummy vertices the highest priority we are also prioritising straightening edges.

A vertex with high priority can override and move lower priority vertices as needed, making sure that those highly connected vertices or long edges get placed where they should. The reasoning behind this is that it "costs" more to move a highly connected vertex to the wrong place than it does for a less connected one.

Algorithm 4.4.1 Priority method.

1: initialise vertex positions
2: **for** each layer L_i in: $L_1 \to L_n$, $L_{n-1} \to L_0$ and $L_1 \to L_n$ **do**
3: calculate priority for all $v \in L_i$ based on connections to the previous L
4: **for** $v \in L_i$ in order of highest priority **do**
5: place v as close as possible to the barycenter of connected vertices in previous L while, if needed, moving only vertices of lower priority
6: **end for**
7: **end for**

4.4.2 LP formulation

Trademark: Minimise the difference in x-position between connected vertices maintaining separation constraints.

The LP formulation seen in equation 6 comes from Gansner et al. [3] and describes how to minimise the difference in x-position between all pairs of vertices that share an edge while favoring to straighten long edges.

$$\begin{aligned} \min \quad & \sum_{(v,w)\in E} \Omega(e)\omega(e)|x_w - x_v| \\ \text{subject to} \quad & x_b - x_a \geq \rho(a,b) \\ & \rho(a,b) = \frac{xsize(a) + xsize(b)}{2} + vertexsep \end{aligned} \tag{6}$$

$\omega(e)$ is the weight of the edge; in our case we don't have any weights so it is always equal to 1. $\Omega(e)$ is more interesting, however; it is used to make the solution favor straightening long edges over shorter ones. It can take three different values depending on the endpoints of e being dummy vertices. e can have both endpoints as dummy vertices, only one endpoint as a dummy vertex or none of them. In order to favor straightening long edges we then get $\Omega(none) < \Omega(one) < \Omega(both)$. In accordance with Gansner et al. we use actual values 1, 4 and 8. $xsize(v)$ is the width of the vertex v, often depending on the size of a text label associated with it. *vertexsep* is the minimum separation between vertices in the drawing.

5 Evaluation of implementations

The following results come primarily from benchmarks of graphs that represent nine of Ericsson's state machines, but we will also backup our claims with benchmarks of a larger test set of DAGs called the North DAGs[1]. Around 200 of these are used, chosen to be comparable in size to the Ericsson graphs, providing us with more statistical certainty for our claims. The criteria we will investigate are some of those mentioned earlier (see section 2.3) plus some additional criteria applicable to the specific subproblems in order to evaluate them against each other. After having evaluated the algorithms within their subproblems we will continue by looking at the relations between algorithms in different subproblems.

In order to evaluate cycle removal algorithms against the larger data set of the North DAGs (directed *acyclic* graphs), we have actually modified them by randomly reversing edges to create cycles. Henceforth, whenever we mention the North DAGs we are referring to this modified set.

The amount of results received from running these nine graphs and 200 of the North DAGs is quite massive. This arises from the large amount of combinations of the different algorithms described. We have a total of $4 * 4 * 5 * 2 = 160$ different combinations to test for each graph; four algorithms for cycle removal, four for layer assignment (two runs with Coffman-Graham with different W parameter), five for vertex ordering and two for coordinate assignment. Because of this several limitations are necessary in order to be able to analyse this data.

When it comes to the North DAGs we have to normalise results in order to compare between runs on the many different graphs and to produce an average between them. A graph with a lot of edges, for example, might have a lot of crossings where a graph with only a few edges might not have as many. This particular case can be normalised by dividing the number of crossings with the number of edges in the graph and thus make it possible to compare an algorithm's results on different graphs with each other. However, these normalisations are not 100% correct since there might not be a linear relation between for example crossings and the number of edges; it might only be correlated and this will limit the accuracy of our results.

The main diagrams we use for presenting our data are histograms where there is one set of stacks for each file and for each file there is one stack for the pure LP combination and then one for each algorithm (see table 1) in the current subproblem. Also, each algorithm stack is divided into three parts: the lower part is the average of the lowest 10% in the data, the middle part is the average of all the data and the upper part is the average of the upper 10%. The reasoning behind having a separate stack for the pure LP combination is to have somewhat of a baseline for comparison and, possibly, to see whether the different steps can interfere with each other to make a pure LP solution perform worse than one including heuristics.

We also refer to tables in the appendix which represent test data for both graph sets. These are the normalised values: for example, the average number of crossings achieved for a graph by a specific algorithm, over all the combinations the algorithm appears in, is divided by the number of edges in the graph. The average of this value over all the

[1]Download page for North DAGs (directed acyclic graphs) defined in GraphML: http://www.graphdrawing.org/data/

Subproblem	Full name	Abbreviation
All	Pure LP combination	X
Cycle removal	Simple heuristic	S
	Greedy heuristic	G
	Depth-first search	D
	Linear programming	LP
Layer assignment	Longest path	L
	Coffman-Graham $W = 2$	C2
	Coffman-Graham $W = \sqrt{\|V\|}$	CS
	Linear programming	LP
Vertex ordering	Barycenter	B
	Barycenter with transpose	BT
	Median	M
	Median with transpose	MT
	Linear programming	LP
Coordinate assignment	Priority method	P
	Linear programming	LP

Table 1: Shortened names of algorithms.

graphs is then taken. To get a value centered around 1 we then also divide by the average achieved over all algorithms. This makes it easier to see how the different algorithms compare against each other.

However, by looking at the tables in the appendix we can see that there is no noticeable difference in performance of algorithms between Ericsson's graphs and the North DAGs. For this reason we will focus our discussion on the Ericsson graphs.

5.1 Cycle removal

All cycle removal algorithms solve the problem correctly, i.e. they all produce acyclic graphs. The measurement used here is how many edges the algorithm has reversed. The reversed edges are the ones that point "against the flow" in the final drawing; this number is therefore useful to minimise in order to properly show the hierarchical nature the graph.

File	$\|V\|$	$\|E\|$	Cycles	Self-edges
F1	7	13	2	0
F2	7	27	25	3
F3	10	33	16	2
F4	14	24	0	0
F5	15	38	17	8
F6	17	29	13	0
F7	22	41	1	0
F8	22	53	14	0
F9	22	58	21	1

Table 2: Number of cycles for the files.

For reference, we give the number of cycles in each graph in table 2. These impose an

upper bound on the number of reversed edges in the optimal solution. This bound is weak since an edge may participate in many cycles. We also give the number of self-edges, i.e. cycles involving only one vertex. These are included in the number of cycles but are handled separately by the implementation; they are trivial to find and are not taken into account by our layout algorithms, so they can safely be removed during the computation.

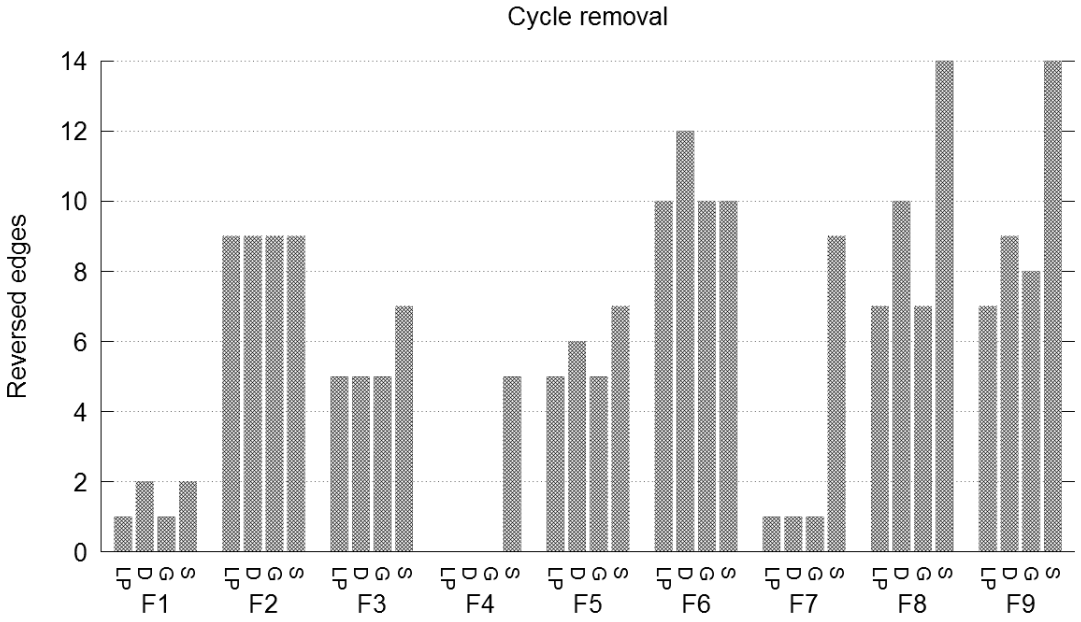


Figure 12: Number of reversed edges for the Ericsson graph set.

From figure 12, we can see one clear outlier in the simple heuristic. Notably, it reverses five edges in a graph (F4) which our table (2) shows us is already acyclic and which the other algorithms manage to leave unmolested. From the definition we can expect this - the only kind of graph for which no edges would be removed is one with exactly two vertices and edges from one to the other. The only apparent strengths of this heuristic are its linear time complexity and its simplicity. This result suggests a possible addition of a procedure to check if the graph is already acyclic - this could be implemented linearly and easily using a depth-first search (basically similar to the depth-first search algorithm used here with the exception of stopping execution as soon as a cycle is found).

Another that does not perform as well is the depth-first search algorithm. It is the worst of the lot when looking at graph F6. Overall, the difference is not that great, though. The enhanced greedy heuristic, however, excels. It is close to matching the (optimal) value of the LP formulation. This, coupled with its good theoretical bounds on performance and running time, makes it our winner in this category when looking at the heuristics.

We must also stress that the particular solution generated by a cycle removal algorithm can greatly affect the later stages in the layout process. Though two solutions may reverse the same number of cycles, thus performing equally well according to the measure used here, the particular edges that are reversed may differ. How this, in turn, affects the quality of the final drawing is hard to predict, however.

5.2 Layer assignment

We measure the performance of layer assignment algorithms by the size of the layering and the number of dummy vertices created. The more dummy vertices a graph has, the bigger and more complex the graph becomes for subsequent steps.

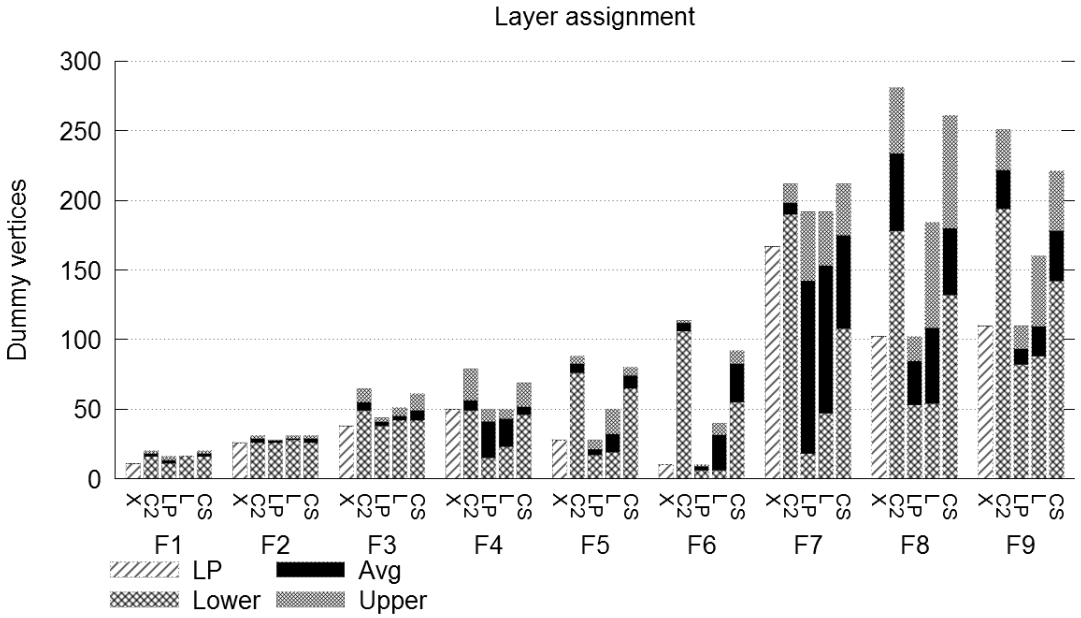


Figure 13: Number of dummy vertices for the Ericsson graph set.

We take a look at the dummy vertices in figure 5.2. Remember that the amount of these is preferred small since less dummy vertices means shorter edges which results in a graph that is easier to follow. Also, the total number of vertices (dummy and otherwise) impacts the size of the input of subsequent steps in the framework and thus the running time. We see that in some cases the number is very large - graphs F7, F8 and F9 all reach over 200 in some runs, to be compared with the number of actual vertices which is only 22. This is because the layering algorithms have placed some connected vertices at very different layers and thus created very long edges containing dummy vertices.

We see that the LP solution performs best, which is expected since it is actually dummy vertices that it minimises (minimising edge length is the same as minimising dummy vertices). The longest path algorithm takes second place and this is also a bit expected since as we mentioned earlier it produces a layering with the minimum makespan, the layer height, which in turn is related to the number of dummy vertices since a shorter graph equals shorter edges and thus fewer dummy vertices.

We can clearly see also that Coffman-Graham's algorithm with $W = 2$ performs horribly; this is to be expected with the restriction of two vertices per layer. As mentioned in section 4.2.2, the layering thus produced is optimal for that case, but this is not of great practical value since a maximum width of two is often too small even with our small graphs. The variant of the algorithm with $W = \sqrt{|V|}$ overall performs better, though generally not as good as the other algorithms. The algorithm's strength, which lies in the ability to specify the maximum width W, was not considered an important feature in our application and therefore we consider it a loser according to our tests.

Notably, on file F7 in the diagram, the pure LP combination (labeled X in the diagram) performs much worse than the best competing combination. The simple cycle removal algorithm here allowed for a much better value for this metric.

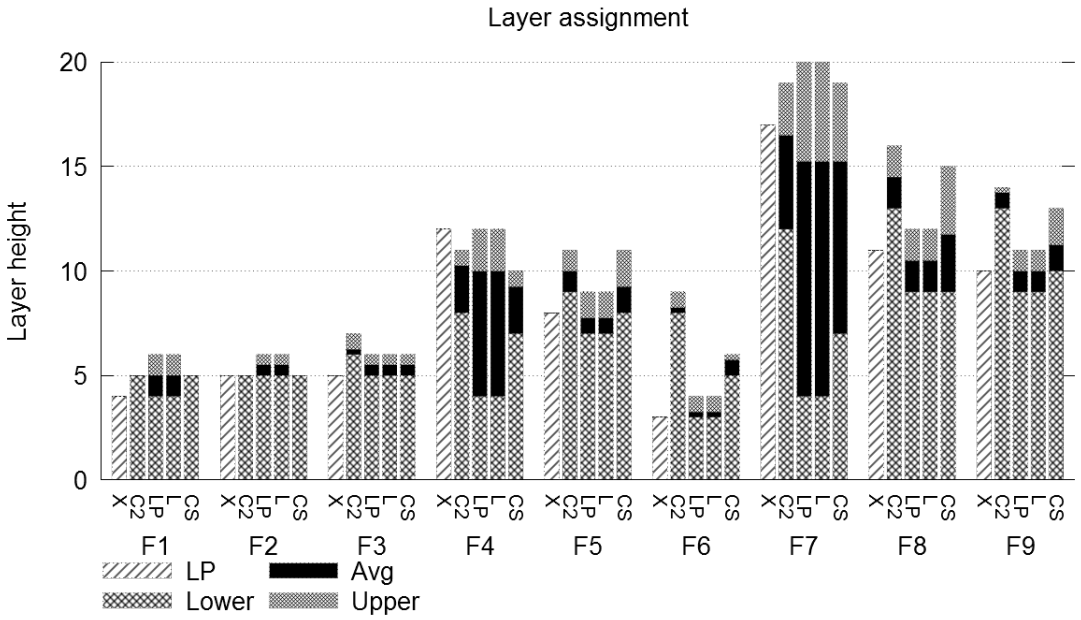


Figure 14: Layer height (number of layers) for the Ericsson graph set.

Figure 14 is interesting mainly to show the similarity between the LP solution and the longest path heuristic. The latter guarantees the lowest height possible, which in itself can be very valuable and makes for a more condensed graph with several vertices in the same layer. This does not in general imply the lowest total dummy vertices, however. The two are obviously correlated and seem to be so at a high degree with these test graphs. Looking at the averaged results from the North DAG tests in table 5 however, we see a small difference.

5.3 Vertex ordering

The sole purpose of vertex ordering is to minimise edge crossings and therefore that is our only measurement. It might be considered one of the most important measurements for graph drawing. In order to permit some degree of comparison of an algorithm's performance in this regard on different graphs, we have calculated the crossing number of each graph involved. This number shows the minimum amount of crossings possible. However, one should note that it is not always possible for an ordering algorithm to meet this number since it depends on the layer assignment and cycle removal steps.

When benchmarking the number of crossings we could unfortunately not run all combinations with the LP formulation on graphs F8 and F9 since they were apparently too big or complex for our formulation to compute in a reasonable time; this is further discussed in section 4.3.2. A single combination could take over two days and considering that we would need to run it with all combinations we simply did not have the time to run them all through; we made an exception for the pure LP combination. The rest of the graphs did not pose this problem.

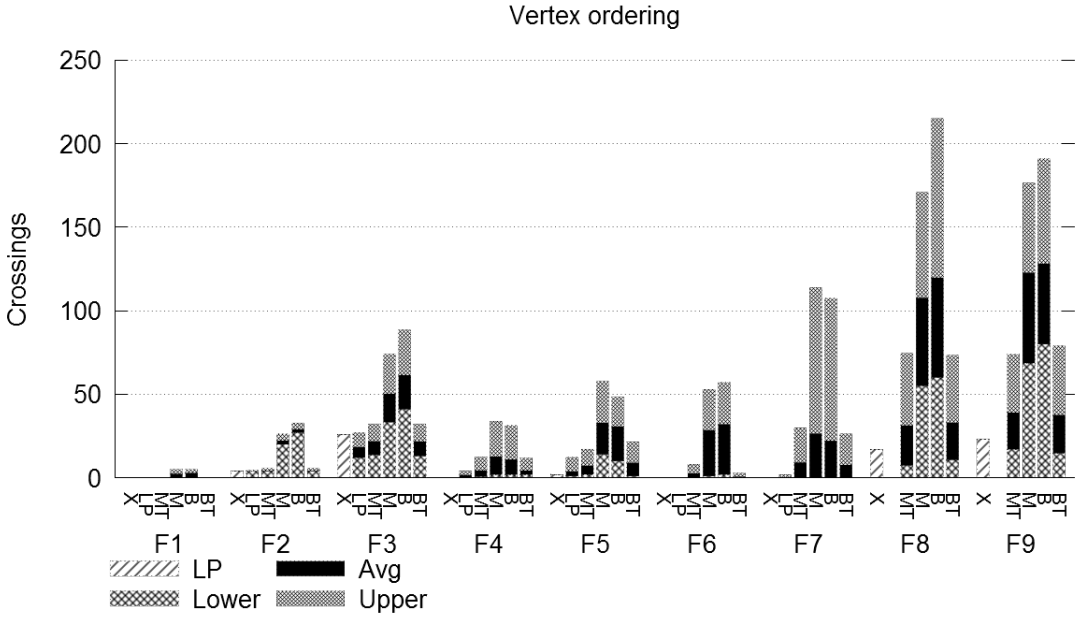


Figure 15: Number of crossings for the Ericsson graph set.

As you can see in figure 15 the transpose method (used in MT and BT) introduced by Gansner et al. [3] is a very effective addition and also quite close to the optimal solution offered by LP on some of the graphs. However, it is nowhere near optimal in the case of F9 as seen in table 3 where F9 has a crossing number of 5 while the best achieved by us is 14. The barycenter and median methods perform very similarly both with and without transpose. It is also very interesting to note how big of a difference there can be in the number of crossings depending on which algorithms are used, like in F8 where the minimum is 7 and the maximum is well above 200.

File	$\|V\|$	$\|E\|$	Crossing number	Best achieved					
				X	LP	MT	BT	M	B
F1	7	13	0	0	0	0	0	0	0
F2	7	27	4	4	4	4	4	20	27
F3	10	33	11	26	12	13	12	33	40
F4	14	24	0	0	0	0	2	2	2
F5	15	38	0	2	1	2	1	14	10
F6	17	29	0	0	0	0	0	1	2
F7	22	41	0	0	0	0	0	0	0
F8	22	53	3	17	-	7	9	55	60
F9	22	58	5	23	-	17	14	68	77

Table 3: Crossing number for the files.

In table 3 we can see the crossing number for each file and the best achieved results for each algorithm. One must first note that the crossing number is not always possible to achieve in our case since it also depends on the previous steps, cycle removal and layer assignment. However, we do come quite close on most of the files. We can easily see that the pure LP solution never performs better than the best heuristic. This is because its previous steps, although optimal in their own formulations, does not look ahead to

consider crossings.

5.4 Coordinate assignment

Coordinate assignment aims to place vertices as close to its connected vertices as possible while also trying to straighten long edges as much as possible. Because of this we focus our measurements on the bends of edges, both the number of them and their angles. It is also interesting to look at the area of the final drawing.

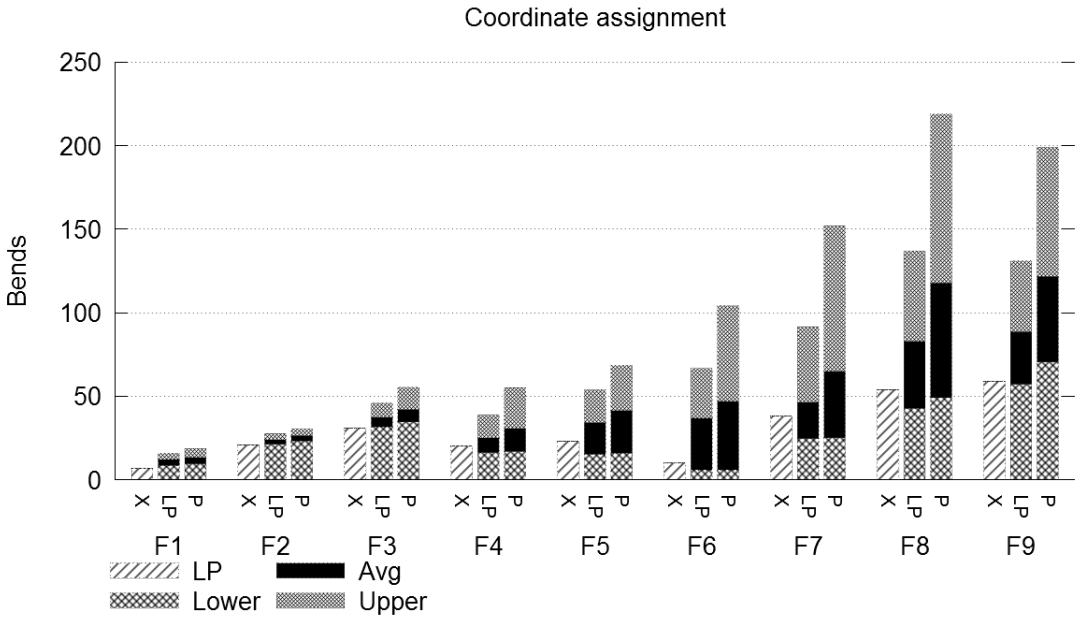


Figure 16: Number of bends for the Ericsson graph set.

To count bends we have iterated over all dummy vertices and compared the x-coordinates of their connected vertices. If they are not equal in a certain small interval ϵ then the dummy vertex is counted as a bend. A high number of bends may indicate that the graph has many jagged edges and thus could be hard to read. However, by not knowing the angles of the bends we cannot draw any clear conclusions. Interestingly, the LP solution, both when run on the Ericsson's and North DAGs' graphs, always gets the smallest amount of bends (see figure 16 and table 7) though it seeks only to minimise differences in vertex x-coordinates (admittedly, the two are correlated).

The average for edge angles is in some ways a better measurement than the amount of bends since it takes all bends into account and also gives us the average angle, which can tell us more about how jagged the majority of the edges are. One should take note here that this average not only considers angles of bends but also those of dummy vertices that are not bent (near straight edges) and therefore the average might be lower than the real average of the bend angles. The same reasoning about LP for the number of bends fits here also; the LP solutions tries to minimise the difference between connected vertices, especially dummy vertices, and in the process seems to achieve a good number also for average edge angles.

As you can see in figure 18 the priority method tends to have a quite big area. When looking at the raw data of height and width we can clearly see that it is the width that

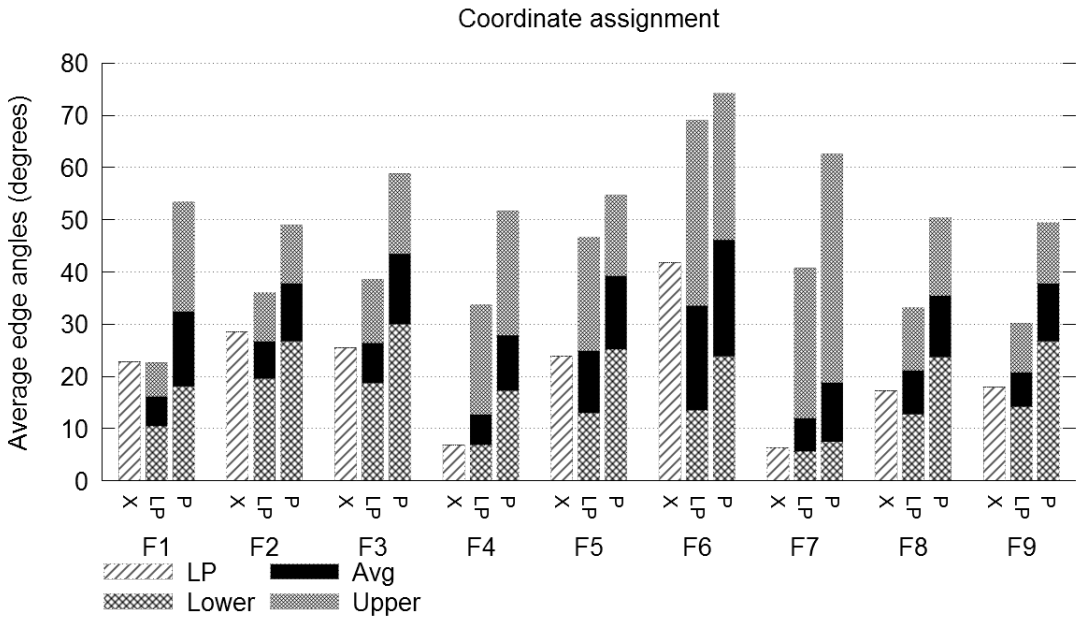


Figure 17: Average angle of edges for the Ericsson graph set.

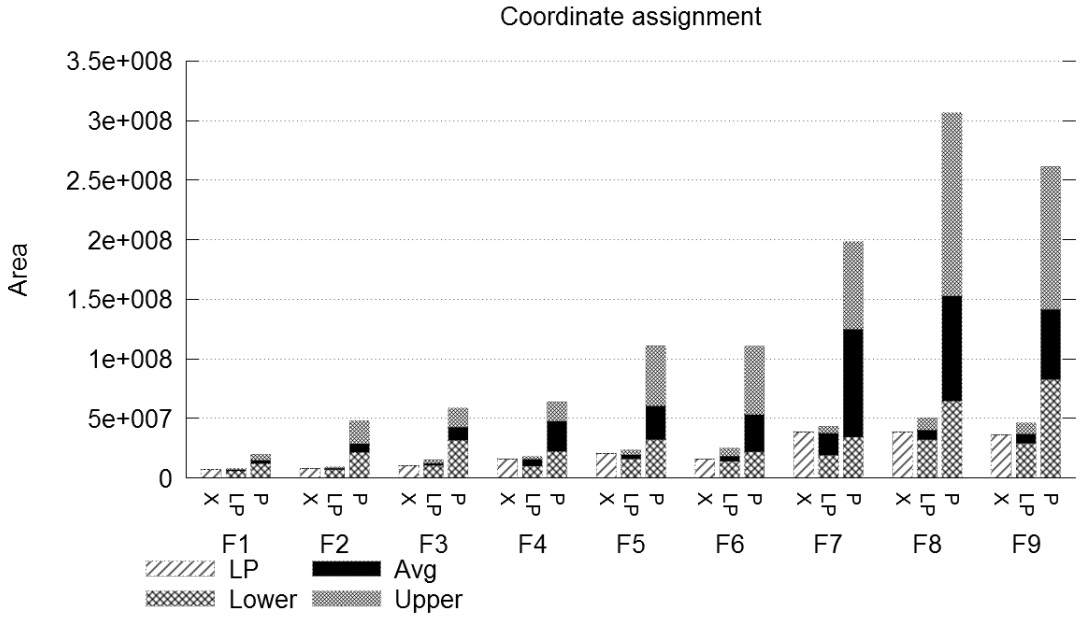


Figure 18: Area of final drawing for the Ericsson graph set.

becomes too large. Actually, the height is the same for the two algorithms since it is trivially computed beforehand. The reason for the priority methods widening is its layer-by-layer concept. After computing a layer it becomes locked for the next layer (we only compute coordinates for the current layer in each step) which means that the current layer can be skewed to the side because of the inability to edit the previous layer. If this happens to several layers, which seems to happen frequently, then we get a lot of skewing and thus a wide drawing.

We can easily see that the LP solution wins for all criteria in this subproblem, but it is important to note that if the algorithms had performed more evenly or won half of the criteria each then we would have no way to say which performed best overall since we do not know how to prioritise these criteria.

5.5 Relations between subproblems

In order to investigate how different steps in the framework affect each other, we have plotted two of the most interesting criteria against specific algorithms from previous steps. These criteria are the number of crossings and the number of dummy vertices.

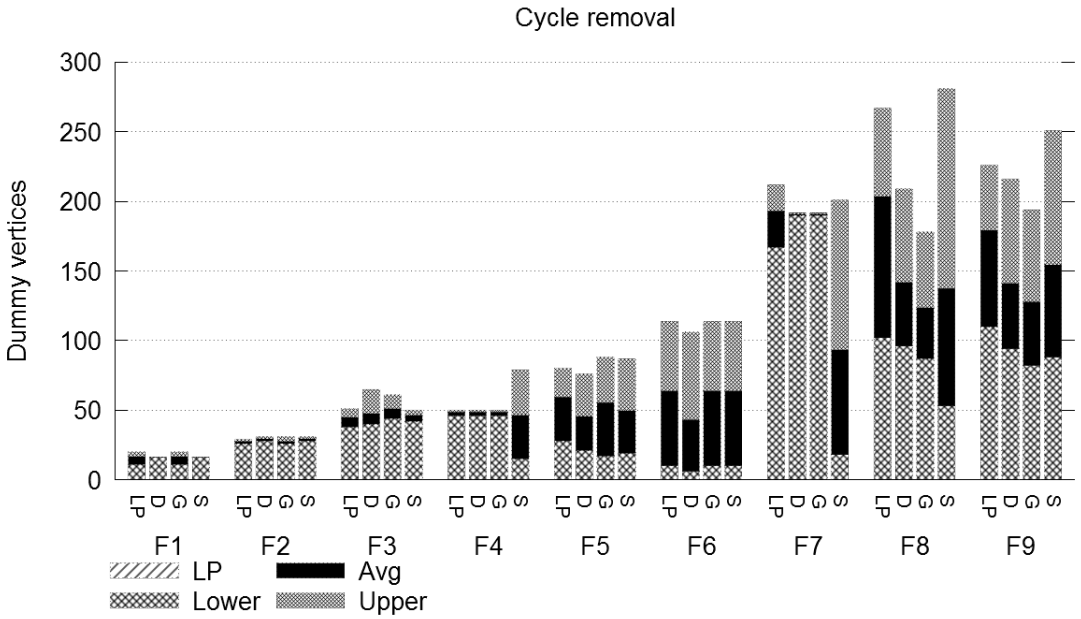


Figure 19: Number of dummy vertices for the Ericsson graph set, in relation to cycle removal algorithms.

When looking at how dummy vertices are affected by the cycle removal algorithm (see figure 19) we get some mildly astonishing results. For some reason the simple heuristic outperforms all the others (F4, F7 and F8) in some very lucky combinations. These are the simple heuristic coupled with either the LP solution or the longest path solution, where LP was a little bit better. This is very interesting results since earlier (section 5.1) the results pointed towards the simple heuristic being quite bad. But as it seems it can in some cases actually create better solutions for subsequent steps than the optimal LP solution for cycle removal when it comes to dummy vertices.

Figure 20 shows the number of crossings for each cycle removal algorithm. Here the depth-first search heuristic performs consistently better than the others, though not by a large margin.

Taking crossings against the layer assignment algorithms in figure 21, we see that the LP solution outmatches the others in some cases with longest path coming in a quite close second. Many of the graphs show very similar results, however.

From all these diagrams we can see hints that there are relations between the different

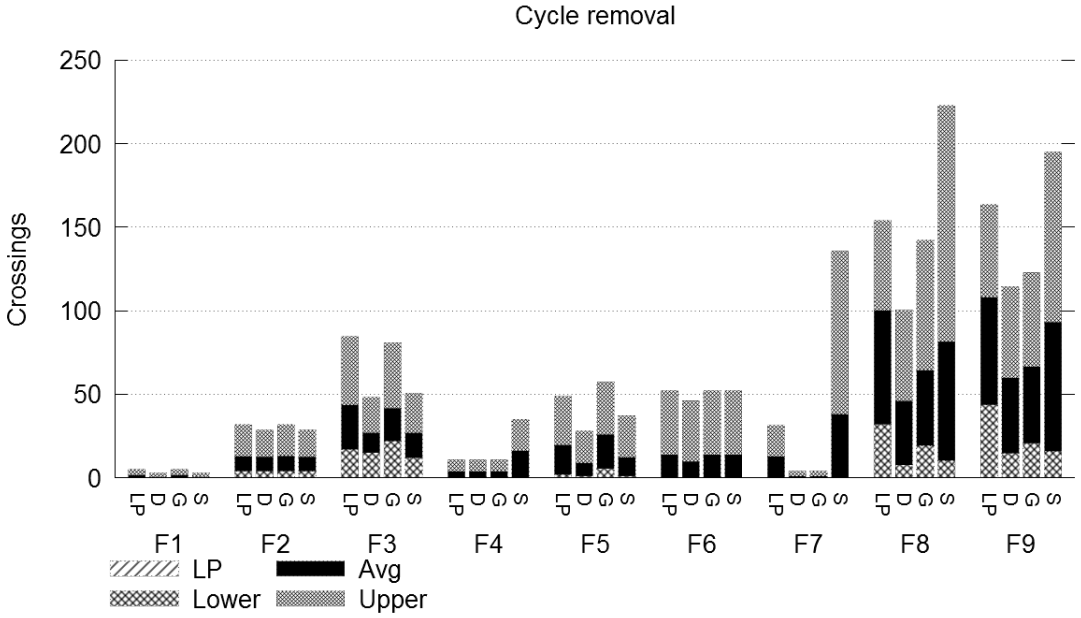


Figure 20: Number of crossings for the Ericsson graph set, in relation to cycle removal algorithms.

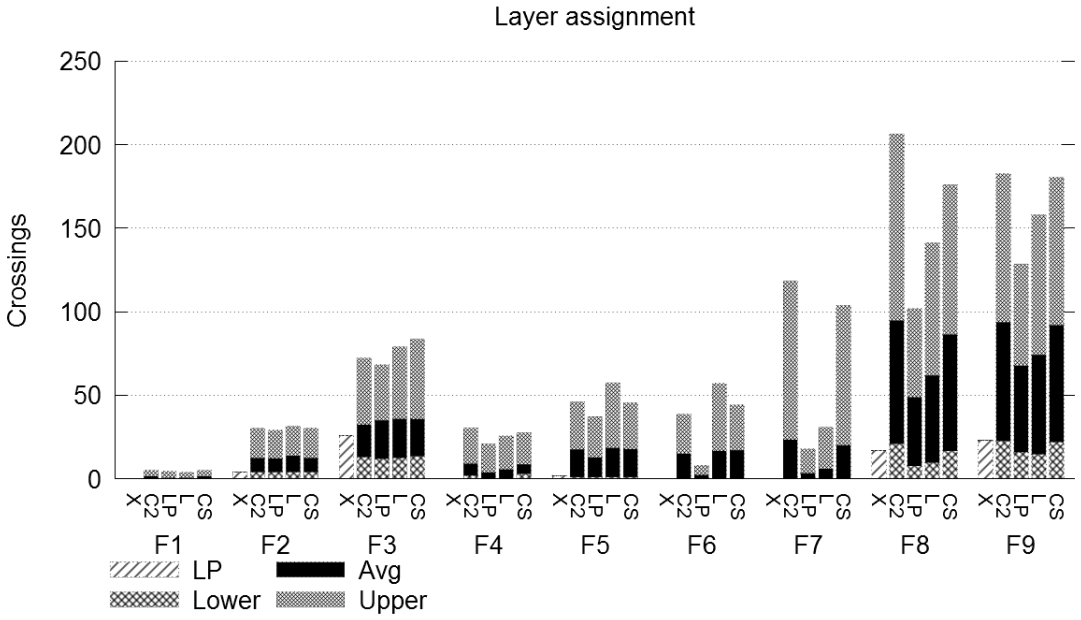


Figure 21: Number of crossings for the Ericsson graph set, in relation to layer assignment algorithms.

steps of the Sugiyama framework that are not caught in our criteria. Perhaps there is some yet unknown criteria or property, whether of the graphs or the algorithms, that can tell us how and why these steps interfere with each other. In our investigation we have not seen any research covering this question.

Unfortunately, we can't pick any winners from these results. In order to pick a *best* combination one would have to assign weights to these criteria so as to be able to consider

all of them. We have not come up with any good way of assigning these weights and we have not seen any research on this problem. However, we can see that the algorithms that *dot* use, that is, depth-first search for cycle removal, LP for layer assignment, median with transpose for ordering, seems to performs well and thus is adequate to use for the plugin that was developed. For coordinate assignment however *dot* uses its own quite complex heuristic which is not investigated here.

6 Conclusion

A tool for visualising state machines was developed for use by Ericsson software engineers. It needed to create good layouts of state diagrams; this prompted the investigation of existing graph layout methods. The Sugiyama framework was chosen, and because of the many variations possible within it, a more thorough investigation was done in order to possibly find one that was particularly suited to the graphs as defined by Ericsson state machines.

The four steps of the Sugiyama framework were implemented with some popular approaches from the field of graph layout research. These approaches were then benchmarked and compared against each other to find a suitable solution to the problem of visualising state machines. A major focus was to look at how different steps interfere with each other to impact the final drawing, making it necessary to not only look at the individual steps but also at the combinations of all steps.

For cycle removal algorithms, we saw that heuristics can easily compete against and quite nearly match an exact solution, while being considerably simpler to implement and having linear time complexity.

For layer assignment, an exact solution using LP is feasible since the problem is efficiently solvable using the simplex method. Coffman-Graham's algorithm did not perform well in our tests and is unsuited for our purposes. A longest path heuristic however fares reasonably well.

For vertex ordering, however, an LP solution is not reasonable, taking days to complete in some cases. Looking at the barycenter and median heuristics there is no clear winner, but adding the transpose heuristic to either is greatly beneficial.

For coordinate assignment, an LP solution is feasible for our purposes, being efficiently solvable similarly to layer assignment. It is preferred above the priority method, whose performance in our tests was comparatively bad.

Finally, we showed how heuristics can perform better than an optimal solution when looking at the subsequent steps. This opens up to a new approach to solving the Sugiyama framework's subproblems; looking at combinations of solutions might be more efficient than simply picking the best for each step. Also, this might mean that there is some other kind of property or criteria of the graph which has not been studied here that would highlight why a specific algorithm makes subsequent algorithms perform better.

These results did not show that there existed a combination of algorithms that clearly outperformed the implementation in the *dot* tool. This, coupled with the facts that a reliance on additional software turned out to not be an issue for the deployment of our plugin and that *dot* contained some functionality that, due to time constraints, was not implemented by us, motivated the use of *dot* in the final product.

6.1 Future work

As mentioned in section 5.5 we suspect there might be some yet unknown property or criteria of the graph that can tell how a solution to one subproblem effects another. We suggest that this is studied further since finding a property like that might make it possible to create much better layout algorithms.

It would also be possible to extend the investigation by implementing more algorithms and adding them to our test framework. For example, adding a network simplex solution to the layer assignment equation (as used by Gansner et al. [3]) or a fixed-parameter algorithm for cycle removal as shown in Chen et al. [9].

When it comes to layer assignment there is a popular addition called Promote Layering [28] that much like the transpose method for ordering augments an existing algorithm. This would be interesting to test in combination with our implemented layering methods to see if they can be improved.

The vertex ordering heuristics tested here are all quite similar and it would be very interesting to test some heuristics that differ more.

The coordinate assignment step has not been studied very deeply in this thesis and therefore it would be interesting to test some more approaches to it.

The results of this thesis can be used to choose a combination of algorithms for use by Ericsson to remove the *dot* dependency. There is some remaining work to be done before the drawing algorithm can be considered complete, however. Though these issues have not been thoroughly described in the paper we list them here briefly:

- Better edge drawing, e.g. using splines.
- Better handling of edge labels.
- Drawing self-edges.
- General optimisation of implemented algorithms.

References

[1] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 11, pp. 109 –125, feb. 1981.

[2] R. Tamassia, *Handbook of Graph Drawing and Visualization*. Discrete Mathematics And Its Applications, Taylor and Francis, 2010.

[3] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo, "A technique for drawing directed graphs," *Software Engineering, IEEE Transactions on*, vol. 19, pp. 214 –230, mar 1993.

[4] W. Huang, S.-H. Hong, and P. Eades, "Effects of crossing angles," in *Visualization Symposium, 2008. PacificVIS '08. IEEE Pacific*, pp. 41 –46, march 2008.

[5] L. Nachmanson, G. G. Robertson, and B. Lee, "Drawing graphs with glee," in *Graph Drawing, 15th International Symposium, GD 2007, Sydney, Australia, September 24-26, 2007. Revised Papers* (S.-H. Hong, T. Nishizeki, and W. Quan, eds.), vol. 4875 of *Lecture Notes in Computer Science*, pp. 389–394, Springer, 2007.

[6] D. Auber, "Tulip — a huge graph visualization framework," in *Graph Drawing Software* (M. Jünger and P. Mutzel, eds.), Mathematics and Visualization, pp. 105–126, Springer Berlin Heidelberg.

[7] C. Bachmaier, "A radial adaptation of the sugiyama framework for visualizing hierarchical information," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, pp. 583–594, May 2007.

[8] R. M. Karp, "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008* (M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, eds.), pp. 219–241, Springer Berlin Heidelberg.

[9] J. Chen, Y. Liu, S. Lu, B. O'sullivan, and I. Razgon, "A fixed-parameter algorithm for the directed feedback vertex set problem," *J. ACM*, vol. 55, pp. 21:1–21:19, Nov. 2008.

[10] J. Pach and G. Tóth, "Graphs drawn with few crossings per edge," *Combinatorica*, vol. 17, pp. 427–439.

[11] M. R. Garey and D. S. Johnson, "Crossing number is np-complete," *Siam Journal On Algebraic And Discrete Methods*, vol. 4, no. 3, pp. 312–316, 1983.

[12] P. Eades and N. C. Wormald, "Edge crossings in drawings of bipartite graphs," *Algorithmica*, vol. 11, pp. 379–403.

[13] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Inf. Process. Lett.*, vol. 31, pp. 7–15, Apr. 1989.

[14] P. A. Eades, "A heuristic for graph drawing.," in *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[15] R. Adams, *Calculus: A Complete Course*. Pearson Addison Wesley, 2006.

[16] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Softw. Pract. Exper.*, vol. 21, pp. 1129–1164, Nov. 1991.

[17] J. Matoušek and B. Gärtner, *Understanding and Using Linear Programming (Universitext)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[18] B. Berger and P. W. Shor, "Approximation algorithms for the maximum acyclic subgraph problem," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, (Philadelphia, PA, USA), pp. 236–243, Society for Industrial and Applied Mathematics, 1990.

[19] P. Eades, X. Lin, and W. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Information Processing Letters*, vol. 47, no. 6, pp. 319 – 323, 1993.

[20] O. Bastert and C. Matuszewski, "Layered drawings of digraphs," in *Drawing Graphs* (M. Kaufmann and D. Wagner, eds.), vol. 2025 of *Lecture Notes in Computer Science*, pp. 87–120, Springer Berlin / Heidelberg.

[21] R. Kaas, "A branch and bound algorithm for the acyclic subgraph problem," *European Journal of Operational Research*, vol. 8, no. 4, pp. 355 – 362, 1981.

[22] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.

[23] E. G. C. Jr. and R. L. Graham, "Optimal scheduling for two-processor systems.," *Acta Inf.*, vol. 1, pp. 200–213, 1972.

[24] J. La Poutré and J. van Leeuwen, "Maintenance of transitive closures and transitive reductions of graphs," in *Graph-Theoretic Concepts in Computer Science* (H. Göttler and H.-J. Schneider, eds.), vol. 314 of *Lecture Notes in Computer Science*, pp. 106–120, Springer Berlin / Heidelberg.

[25] P. Eades and K. Sugiyama, "How to draw a directed graph," *J. Inf. Process.*, vol. 13, pp. 424–437, Apr. 1991.

[26] M. Jünger, E. Lee, P. Mutzel, and T. Odenthal, "A polyhedral approach to the multi-layer crossing minimization problem," in *Graph Drawing* (G. DiBattista, ed.), vol. 1353 of *Lecture Notes in Computer Science*, pp. 13–24, Springer Berlin / Heidelberg.

[27] K. Sugiyama, *Graph Drawing and Applications for Software and Knowledge Engineers*. Series on Software Engineering and Knowledge Engineering, World Scientific, 2002.

[28] N. S. Nikolov and A. Tarassov, "Graph layering by promotion of nodes," *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 848 – 860, 2006.

A Additional benchmark data

Here we show tables of normalised data for the different criteria for both the Ericsson and North DAG graph sets.

Graph set	1st	2nd	3rd	4th
Ericsson	LP 0.827	G 0.836	D 0.988	S 1.348
North DAG's	LP 0.539	G 0.611	D 0.895	S 1.953

Table 4: Normalised average of cycles removed for both graph sets.

Graph set	Measurement	1st	2nd	3rd	4th
Ericsson	Dummy vertices	LP 0.661	L 0.792	CS 1.127	C2 1.326
North DAGs	Dummy vertices	LP 0.461	L 0.657	CS 1.232	C2 1.649
Ericsson	Layer height	L 0.945	LP 0.945	CS 0.994	C2 1.116
North DAGs	Layer height	L 0.877	LP 0.888	CS 1.007	C2 1.225

Table 5: Normalised average of layering measurements for both graph sets.

Graph set	1st	2nd	3rd	4th	5th
Ericsson	LP 0.245	BT 0.458	MT 0.470	M 1.749	B 1.917
North DAGs	LP 0.300	MT 0.515	BT 0.528	M 1.794	B 1.860

Table 6: Normalised average of crossings for both graph sets.

Graph set	Measurement	1st	2nd
Ericsson	Bends	LP 0.884	P 1.115
North DAGs	Bends	LP 0.917	P 1.082
Ericsson	Average angles	LP 0.741	P 0.755
North DAGs	Average angles	LP 0.643	P 1.356
Ericsson	Max angles	LP 0.806	P 1.194
North DAGs	Max angles	LP 0.795	P 1.204
Ericsson	Area	LP 0.460	P 1.540
North DAGs	Area	LP 0.684	P 1.315

Table 7: Normalised average of coordinate measurements for both graph sets.