Automatic Layout of Graphs for LOEWE Automated Testing System

Master Thesis

at the University of Applied Sciences Hof
Faculty IT
Master of Software Engineering for Industrial Applications

Presented to	Presented by
Prof. Dr. Schaller	Markus Koenig
Alfons-Goppel-Platz 1	Fichtelgebirgsstrasse 9
95028 Hof	95173 Schoenwald
Germany	Germany

19th June 2007

Contents

I. Introduction x

1. Introduction 1

1.1. LOEWE Automated Testing System . . . 2
1.2. Definitions . . . 3
1.3. Graphical Notations . . . 5
1.3.1. Element Symbol . . . 5
1.3.2. Control-Flow Operators . . . 5
1.3.3. LAUTS Elements . . . 5
1.4. Problem Definition . . . 6
1.5. Project Aims . . . 6
1.6. Overview . . . 6

2. Basics of Graph Drawing 7

2.1. Diagram Language . . . 7
2.2. Aspects of Graph Drawing . . . 8
2.3. Details of Graph Drawing Parameters . . . 10
2.3.1. Graph Types . . . 10
2.3.2. Drawing Conventions . . . 11
2.3.3. Aesthetic Drawing Rules . . . 12
2.4. Graph Drawing Approaches . . . 13
2.4.1. Topology-Shape-Metrics Approach . . . 13
2.4.2. The Hierarchical Approach . . . 14
2.4.3. The Force-Directed Approach . . . 16
2.4.4. The Divide and Conquer Approach . . . 16

II. Graph Drawing 17

3. Classification of Graph Type 18

3.1. Definition given by LAUTS Structure . . . 18
3.2. Classification . . . 20
3.2.1. Special Case Sequence . . . 21
3.2.2. Conclusion of Classification . . . 21
3.3. Planarity . . . 21

4. Graph Transformations **23**
4.1. Defect-Draw-Repair . . . 23
4.2. Cycle Removal . . . 24
4.3. Cycle Substitution . . . 25
4.3.1. Additional Definitions . . . 25
4.3.2. The Algorithm of Cycle Substitution . . . 26
4.4. Bi-Treeing . . . 27
4.4.1. Augmentation . . . 28
4.4.2. Drawing . . . 28
4.4.3. Reverse Conversion . . . 29

5. Prearrangements for Rating Algorithms **31**
5.1. Rating of Graph Drawing Algorithms . . . 31
5.2. Arrangements for the Test Objects . . . 32
5.3. Characteristics of the Graphs used as Test Objects . . . 32
5.3.1. Shape of the Graph . . . 32
5.3.2. Size of the Graph . . . 33
5.4. Set of Graphs to Test . . . 34

6. Graph Drawing Algorithms **36**
6.1. Leaf-First-Layering . . . 36
6.1.1. Overview . . . 36
6.1.2. Mode of Operation . . . 36
6.1.3. Complexity . . . 40
6.1.4. Results . . . 40
6.1.5. Evaluation . . . 41
6.2. Dominance-Straight-Line . . . 42
6.2.1. Overview . . . 42
6.2.2. Mode of Operation . . . 43
6.2.3. Complexity . . . 45
6.2.4. Results . . . 45
6.2.5. Evaluation . . . 45
6.3. Improved Walker . . . 47
6.3.1. Overview . . . 47
6.3.2. Mode of Operation . . . 47
6.3.3. Complexity . . . 53
6.3.4. Results . . . 54
6.3.5. Evaluation . . . 54
6.4. Magnetic Spring Model - Centric . . . 55
6.4.1. Overview . . . 55
6.4.2. Mode of Operation . . . 56
6.4.3. Complexity . . . 59
6.4.4. Results . . . 59
6.4.5. Evaluation . . . 60

7. Analysis and Perspectives **62**
7.1. Comparison of the Algorithms . . . 62
7.1.1. Transformers . . . 62
7.1.2. Graph Drawing Algorithms . . . 65
7.2. Further Work and Perspectives . . . 67
7.2.1. Perspectives . . . 67
7.2.2. Further Work . . . 67

III. Appendix **68**

A. Tree Diagram Style Guide **69**
A.1. Placement of Vertices . . . 69
A.1.1. Vertical Vertex Layout . . . 69
A.1.2. Horizontal Vertex Layout . . . 70
A.2. Spacing . . . 70
A.2.1. Vertical Spacing . . . 70
A.2.2. Horizontal Spacing . . . 70
A.3. Edges . . . 71
A.3.1. Inter-Layer Edges . . . 71
A.3.2. Inner-Layer Edges . . . 71
A.3.3. Orthogonal Edges . . . 72

B. Set of Graphs used as Test Objects **73**
B.1. Details of the Test Objects . . . 73
B.1.1. Minimal Decision . . . 73
B.1.2. Minimal Sequence . . . 74
B.1.3. Minimal Other . . . 74
B.1.4. Pure Decision . . . 75
B.1.5. Pure Sequence . . . 75
B.1.6. Pure Other . . . 76
B.1.7. Special Case with 96 Vertices - Special_(100) . . . 76
B.1.8. Special Case with 10,450 Vertices - Special_(10000) . . . 77
B.1.9. Small Graph with $g_G^+ = 5$ - 1_(563) . . . 77
B.1.10. Huge Graph with $g_G^+ = 5$ - 1_(10139) . . . 78
B.1.11. Small Graph with $g_G^+ = 50$ - 2_(471) . . . 78
B.1.12. Huge Graph with $g_G^+ = 50$ - 2_(7311) . . . 79
B.1.13. Small Graph with $g_G^+ = 20$ - 3_(670) . . . 79
B.1.14. Huge Graph with $g_G^+ = 20$ - 3_(9551) . . . 80
B.1.15. Small Graph with $g_G^+ = 3$ - 4_(564) . . . 80
B.1.16. Huge Graph with $g_G^+ = 3$ - 4_(9787) . . . 81

C. Results of Drawings **82**
C.1. Leaf-First-Layering . . . 82
C.1.1. Min Decision . . . 82
C.1.2. Min Other . . . 82
C.1.3. Min Sequence . . . 82
C.1.4. Pure Decision . . . 82
C.1.5. Pure Other . . . 82
C.1.6. Pure Sequence . . . 82
C.1.7. Special(100) . . . 83
C.1.8. Special(10000) . . . 83
C.1.9. 1_(10139) . . . 84
C.1.10. 1_(563) . . . 84
C.1.11. 2_(471) . . . 84
C.1.12. 2_(7311) . . . 84
C.1.13. 3_(670) . . . 84
C.1.14. 3_(9551) . . . 84
C.1.15. 4_(564) . . . 85
C.1.16. 4_(9787) . . . 85
C.2. Dominance Straight Line . . . 85
C.2.1. Min Decision . . . 85
C.2.2. Min Other . . . 85
C.2.3. Min Sequence . . . 85
C.2.4. Pure Decision . . . 86
C.2.5. Pure Other . . . 86
C.2.6. Pure Sequence . . . 87
C.2.7. Special(100) . . . 87
C.2.8. Special(10000) . . . 88
C.2.9. 1_(10139) . . . 88
C.2.10. 1_(563) . . . 89
C.2.11. 2_(471) . . . 90
C.2.12. 2_(7311) . . . 91
C.2.13. 3_(670) . . . 92
C.2.14. 3_(9551) . . . 93
C.2.15. 4_(564) . . . 94
C.2.16. 4_(9787) . . . 95
C.3. Improved Walker . . . 95
C.3.1. Min Decision . . . 95
C.3.2. Min Other . . . 95
C.3.3. Min Sequence . . . 95
C.3.4. Pure Decision . . . 96
C.3.5. Pure Other . . . 96
C.3.6. Pure Sequence . . . 96
C.3.7. Special(100) . . . 96
C.3.8. Special(10000) . . . 97

C.3.9. 1_(10139) . . . 97
C.3.10. 1_(563) . . . 97
C.3.11. 2_(471) . . . 97
C.3.12. 2_(7311) . . . 97
C.3.13. 3_(670) . . . 97
C.3.14. 3_(9551) . . . 98
C.3.15. 4_(564) . . . 98
C.3.16. 4_(9787) . . . 98
C.4. Magnetic Spring Model - Centric . . . 98
C.4.1. Min Decision . . . 98
C.4.2. Min Other . . . 98
C.4.3. Min Sequence . . . 98
C.4.4. Pure Decision . . . 102
C.4.5. Pure Other . . . 103
C.4.6. Pure Sequence . . . 103
C.4.7. Special(100) . . . 104
C.4.8. Special(10000) . . . 104
C.4.9. 1_(10139) . . . 105
C.4.10. 1_(563) . . . 106
C.4.11. 2_(471) . . . 106
C.4.12. 2_(7311) . . . 107
C.4.13. 3_(670) . . . 107
C.4.14. 3_(9551) . . . 108
C.4.15. 4_(564) . . . 109
C.4.16. 4_(9787) . . . 110

D. Definitions **111**

E. CD **116**

F. Affidavit **117**

List of Figures

2.1. Examples for the four basic language types. 7
2.2. A composite of net and region type. 7
2.3. An example that shows the two competitive rules *clear symmetry* and *minimal number of edge crossings*. 9
2.4. One possible classification of graphs. 10
2.5. Different drawings of the same graph with different edge drawing conventions. 11
2.6. Drafts for Vertex Placement Conventions. 12
2.7. Steps of the topology-shape-metrics approach. 14
2.8. Steps of the hierarchical graph drawing approach. 15

3.1. An example of a part of a test in the LAUTS' structure. 19
3.2. Difference regarding the Control-Flow Operator Sequence between the LAUTS structure and the drawn graph. 20
3.3. Kuratowski graphs. 22
3.4. Proofing planarity of Sequence. 22

4.1. Operating principle of the Transformers. 23
4.2. Edges that can be removed by Defect-Draw-Repair method. 24
4.3. Conversion of a graph with Cycle Removal method. 25
4.4. Process of Cycle Substitution. 27
4.5. Cases to distinguish for bi-tree conversion. 28
4.6. Process of bi-treeing. 29
4.7. Reverse Conversion of the bi-tree. 30

5.1. Influence of a Sequence to tree-depth. 34

6.1. Minimum- and maximum-layer-approach for Leaf-First-Layering. 37
6.2. Placement of bends in upward and downward edges for Leaf-First-Layering. 39
6.3. Exemplary results of the Leaf-First-Layering algorithm. 41
6.4. The lower right vertex is reachable via two connections. 43
6.5. Exemplary results of the Dominance-Straight-Line algorithm. 46
6.6. Edge length differences at one node. 47
6.7. Illustration of thread-pointer in Improved-Walker algorithm. 48
6.8. Inner and outer contours of two subgraphs. 50
6.9. The worst case for the apportion-procedure of Improved Walker algorithm. 53
6.10. The ideal case for the apportion-procedure of Improved Walker algorithm. 53

6.11. The drawings of the smallest graphs done by Improved Walker. 54
6.12. Required area and edge lengths are less at graphs with crossings. 55
6.13. Illustration of the Magnetic Spring Model. 56
6.14. Common types of magnetic fields. 56
6.15. Preprocessing for Magnetic Spring Model. 57
6.16. Five drawings of the same graph with Magnetic Spring Model - Centric. . 60

7.1. Problem of drawings done after Defect-Draw-Repair method. 64
7.2. Complexity of graph drawing algorithms. 66

A.1. Layer-oriented appearance. 69
A.2. Height optimization problem. 70
A.3. Inter-layer edges with a Sequence as cycle root. 72

List of Tables

1.1. Symbols used for Control-Flow Operators. 5
1.2. Symbols used for LAUTS Elements. 5

6.1. Result of Leaf-First-Layering. 41
6.2. Result of the Dominance-Straight-Line. 46
6.3. Result of the Improved Walker algorithm. 54
6.4. Average result of Magnetic Spring Model with a centric magnetic field. . 60

A.1. Appearance of edges subject to the preceding node. 71

C.1. Result of first run of Magnetic Spring Model with a centric magnetic field. 99
C.2. Result of second run of Magnetic Spring Model with a centric magnetic field. 99
C.3. Result of third run of Magnetic Spring Model with a centric magnetic field. 100
C.4. Result of fourth run of Magnetic Spring Model with a centric magnetic field. 100
C.5. Result of fifth run of Magnetic Spring Model with a centric magnetic field. 101

Part I.

Introduction

1. Introduction

More and more of todays electronical equipment contains microprocessors. This starts with devices of daily use like toasters [16] and ends with highly automated factories. With devices more complex than toasters, one microprocessor alone is not enough anymore. Normally, it is not enough to just have some processors - they also have to work together. This means they have to communicate with each other which is typically done via electric signals. Unfortunately, many problems arise with that. Different temperatures and humidity, broken cables or interferences diversifying the signal due to induction are only some examples of physical disturbances which have to be handled.

A microprocessor on its own does nothing. Software is needed to define the instructions. However, with the employment of software further problems appear. In addition to bugs like infinite loops or wrong memory allocations which are noticed relatively easy in a crash or a not responding of the processor, there are errors which are hard to find. For example, a fault in a rarely used part of a module's implementation which does not lead to a crash but causes wrong results would be a worst case.

We are quite used to have software problems frequently with our personal computers (which is annoying enough). But with the increasing number of microprocessors in daily used devices, these problems arise there, too. While a crashing clock radio might still be an annoyance, a failing processor in a car can matter of life and death.

In case of TV set, the effects of a crashing processor or a software failure are at first sight not that critical. However for a company like LOEWE, which produces high quality and luxury TV sets, such errors would have a negative impact on the image of the company. In TV sets, the number of embedded processors has increased significantly during the last few years. The growing display size, High-Definition Television (HDTV) and digital television are only some of the reasons for that.

Problems arise even if there were changes to only one module of the device. It is not only necessary to ensure that the module itself works fine but also that the overall system containing this module still works like before. Tests have to be done to ensure that the whole hardware and software works fine. As always, testing is a long-winded, often tedious process and as such it costs a lot of money. This is even more annoying when considering the fact that the test cases for one product are very similar all the time. For instance the test for switching the program would not change only because there was a software update for image processor.

Processes that are always the same are predestined to be automated. This holds true both for assembly belts and software. For that purpose, the LOEWE Automated Testing System (LAUTS) is developed within project "Software Platform for Intelligent Consumer Electronics" in a cooperation of LOEWE and the University of Applied Sciences Hof. LAUTS is able to execute and evaluate test cases. One of the major demands is

that the software is usable without knowledge about (textual) programming languages. Therefore, a graphical representation (diagram) of the test cases is used.

1.1. LOEWE Automated Testing System (LAUTS)

LAUTS is a software which is used mainly for testing TV sets but is not limited to this appliance. It is capable of executing and evaluating test cases on every command-based system. The main focus is on devices with bus-based communication where LAUTS acts as an additional module connected to this bus. It listens to all of the communication on the bus and is able to emit own commands for testing if the necessary reactions occur.

However, LAUTS is also able to handle direct connections (which are seen as a buses with only two modules). Additionally, it can handle different communication channels simultaneously.

Another possible application area is the simulation of modules at least in some degree. This was never an objective and is actually a by-product. Nevertheless it might be useful in the development of new systems, where not all modules yet exist. Moreover, it could be used to test single modules by simulating the others to ensure that this module works as specified.

Two types of commands are distinguished in LAUTS. A command sent by LAUTS is termed “Action” as it is an activity done by LAUTS on the bus of the “System Under Test” (SUT). Accordingly a command originating from one of the other modules is named “Reaction”. Those two types are mapped to the Action-Node and the Reaction-Node in LAUTS’ diagrams. From all the elements in a LAUTS’ diagram these two are closest to the SUT (low-level).

Opposite these two nodes, TestSuite-Node depicts an executable test (highest level). There may only be one TestSuite-Node per test at maximum as it in fact represents the whole test itself. To be able to reuse parts of the tests, TestCase-Node and the Function-Node were introduced. Both are in between the TestSuite- and the Action- respectively Reaction-Node. One TestCase-Node (including its inferior nodes) can be reused in several tests at once. The same is true for the Function. This node additionally allows to pass parameters which is useful if the same part is used in several contexts. Those two node types facilitate the maintainability of existing tests as well as the creation of new tests as those can use already existing components.

Up to now, only containers (TestSuite, TestCase and Function) and actors (Action and Reaction) have been addressed. To model the interdependency, Control-Flow-Operators are used. Each of these elements has exactly one parent node and all except one have an arbitrary number of child nodes. The one exception is mentioned below. Three basic types are necessary. The first one is the Sequence-Node. It implies that the child nodes are handled in exactly the order given by the user.

The second one is Random-Node which selects its child nodes randomly. Random exists in three characteristics: RandomAll-Node which assures that each of its child nodes is chosen exactly once, RandomPutBack-Node picks out a specified number of child nodes whereas some children may occur several times while others occur not at all.

In its pure form a specified number of its children is randomly chosen at which each one may occur only once.

The exception mentioned before is the Decision-Node which has exactly two child nodes. The one to use is chosen by a boolean expression[1] (e.g. x > 5).

To model the tests all those elements, or more precisely their graphical representations, are arranged in diagrams. LAUTS is open to support different diagram types. As a start tree(like)[2] diagrams are used. The elements represent the nodes and the relations are mapped as connections between them.

Especially if large tests are modelled, the need to automatically layout the diagram becomes clear. The user should be able to focus on modelling the test itself instead of wasting much time with arranging the nodes. Although layout is not important for the functionality, a clearly laid out diagram is crucial for understanding its meaning. This is important for example during maintenance of the test pictured by the diagram. This thesis discusses and evaluates different techniques and algorithms to layout the special kind of graph used by LAUTS' diagrams.

1.2. Definitions

This section defines most of the terms used in this thesis. For a complete mathematical definition have a look at Appendix D.

- **Graph:** A graph is a structure consisting of a finite set of vertices (or nodes) which are connected by edges (also called connections, links, or arcs). The mathematical definition is as follows[3]: A Graph G consists of an ordered pair $G := (V, E)$ where $V \neq \emptyset$ and $V \cap E = \emptyset$. V (vertex) and E (edge) are a finite set each.
- **Drawing:** For this thesis a drawing Γ is (one) representation of a graph. A graph itself has the information about nodes and the connections between them. However, nothing is given about the position, shape or size of any of them. So a drawing is in its simplest case the result of a function mapping each vertex v to a distinct point $\Gamma(v)$ and each edge (u, v) to a line $\Gamma(u, v)$. A graph has an unlimited number of drawings. If no distinction between the graph and the drawing is required, the term "the edge (u, v) is ..." and "the drawing $\Gamma(u, v)$ is ..." are the same [4].
- **Diagram:** Diagram is used in three contexts. Firstly, it is a synonym for drawing and secondly, it is used as a short version of "diagram type". Thirdly, it describes the space in which the drawing is placed. If the distinction between these meanings

[1] A boolean expression means that the result is of type boolean. The operands may be of any other type.

[2] The diagram is not really a tree diagram as by definition there are no cycles allowed in a tree while this diagram type is able to display such for the Sequence-Nodes. This topic is discussed in detail in chapter 3.

[3] The definition given here is a shortened one for undirected graphs. For the complete definition and the definition of directed graphs have a look at appendix D.

is not clear on the basis of the context, it will be specially mentioned which types of diagram is meant.

- **Diagram Language:** The diagram language is the sum of all rules which exist for creating and reading a diagram type. Thus it can be seen as the definition of a diagram type.
- **Syntactic grammar:** The syntactic grammar specifies the rules of arrangement for the elements in a diagram. For instance, the rules which elements are allowed to be connected with each other are part of the syntactic grammar.
- **Unit of Meaning:** The unit of meaning of a diagram defines the elements appearing in the diagram. Together with the syntactic grammar, it forms the Meta-Model of the diagram.
- **Aesthetic Criterium:** An aesthetic criterium is a rule concerning the shape of a drawing to make it "look good" to the user.
- **Vertex, Node:** Those two expressions are used for the elements out of the set V of a graph.
- **Edge, Connection, Arc, Link:** These four terms refer to the elements out of the set E of a graph. For directed edges "outgoing" means that the edge starts at a vertex and "incoming" denotes that the edge ends at a vertex. Edges are visualized in drawings as lines between vertices.
- **LAUTS structure:** The LAUTS structure describes the elements inside the software LAUTS. These elements are the basis for the graph which will be drawn.
- **Control-Flow Operator:** (*short*: Control-Flow) is one of the two groups of elements in the LAUTS structure. It contains the elements: *Sequence*, *Random*, *RandomPutBack*, *RandomAll* and *Decision*.
- **LAUTS Element:** A LAUTS Element is one of two groups of elements in the LAUTS structure. It contains the elements: TestSuite, TestCase, Function, Action and Reaction.
- **Top-Down:** Top-Down means that something starts at the top and ends at the bottom. Top and bottom are seen either as coordinates or in terms of graph theory as root (top) and leafs (bottom).
- **Bottom-Up:** Bottom-Up means that something starts at the bottom and ends at the top. Top and bottom are seen either as coordinates or in terms of graph theory as root (top) and leafs (bottom).

1.3. Graphical Notations

For all diagrams and outlines in this thesis, there are defined common symbols for all elements of LAUTS.

1.3.1. Element Symbol

A symbol which can be any LAUTS Element or Control-Flow Operator has the following appearance: (*something*)

1.3.2. Control-Flow Operators

Control-Flow Operators are represented by a sign surrounded by a circle (e.g.Ⓐ for Sequence). Table 1.1 shows all the symbols.

Symbol	Control-Flow Operator
(&)	Sequence
(|)	Random
(↺)	RandomPutBack
(A)	RandomAll
(Y)	Decision

Table 1.1.: Symbols used for Control-Flow Operators.

1.3.3. LAUTS Elements

LAUTS Elements are shown as rectangles surrounding the written type (e.g. [TestCase]). Table 1.2 shows all the symbols.

Symbol	LAUTS Element
[TestSuite]	TestSuite
[TestCase]	TestCase
[Function]	Function
[Action]	Action
[Reaction]	Reaction

Table 1.2.: Symbols used for LAUTS Elements.

1.4. Problem Definition

This thesis is assigned to LAUTS, a software for automated testing developed in the SPLICE project at the University of Applied Sciences Hof. More precisely, the thesis is about automatically drawing the graphs which represent the structure of tests in LAUTS.

As these structures may become very big, it has to be possible to automatically layout the graph. This has to be done both in good time and in a user-friendly layout.

1.5. Project Aims

This thesis aims to find an algorithm or a combination of algorithms which create a visual representation of the structure of information used in LAUTS to represent tests in easily comprehendible layout and reasonable time.

1.6. Overview

The thesis is divided into three parts. In the introduction, common definitions and graphical notations as well as the problem definition and the aim of the thesis are given. Furthermore, the basics of graph drawing are described to provide the fundamental knowledge about this topic which is necessary to understand the subject matter discussed in the following chapters.

In the second part, the different graph drawing algorithms will be analyzed and evaluated. For drawing graphs, it first has to be clarified which kind of graph is used for describing the test structure. Chapter 4 deals with methods to convert one graph type into another offering the possibility to apply graph drawing algorithms, which are not applicable to this type of graph without that conversion. Subsequently, the possibilities of rating algorithms are discussed so that different algorithms can be compared to each other. In chapter 6, graph drawing algorithms are explained, the results are presented and finally evaluated. The thesis culminates in a final discussion of the most appropriate algorithm for LAUTS' specific graph type.

The appendix forms the third part of the thesis. Appendix A contains a style guide describing the goal of the drawing process. All aesthetic rules for the graph drawing algorithms are derived from this description. In appendix B, the set of graphs which are used as input for the algorithms are described. This is followed by appendix C containing the results of the drawings produced by the tested algorithms. Finally, appendix D contains the mathematical definitions which provide the basis for this thesis.

2. Basics of Graph Drawing

2.1. Diagram Language

A diagram is an environment for the drawing of a graph with clearly defined rules. The process of producing a drawing (Γ) out of a graph (G) is called *graph drawing*. The rules of how to draw are defined in the *diagram language*. This language consists of two parts. The *units of meaning* which define the elements of the diagram and the *syntactic grammar* which specifies the rules of arrangement of the elements [12]. Together they define the *meta-model* of the diagram.

Referencing [12], there are four basic types of diagram languages: *matrix*, *net*, *region* and *coordinate type*. Figure 2.1 shows examples of these types. Normally, diagrams are not a pure type but rather a mixture of those basic types. Figure 2.2 shows an example for a composite of net and region type.

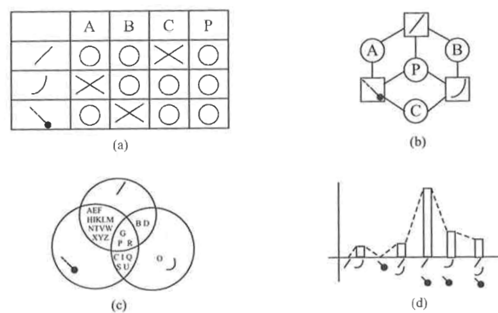


Figure 2.1.: Examples of the four basic language types (cp. [12], p. 2): (a) matrix type; (b) net type; (c) region type; (d) coordinate type.

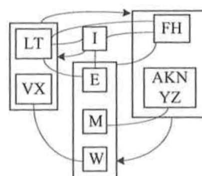


Figure 2.2.: A composite of net and region type (cp. [12], p. 2).

2.2. Aspects of Graph Drawing

For graph drawing, the following aspects have to be taken into account ([12] chapter 2, [4] chapter 2):

- **Drawing Object:** Defines the type of graph which is the object to draw. This has to be taken into consideration because not every graph drawing algorithm can handle every type of graph (see *2.3.1 Graph Types* for further detail).
- **Drawing Conventions:** These are the conventions concerning the placement of vertices and the routing of the edges which must be fulfilled. As the conventions concerning vertices are quite different from those for edges, there is a distinction between *placement conventions* and *routing conventions*. Drawing conventions are discussed in more detail in section 2.3.2. The drawing conventions for edges are defined in table A.1 in the Style Guide.
- **(Aesthetic) Drawing Rules:** While drawing conventions only generally define the possible placement of vertices and routing of edges for each possible graph, drawing rules define the concrete placement and routing for a specific graph. These rules are often referred to as "*Aesthetic Rules*" because they decide on how the graph will look like and if it looks "nice". The graph is determined by different types of rules:
  - **Static Rules:** The rules that are applied if the graph is drawn for the first time.
    * **Semantic Rules:** Derive the placement (vertex) and routing (edge) rules from the *meaning* of the vertex or edge. The *meaning* has to be defined in some way (e.g. the root-node is placed on the topmost layer).
    * **Structural Rules:** Do not take the meaning of the vertex or an edge into account. These rules consider each vertex or edge to be equal. The only factor considered is the graph-theoretic features of the graph to draw (e.g. If vertex a follows vertex b, draw vertex b below vertex a).
  - **Dynamic Rules:** Are used if the (already) drawn graph is redrawn because of a change. Those rules are not of interest for this thesis.

  In addition to these types, there are four axis of classification used for drawing rules.

  1. Determination if the rule has a unique solution.
  2. Determination whether the rule is *topological*, *shape-oriented* or *metric*.
  3. Determination whether the rule applies to the whole graph (globally) or only to parts of the graph (locally).
  4. Determination if the rule is *hierarchical*, *flat* or *both*.

Normally the aesthetic rules are an optimisation of something, for example, a minimisation of the total number of edge crossings or the maximum symmetry. A variety of possible drawing rules is given in chapter 2.3.3.

Normally the aesthetic criteria are conflicting. And even if not, it is difficult for the algorithm to deal with all of them at once (cf. [4], p. 17). Thus there has to be an order which defines the importance of a rule / convention. This is called *Priority Relationship.*

- **Priority Relationship:** It might not be possible to reconcile all the rules / conventions at once. For example, a first rule is that the graph should have a clear symmetry and the second is to have as few edge crossings as possible. Both rules might be possible on their own. But if both rules should be applied simultaneously, there would not be a solution (see figure 2.3). This is why there have to be priorities for rules and conventions defining which rule should be applied first in such cases.

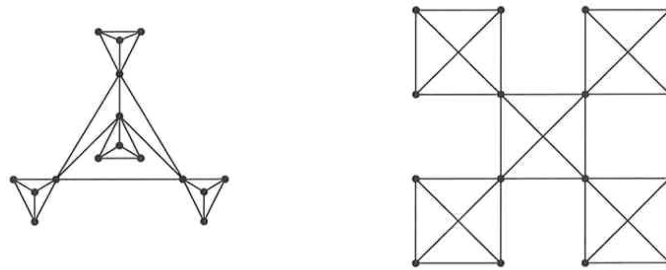


Figure 2.3.: An example that shows the two competitive rules *clear symmetry* and *minimal number of edge crossings.* On left side the *minimal number of edge crossing* and on the right side clear symmetry has the higher priority (cp. [5]).

In addition to the priorities which have to be set individually for each graph type, there is one global rule: drawing conventions have a higher priority than drawing rules. This is obvious because by definition drawing conventions always have to be fulfilled whilst drawing rules are goals which should be fulfilled.

- **Features of Drawing Algorithms:** According to [12] page 17, algorithms can be divided into five categories:

  1. "Algorithms that use graph theory and graph algorithm."
  2. "Heuristic algorithms."
  3. "Those that use force directed models (such as the ring and spring model), thermodynamics models (simulated annealing), bioinformatics model (genetic

algorithm) and other simulations."[1]

4. "AI algorithms such as *layout by example*."
5. "Hybrid algorithms combining different types of algorithms mentioned above."

- **Computational Efficiency:** Efficiency is nothing directly seen by the user. Depending on the field of application it is more or less important. If the algorithm has to do the drawing in real-time, it is fundamental that the result comes up fast. On the other hand, if the computation may take several hours, the computational efficiency is not as crucial.

Of course this is neither the only way to classify algorithms for graph drawing nor is it an official standard. This classification is used in this thesis to have a single framework to classify and evaluate the algorithms.

2.3. Details of Graph Drawing Parameters

2.3.1. Graph Types

Depending on its properties, a graph is assigned a type. Image 2.4 shows the different classes of graphs. It has to be noted that a graph may be a combination of more than one class. For example a binary rooted tree with directed edges is a binary tree as well as a directed graph (digraph) which is of course acyclic by definition. As those classifications are standard in graph theory, they will not be explained here in detail.

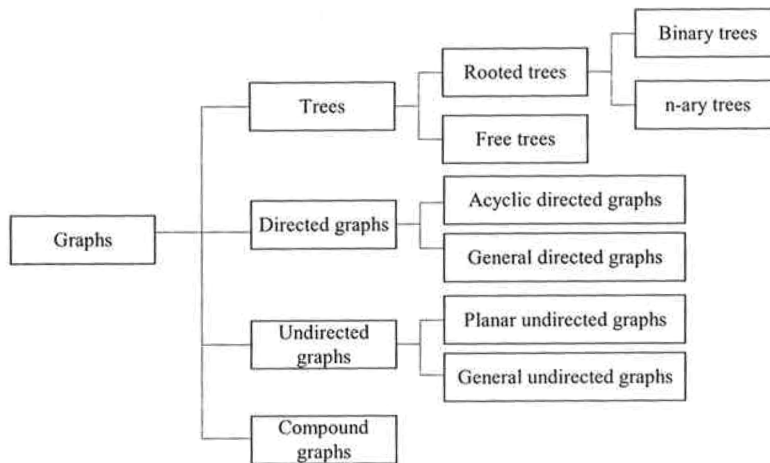


Figure 2.4.: The classification of graphs which is used in this thesis (cp. [12], p. 8).

[1]In this thesis, such algorithms will be summarized as natural law algorithms, as they emulate the laws of nature inside a computer to do the layout.

2.3.2. Drawing Conventions

Normally, there are a lot of drawing conventions for vertices but only a few conventions for edges.

2.3.2.1. Edge Drawing Conventions

The drawing conventions presented here are adopted from [12] page 9 - 11 and [4] page 12 - 17.

Poly-line Drawing The edges are drawn as a chain of straight lines. The end points of such straight lines which are neither the beginning nor ending of the whole edge are called *via* or *bend*. An example is shown in figure 2.6(a).

Straight-line Drawing Edges are drawn as one straight line connecting the two incident vertices. Figure 2.6(b) shows an example.

Orthogonal Drawing Edges are drawn as poly-lines altering horizontal and vertical segments. An example is shown in figure 2.6(c).

Directed Drawing This is usable for directed graphs only. It tries to draw the graph so that (most of) the edges point to the same direction (upward, downward, left, right). If all edges point to the same direction it is called *strict directed drawing*.

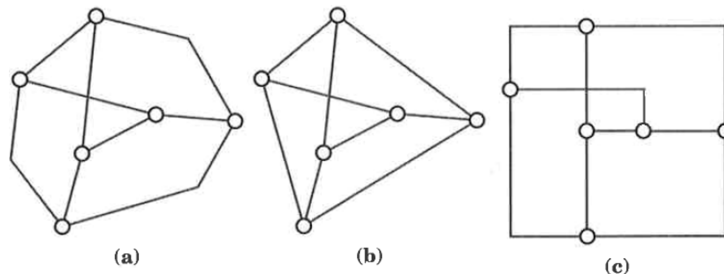


Figure 2.5.: Drawings of the same graph with different edge drawing conventions: (a) poly-line; (b) straight-line; (c) orthogonal. ([4], p. 13).

2.3.2.2. Vertex Drawing Conventions

In [12], frequently used types of vertex drawing conventions are presented . The following itemization presents the fundamental types of these conventions. Of course combinations between the types are possible. Figure 2.6 shows examples for these vertex drawing conventions.

- **Free Placement:** This is equivalent to no rules at all because it allows placing the vertices everywhere.
- **Parallel Line Placement:** Allows placing the vertices on (or in between) parallel lines. Naturally, these lines are horizontal or vertical. But the lines are not limited to that alignment. They can be in any angle to the coordinate axis as long as the lines are still parallel to each other. The horizontal parallel line placement is widely used and called *layered placement*. Figure 2.6(a) shows an example.
- **Concentric Circle Placement:** The vertices are arranged on (or in between) concentric circles. Of course variations are possible. For example, a tree drawn with parallel line placement might only use the lower 180 degrees of the circles. This convention has the advantage that there is an equal distance between the root and any vertex on a layer. An example is shown in figure 2.6(b).
- **Radial Line Placement:** Allows the placement of vertices on (or in between) radial lines. An example for that is shown in figure 2.6(c).
- **Orthogonal Grid Placement:** Places the vertices on an orthogonal grid. Either on the crossing of a horizontal and a vertical line or in the area surrounded by two neighbouring horizonal and two neighbouring vertical lines. In figure 2.6(d) an example is shown.
- **Polar Grid Placement:** This is some kind of mixture between a concentric and an orthogonal placement. The vertices are placed on the crossings of the concentric circles and the radial lines. The example is shown in figure 2.6(e).

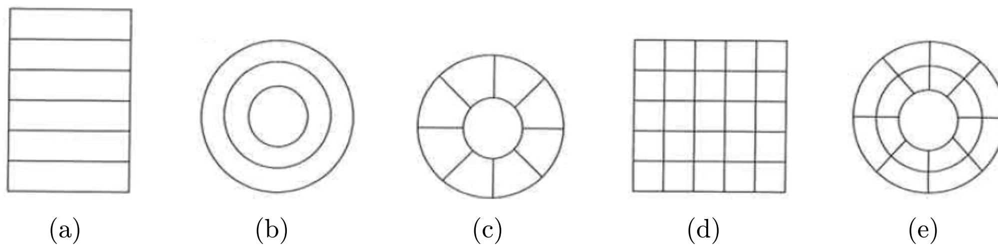


Figure 2.6.: Drafts for vertex placement conventions. (a) Parallel Lines, (b) Concentric Circles, (c) Radial Lines, (d) Orthogonal Grid, (e) Polar Grid. (cp. [12], p. 9).

2.3.3. Aesthetic Drawing Rules

In the following, the most important aesthetic drawing rules are shortly introduced (cp. [4], p. 14 - 16).

Minimisation of Edge Crossings The total number of crossing edges should be as small as possible. The ideal case would be a planar graph.

Minimisation of Area The area used by the elements of the drawing should be as narrow as possible. If the area of the drawing becomes too big, it becomes very difficult to get an overview. There are different options of measuring the area:

- The smallest polygon covering the whole graph (circumscribing polygon).
- The smallest rectangle covering the whole graph (circumscribing rectangle).
- The smallest circle covering the whole graph (circumscribing circle).

Naturally, other ways of measurement are possible. For this thesis, the circumscribing rectangle will be used because the graphs are drawn for visualisation on computer screens and for printing, which are both rectangular media.

Total Edge Length The sum of the lengths of all edges should be as small as possible.

Maximum Edge Length The maximum length of one edge should be as small as possible.

Uniform Edge Length The edges should all have approximately the same length.

Cluster A subset of vertices of the graph should appear preferably close together.

Shape A subset of vertices of the graph should be drawn in a predefined shape.

2.4. Graph Drawing Approaches

This section presents some of the standard approaches used for graph drawing. Of course, these are only a selection from the large number of available algorithms and methods.

2.4.1. Topology-Shape-Metrics Approach

This method was proposed in [1],[14] and [15]. It constructs orthogonal grid drawings and facilitates a homogenous treatment of many (aesthetic) rules and criteria. Tree steps are to be done:

1. **Planarization:** Determines the topology of the drawing. In this step, the number of edge crossings is to be reduced as much as possible.

2. **Orthogonalization:** Determines the shape of the drawing. Here, orthogonal representation of the graph has to be created. This means that the vertices do not yet have concrete coordinates but each edge gets a list of angles.

3. **Compaction:** Determines the final coordinates of the vertices and the vias of edges. Normally, the main problem here is minimizing the area for the drawing.

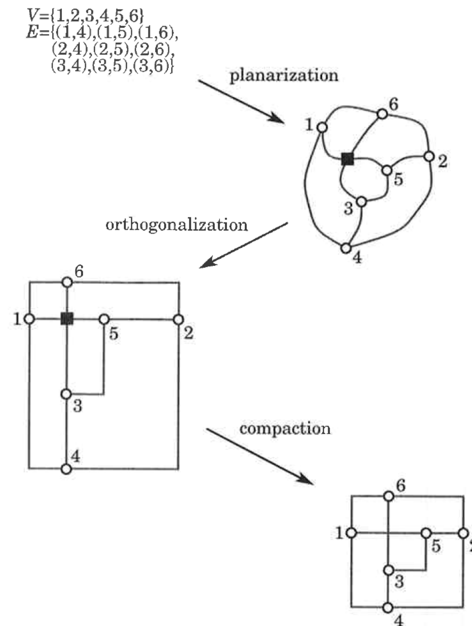


Figure 2.7.: Steps of the Topology-Shape-Metrics approach. (cp. [4], p. 21).

With the Topology-Shape-Metrics approach the minimisation of edge crossings has the the highest priority, followed by the minimisation of bends. Minimization of the area needed for drawing has the lowest priority. This is due to the order of the steps. The step done first has the most influence on the drawing as it has the most freedom. Each following step has to consider the work done by the steps before. Figure 2.7 shows the order of the steps graphicly.

This approach is open for additional rules and constraints. The step in which the rules are applied depends on the influence they should have on the drawing. If, for instance, the additional rule concerns the initial placement of the vertices, the first step is the one to choose. A rules affecting the routing of the edges, is to be added to the second step. And finally, if the rule has an effect on the concrete placement of vertices and bends, it has to be applied in the third step.

2.4.2. The Hierarchical Approach

The hierarchical approach originally proposed by [13],[3] and [18] is a five-step (for acyclic digraphs three-step) process. Figure 2.8 shows a draft of this approach. Figure 2.8(a)

visualizes the process for acyclic graphs and (b) shows the additional steps for cyclic graphs.

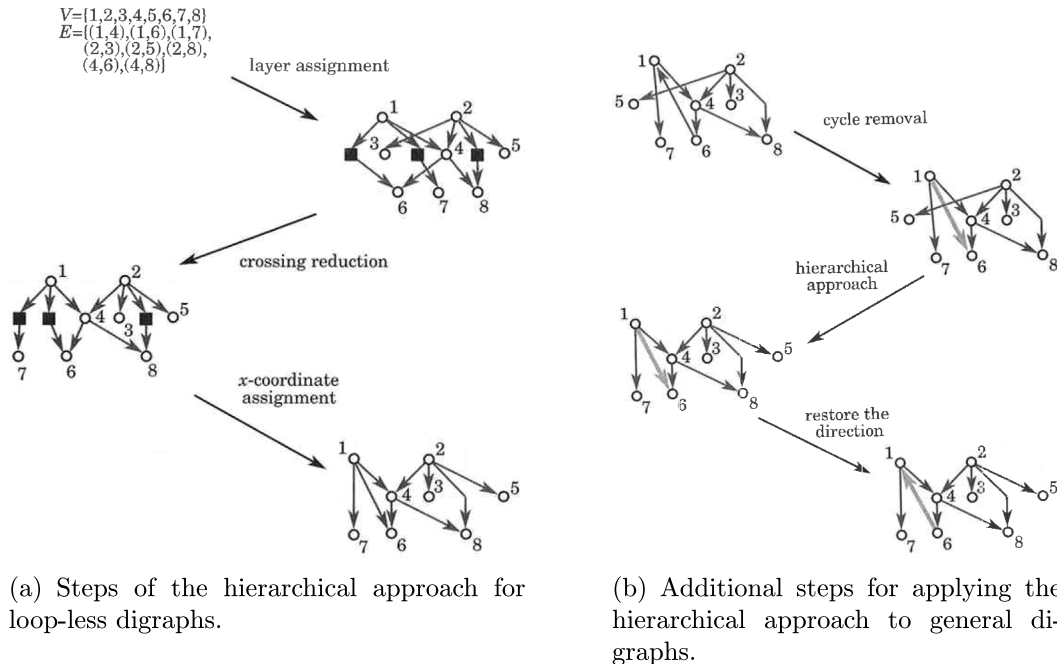
1. **Decycling:** (Only for cyclic graphs.) As the hierarchical approach only works for acyclic graphs, cyclic ones have to be converted to acyclic graphs (temporarily). This is done by reversing edges belonging to the cycles (see *4.2 Cycle Removal*). The set of reversed edges should be as small as possible.

2. **Layer Assignment:** Produces a layered digraph by assigning the vertices of graph G to the layers $L_1, L_2, ..., L_h$ such that if $e(u, v)$ with $u \in L_i$ and $v \in L_j$ then $i < j$ (inter layer edge). Layering means that all vertices on a layer L_k will have the same y-coordinate. Next, the insertion of dummy nodes will assure that an edge only bridges neighbouring layers ($j = i + 1$).

3. **Crossing Reduction:** The task for this step is to find an order for the vertices on each layer so that the total number of edge crossings is as small as possible.

4. **X-Coordinate Assignment:** In the prior step, only the order of the vertices was defined for each layer. Now the assignment of x-coordinates is done, strictly keeping the horizontal order of the vertices.

5. **Recycling:** (Only for cyclic graphs.) The edges reversed in step one are set back to their right direction.



(a) Steps of the hierarchical approach for loop-less digraphs.

(b) Additional steps for applying the hierarchical approach to general digraphs.

Figure 2.8.: Steps of the hierarchical graph drawing approach. (cp. [4], p. 23 - 25).

Additional rules and constraints can only be applied in step four. This is because all steps done before (and afterwards) are strictly defined and normally have no allowance for additional rules or constraints.

2.4.3. The Force-Directed Approach

Algorithms using this approach simulate a physical system of forces to draw undirected graphs as straight-line drawings. The task is to find a *locally minimum energy configuration* (cp. [4], p. 303). Of course drawings of directed graphs are also possible for example, by just ignoring the directions of the edges. If straight-line drawing is not wanted, this approach can be used too. First, the algorithm is used to layout the drawing. Second, the shape of the edges is considered in an additional step for instance the edges are drawn in orthogonal style.

This approach has two prerequisites:

1. A force model: for example considering the edges as springs and the vertices as connections between these springs. This approach is also known as "Rings and Springs" in literature. Another possibility would be to have a model where the vertices are considered to be particles of equal charge so that they reject each other.

2. The technique to find the *locally minimum energy configuration*. Normally simple iterative methods are used for this purpose.

Additional rules and constraints can be used at arbitrary positions, for example by defining special forces to have a set of vertices in a specific region (e.g. clustering).

2.4.4. The Divide and Conquer Approach

Divide and conquer is a technique widely used in computer science and also in graph drawing. First, the graph is split into subgraphs and then these subgraphs are drawn recursively. In the end, the subgraphs are glued together to have the drawing of the complete graph. There are two major steps for trees or tree-like graphs:

1. **Layer Assignment:** Assign each vertex of the graph to one layer (see *chapter 2.4.2*).

2. **Divide and Conquer:** Use the divide and conquer algorithm. For different graph types, the respective algorithms differ substantially.

Part II.

Graph Drawing

3. Classification of Graph Type

In order to find a suitable algorithm for graph drawing, it is necessary to ascertain the type of graph as some graph drawing algorithms work better (or only) for a specific type of graph. Furthermore, the knowledge about the graph type is helpful to define the drawing constraints' combinatorial properties given by the type (cp. [4]).

3.1. Definition given by LAUTS Structure

In LAUTS, there are different elements which occur as vertices in the graph: TestSuite, TestCase, Function, Action, Reaction, Sequence, Random, RandomPutBack, RandomAll and Decision. The connections between these elements are seen as "contains" relationships. Nevertheless, the vertices are referred to as *preceding-* or *parent vertex* for the containing and *following-* or *child vertex* for the contained vertex. If a distinction between these terms becomes necessary, it will explicitly be mentioned.

Rules are defined which element type may follow on a preceding one. Therefore, the vertices are divided into two groups. Sequence, Random, RandomPutBack, RandomAll and Decision are termed *Control-Flow Operators* and TestSuite, TestCase, Function, Action and Reaction called *LAUTS Elements*.

To determine the graph type, it is necessary to consider the design of the graph structure. As the graphical shape of the vertices does not matter, the following itemization contains the rules concerning the structure of LAUTS' graphs. Since the connections between the elements of the pure structure and the connections drawn in the diagram are not the same, one has to distinguish between preceding (following) and containing (contained) elements. Containing (contained) will be used to refer to the elements of the LAUTS structure. Preceding (following) will be used for vertices of the graph which are visualized in the diagram.

- Rules for vertices:
  - A graph begins with one of the following LAUTS Elements: TestSuite, TestCase or Function.
  - Each leaf of a graph is a LAUTS Element.
  - Each LAUTS Element which is not a leaf is followed by exactly one Control-Flow Operator.
  - All Control-Flow Operators except Decision are followed by one or more LAUTS Elements, Control-Flow Operators or a mixture of both.

  - Decision is followed by exactly two LAUTS Elements, two Control-Flow Operators or a mixture of both types.
  - Each element except the topmost has exactly one preceding element.
  - The topmost element has no preceding element.

- Rules for edges:
  - Each edge connects exactly two different vertices.
  - Depending on the containing element there are different rules which vertices are allowed to be connected to each other.
    * Containing element is a LAUTS Element: Each LAUTS Element can have exactly one undirected[1] outgoing connection in LAUTS structure. (except Reaction which must not have any outgoing edges). This edge is shown as (1) in figure 3.1. In the graph to draw a containing element may have a second outgoing connection which is directed. That is the case if the element is part of a Sequence. Sequences are discussed further below.

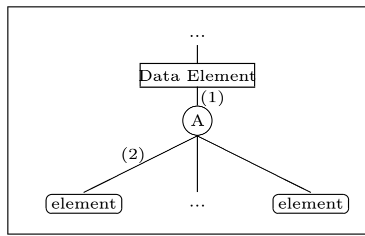


Figure 3.1.: An example of a part of a test in the LAUTS' structure.

    * Containing element is a Control-Flow Operator of any type except Sequence: The connection is exactly the same in the graph as in the LAUTS structure. It is undirected, begins at the containing element (preceding vertex) and ends at the contained element (following vertex). In figure 3.1 such an edge is labeled with (2).
    * Containing element is of type Sequence: The Sequence is the only element of the LAUTS' structure where the order of the contained elements matters. This order is visualized in the diagram by vectors between the contained elements. This is illustrated in figure 3.2.
      In this context, the connections are special in two ways. First of all the vectors are directed in the graph while the connections from all other Control-Flow Operators are undirected. Second, the decision which vertex u is connected from one vertex v depends on the position of the element represented by v in the list of elements contained by the Sequence

[1]An undirected edge can be seen as a directed one if this is necessary. The edge is always heading away from the root.

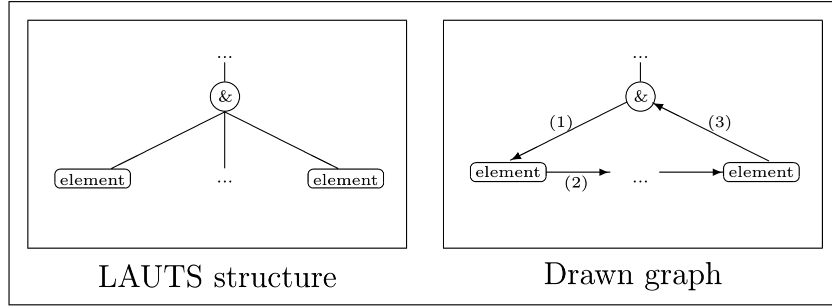


Figure 3.2.: Difference regarding the Control-Flow Operator Sequence between the LAUTS structure and the drawn graph.

(see *Appendix A*). Three cases are possible for the vertices representing an element of the list:

- The element is the first one in the list (*cf. A.3.1*): Beside the fact that the edge is directed, this vertex has a normal incoming one from the Sequence (v) vertex (see (1) in figure 3.2). As the child vertices of the Sequence ($N_G^+(v)$) are connected to each other, there has to be a second edge (e) for this vertex. It is a directed outgoing edge ending at the vertex representing the next element in the list of child nodes of the Sequence ($\alpha(e) \in N_G^+ \wedge \omega(e) \in N_G^+$) (see (2) in figure 3.2). The only exception is a Sequence having only one child node (w). In that case the second edge (f) connects the child node with the Sequence ($\alpha(f) = w \wedge \omega(f) = v$).
- The element is in between the first and the last element of the list (*cf. A.3.2*): These vertices are each connected to the vertex representing the next element of the list by a directed outgoing edge ($\alpha(e) \in N_G^+ \wedge \omega(e) \in N_G^+$).
- The element is the last one of the list (*cf. A.3.1*): The last vertex has a directed outgoing edge back to the vertex representing the Sequence ($\alpha(f) \in N_G^+ \wedge \omega(f) = v$). (see (3) in figure 3.2).

3.2. Classification

The graph is classified using the rules defined in the section before. The original LAUTS structure is an n ary tree which makes sense as it represents a hierarchic structure with single-parent elements. Furthermore, there is one node (r) which has no parent node ($g_G^-(r) = 0$)[2]. This vertex is called root and makes the graph a rooted tree. Unfortunately, the graph to draw is different from the LAUTS structure it represents

[2]The tree is seen as a directed graph to be able to refer to edges leading to a lower depth as *incoming edges* and edges leading to a higher depth as *outgoing edges*.

(*see the Style Guide in Appendix A*). In fact the problem exists only with one single Control-Flow Operator, the Sequence.

3.2.1. Special Case Sequence

The Sequence is different compared to the other elements as it needs an additional piece of information - the order of the contained elements. Looking at the structure, it becomes clear that the Sequence is really the only element of which the sequence of its child elements is important.

LAUTS Elements have only one child either and with only one element there naturally is no sequence. Random, RandomPutBack and RandomAll represent disjunction (coll. "or") relationships which are not order-sensitive. (Concerning propositional logic: $(A \vee B)$ is exactly the same as $(B \vee A)$.

Of course it would be possible to realize the Sequence with weighted edges (e.g. each edge gets a number which tells the position of the vertex in the sequence. However, such an approach would not be user friendly. This is why it is defined in another way in the Style Guide (*see Appendix A Tree Diagram Style Guide*) of the tree diagram (*see figure 4.2 on page 24*). Because of this, the sequence is shown directly in the graph visualized via vectors between the subordinated vertices of the Sequence-node.

From this it follows that the graph could not be a tree anymore but is a simple general directed graph. Further considerations on how to convert the structure to a tree again are made in chapter 4.

3.2.2. Conclusion of Classification

Despite the fact that the graph to draw seems to be a *rooted tree* on the first look it turns out that, it is a *general directed graph* because of the special case of Sequence. In spite of that the graph could be drawn as an at least tree-like structure.

3.3. Planarity

According to the major publications on graph drawing (e.g. [12],[4],[6]), the minimization of edge crossings is defined as an aesthetic criterion of high importance. The ideal case would be a drawing with no edge crossings at all. A graph which can be drawn without edge crossings is called planar (as stated in the definition section). A tree is always planar (see [9] for a proof). Now the question arises whether the tree-like graph considered here is also planar.

Referring Kuratowski's theorem a graph is planar if it does not contain one of the following subgraphs:

- The complete graph of five vertices (*see figure 3.3* K_5).
- The complete bipartite graph with six vertices where three vertices per partition (*see figure 3.3* $K_{3,3}$).

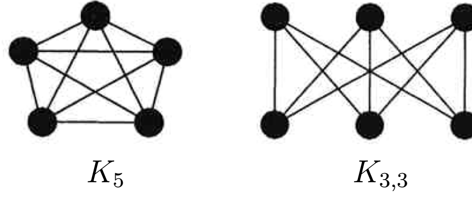


Figure 3.3.: Kuratowski graphs.

As the graph without Sequences is a tree (which is planar as well), only Sequence has to be analysed for planarity. Even if one or more of the cycle sub vertices contain further elements (e.g. vertex 3 in figure 3.4(a)), it would not have any influence on the planarity considerations. This is because any additional incident edge is an inter-layer edge, defined by the LAUTS structure (see *chapter 3.1*). This fact also means that vertex 1 of the figure does not have any meaning concerning the planarity of the Sequence. Due to this reasons only the subgraph $G' \subset G$ containing vertices 2 to 6 and the edges between those vertices have to be analysed. G' is shown in figure 3.4(b).

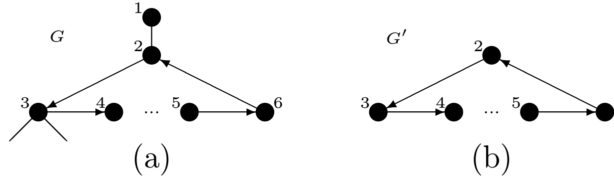


Figure 3.4.: Proofing planarity of Sequence.

With these conditions it is clear that each tree like graph G is planar. Without the Sequence the graph is planar as well. A Sequence represented by the graph $G' \subset G$ is planar because:

$$\begin{aligned} & g(v) = 2 \forall v \in V(G') \\ \wedge \quad & V(G') \ = \ V(P) \\ \wedge \quad & \exists P | \alpha(P) = v \wedge \omega(P) = v \text{ with } n(P) > 1 \end{aligned} \tag{3.1}$$

This means that each vertex has a degree of two (the incoming and the outgoing edge) and there exists a path P which starts and ends at the same point (cycle) and has the same set of vertices as G' which has to contain more than one vertex.

This ensures not only that the Sequence is a cycle but also that it is possible to draw it planar. That is because with these constraints there is no possibility that K_5 or $K_{3,3}$ may occur. This proofs that each possible graph of LAUTS is a planar graph.

4. Graph Transformations

In the preceding chapter, the graph was classified to be a general directed graph. Unfortunately, not every graph drawing algorithm can handle this class of graphs.

But this does not mean that there is no way of using them. If the necessary graph type is not too different, the graph can be converted temporally to fit the algorithm.

The following techniques are presented in this thesis:

1. **Defect-Draw-Repair:** This method converts a (self-)loop less general graph into a rooted tree (see 4.1).

2. **Cycle Removal:** Converts a general directed loop-less graph into an acyclic graph. This method is proposed by [4] page 23 - 25 (see 4.2).

3. **Cycle Substitution:** Converts a general directed loop-less graph into a tree (see 4.3).

4. **Bi-Treeing:** Converts an n ary tree into a binary tree. (see 4.4).

The principle of operation is the same for all transformations. In the first step the graph is *transformed*. Then the graph is drawn by any graph drawing algorithm and after that the Transformer is again in charge to *correct* the graph. Transformation and correction are not two separate tasks but they are connected. For example the information about removed or added edges in the transformation step are used during the correction phase. Figure 4.1 illustrates the operating principle of the Transformers.

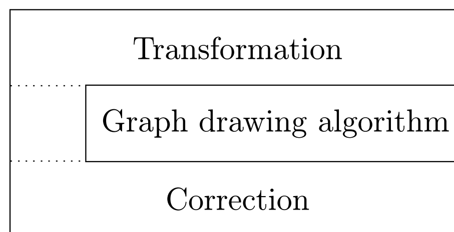


Figure 4.1.: Operating principle of the Transformers.

4.1. Defect-Draw-Repair

To convert a general graph without (self-)loops into a tree, the circuits have to be removed. This can be done by the following three-step process:

1. Place a defect to generate a tree (transformation).
2. Do graph drawing.
3. Repair the defects (correction).

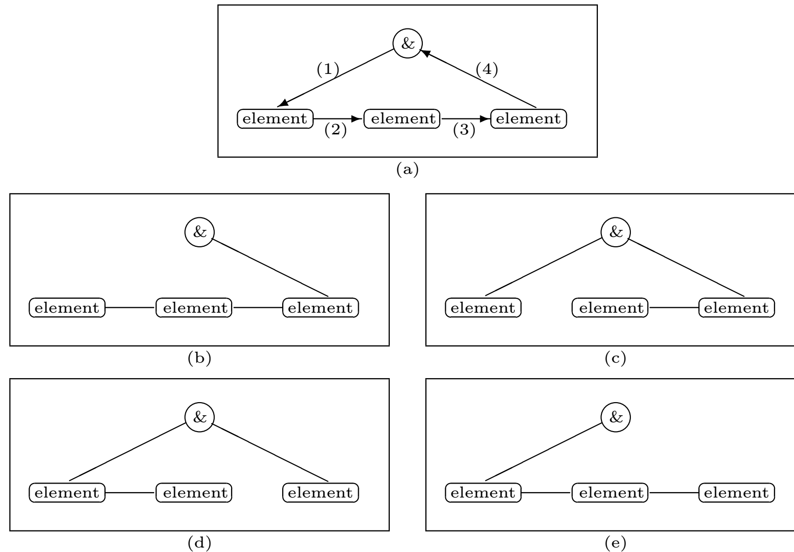


Figure 4.2.: Edges that can be removed by Defect-Draw-Repair method.

Transformation: In this case, a defect is a missing edge. Removing a single edge of the cycle will solve the problem as shown in Figure 4.2. For graph theoretic reasons, it does not matter which edge is removed. But for readability reasons the last edge is removed ((4) in figure 4.2 (a)). This has two reasons. The first one is that ideally all edges should point in the same direction, which is downward here. Second, sub nodes are in the same subtree of the Sequence. Thereby, the former loop becomes a partial tree (or in case of the figure, if no child element of the Sequence has children of its own, it will be a branch).

Drawing: Drawing done by the graph drawing algorithm (see chapter 6).

Correction: To repair the defect, the removed edge just has to be added to the graph. Unfortunately, this may produce edge crossings. That means even if the graph was drawn planar by the graph drawing algorithm there might be edge crossings after the correction step of the Transformer.

4.2. Cycle Removal

This technique converts a general (self-)loop-less directed graph into an acyclic one. The idea proposed in [4] page 23 - 25 is simple: Invert the direction of one edge of a circuit and do so as long as there are any circuits. For reasons of optimisation, the set of

inverted edges should be kept as small as possible, as depicted in figure Figure 4.3. The problem of which edge to invert is the same as it was for Defect-Draw-Repair. Any edge of a cycle will solve the problem. The solution is again to invert the edge from the last cycle sub node back to the cycle root (edge (4) in figure 4.2(a)).

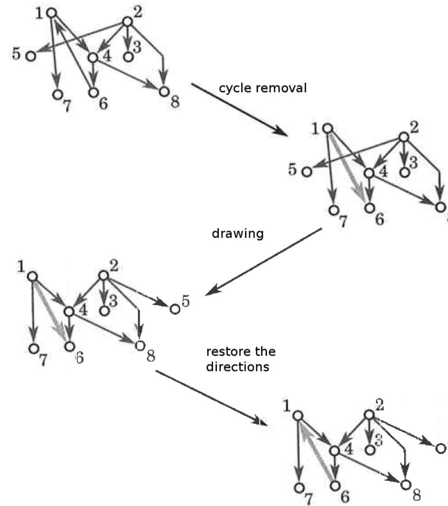


Figure 4.3.: Conversion of a graph with Cycle Removal method. Modification from [4] page 24.

4.3. Cycle Substitution

In contrast to the other Transformers discussed above, this one has the advantage that it affects the drawing algorithms which use layering so that the cycle sub vertices are not placed on different layers by default. In an ideal case, all vertices will be placed on the same layer. Furthermore, this Transformer keeps the order of the nodes. These two attributes make this Transformer more compliant with the style guide than the methods proposed above.

Like all Transformers, this one also is a three step process:

- Substitute the cycles (transformation)
- Draw the graph
- Reconstruct the cycles (correction)

4.3.1. Additional Definitions

While both preceding methods change only one of the edges of the cycle, the Cycle Substitution changes all edges except one. Some definitions have to be made for the

cycle C of G:

$$\begin{aligned} (i) &\quad C \sqsubseteq G | \alpha C = \omega C \\ (ii) &\quad s \in V(G) \wedge g^{-}_{G}(s) = 0 \\ (iii) &\quad O = \{P \sqsubseteq G | \alpha P = s \wedge \omega P \in V(C)\} \\ (iv) &\quad c = \omega Q \text{ with } Q \in O | min\{|V(Q)|\} \end{aligned} \tag{4.1}$$

Formula 4.1 defines that the cycle root (c) is the ending vertex of the shortest path (Q) out of the set of paths between the graphs root (s) and the vertices of the cycle (C). In other words, the vertex which is closest to the graphs root is called *cycle root*.

A second definition has to be made for *cycle vertices* (v) and the set *cycle sub vertices* (F):

$$v \in V(C) \tag{4.2}$$

$$F(C) \subset V(C) \backslash \{c\} \tag{4.3}$$

So the term *cycle vertices* refers to all vertices of the cycle while *cycle sub vertices* refers to all vertices except the *cycle root*.

4.3.2. The Algorithm of Cycle Substitution

In Cycle Substitution, the cycle is converted to a rooted subtree with a depth of one. The cycle root will be the root of the subtree and the cycle sub vertices are the successors.

1. **Transformation:** Remove all edges (D) of the cycle (C) which do not start at the cycle root (c) and replace them by edges (A) connecting the cycle root with each of its successors directly:

$$D = \{e \in R(C) | \alpha(e) \neq c\} \tag{4.4}$$

$$W = \{v \in V(C) \backslash \{c, \omega(s)\} \mid \alpha(s) = c\} \text{ with } s \in R(C) \tag{4.5}$$

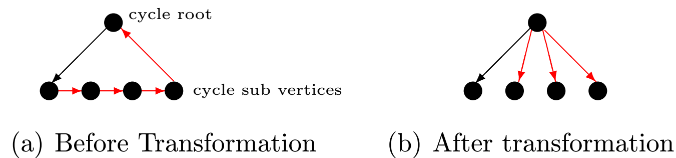
$$A : \exists r \in A \quad | \quad \alpha(r) = c \ \wedge \ \omega(r) = x \ \forall x \in W$$

   Set D contains all edges of the cycle except one. That means $n(D) = n(R(C)) - 1$. This is because there is only one edge which has its starting point at the cycle root by default. The proof of this is given by the definition of a cycle. A cycle is a path (P) with $\alpha P = \omega P$. As $g^{+}_{P}(v) = 1 \forall v \in P$ there can only be one edge starting at the cycle root which is part of the path.

   Set W contains all cycle sub vertices except the vertex already connected to the root. The cardinal number of A is two fewer than the one of D. ($n(A) = n(D) - 2$). This is because the graph is still connected which means one connection per cycle sub vertex which reduces the cardinal number by one. And the edge (s) with $\alpha(s) = c$ is not in A as it already exists and has not to be added.

It is important that the sequence of cycle sub nodes stays the same during the whole process. Otherwise edge crossings will occur during the correction step.

Figure 4.4(a) shows the cycle before the transformation with all edges of D colored red. Illustration 4.4(b) shows the former cycle after transformation with all edges of A colored red.



(a) Before Transformation (b) After transformation

Figure 4.4.: Process of Cycle Substitution.

2. **Drawing:** The graph is drawn by a graph drawing algorithm.

3. **Correction:** After the drawing is done, the transformation is done the other way around to correct the graph. The edges of set A are removed and replaced by those of set D. Again it is really important that the sequence of the cycle sub vertices is kept the same to avoid edge crossings.

Looking at these facts, it becomes clear that the price for better matching the rules of style concerning the layout of Sequence is complexity. While the other transformations only have to change one edge two times[1] independently from the number of cycle vertices, this Transformer has to do $2 \cdot n(F) - 1$ changes. This does not only mean that the effort is higher but it even grows linearly dependent on the number of cycle sub vertices instead of being constant as with the other methods. As the algorithm only depends on the number of cycle vertices, the complexity can be defined as linear: $O(n)$ and $\Omega(n)$.

4.4. Bi-Treeing

There are many elegant drawing algorithms available for binary trees (bi-trees). The problem with those algorithms is simply that they only work for bi-trees. This section describes a method for drawing an n ary tree using bi-tree algorithms. Section 4.1 already described an approach to generate a general (n ary) tree out of a general graph without self-loops. Combining those two techniques it would even be possible to draw general (self-)loop-less graphs with bi-tree drawing algorithms.

The idea behind this procedure is to temporarily reduce the outer degree of the vertices by adding nodes to the graph. Those will be called *dummy nodes*. Three steps are necessary:

[1]Two times because it has to be inverted/removed and after the drawing added/inverted again.

1. **Augmentation:** Generate a bi-tree out of the n ary tree by adding (dummy) nodes and connections (transformation).

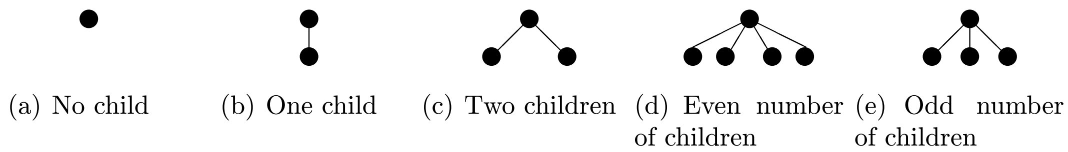
2. **Drawing:** Apply the graph drawing algorithm.

3. **Reverse Conversion:** Generate the drawing back to an n ary tree (correction).

4.4.1. Augmentation

A vertex belonging to a binary tree has at maximum two child nodes ($g_G^+ \leq 2$). All nodes have to fulfill this constraint. This means each node of the tree has to be checked, which is done by a top-down approach.

For each node v of the graph G, the cases shown in figure 4.5 are possible:



(a) No child (b) One child (c) Two children (d) Even number of children (e) Odd number of children

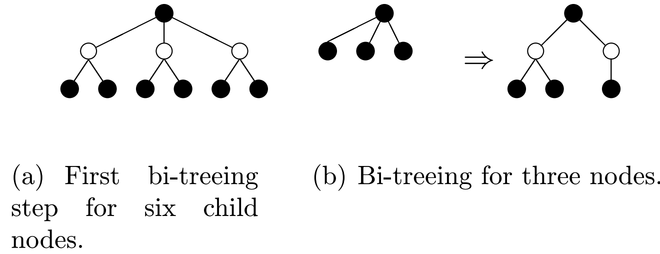
Figure 4.5.: Cases to distinguish for bi-tree conversion.

In case (a), (b) and (c) nothing has to be done because the nodes already match with the constraints. But cases (d) and (e) are not consistent. In (d), dummy nodes have to be introduced, each of them with two child nodes. This is shown in figure 4.6(a). It is easy to see this is not yet a bi-tree conform node. Therefore, the procedure has to be repeated for v until it is conform. Figure 4.6(a) also shows that even if the number of child nodes of v is even this does not mean that the number of introduced dummy nodes is even, too. This is only the case if the number of child nodes $n(N_G^+(v))$ is in 2^i with $i \in \mathbb{N}, i > 1$.

Picture (e) of figure 4.5 shows the case that $n(N_G^+(v))$ is odd. The solution is simple: the dummy node added last has only one child node. Figure 4.6(b) shows an example of this case. The alternative would be not to add a dummy node for the single vertex. But the problem coming up with this approach is that the (real) child nodes of v appear on different layers. This is something which is not desired as it reduces the readability of the graph. Unfortunately, this happens if v has more than four child nodes because at least one additional dummy node layer has to be introduced. The effect can be seen in figure 4.6(c).

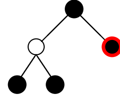
4.4.2. Drawing

After the conversion step, the graph drawing algorithm is applied. As there are additional vertices, the resulting drawing is not the one representing the original graph. Therefore, the reverse conversion step is necessary.



(a) First bi-treeing step for six child nodes.

(b) Bi-treeing for three nodes.



(c) Not inserting a dummy node for the remaining node.

Figure 4.6.: Process of bi-treeing.

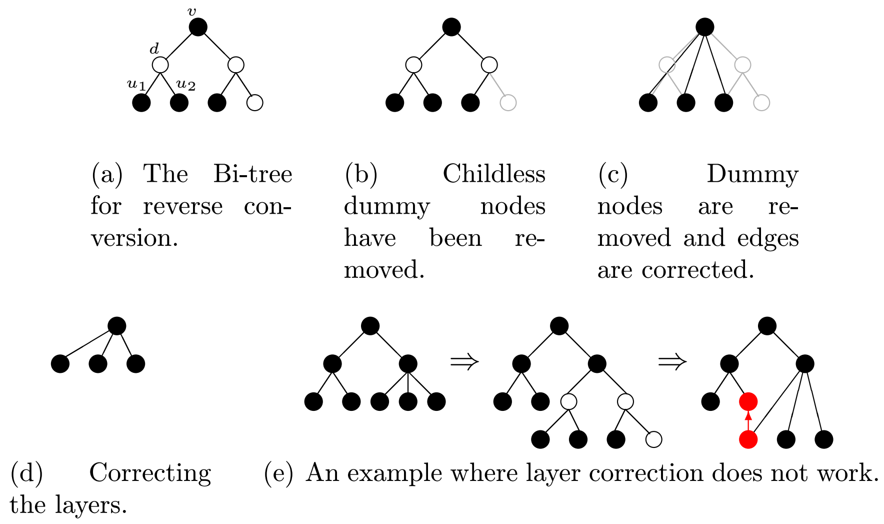
4.4.3. Reverse Conversion

After the drawing is finished, the dummy nodes have to be removed. In contrast to the conversion step, this time a bottom-up approach is used. A distinction has to be made if the dummy node is a leaf or not. If the node is a leaf, it is removed including the adjacent edge. If it is not, the Reverse Conversion is a little more complex. First of all, the edges have to be corrected. Figures 4.7(a) to (c) show this process. Let v be the parent node, u_n the (original) child nodes of v and d, the dummy node. To remove d the following steps have to be taken[2]:

1. Deletion of d and the edges $(v, d), (d, u_1), (d, u_2)$.
2. Insertion of the edges (v, u_1) and (v, u_2).

After correcting the edges, the layers used by the dummy nodes during the drawing are empty and expand the area needed for the graph. Sometimes it is possible to readjust the layering of the nodes as shown in figure 4.7(d). But considering another example shown in 4.7(e), it becomes clear that this technique works only for special cases because vertices which should be moved upward may overlap. Therefore this technique is not taken into the bi-treeing approach by default. However it might be used if possible.

[2]In cases where one of the child nodes of v has been deleted, the procedure is almost the same except the fact that already deleted edges and nodes of course do not have to be considered anymore.



(a) The Bi-tree for reverse conversion.

(b) Childless dummy nodes have been removed.

(c) Dummy nodes are removed and edges are corrected.

(d) Correcting the layers.

(e) An example where layer correction does not work.

Figure 4.7.: Reverse Conversion of the bi-tree.

5. Prearrangements for Rating Algorithms

This chapter contains the prearrangements which are needed to tests and evaluate algorithms. First of all, the aspects rated have to be defined. Secondly, the common rules for the subjects under test are set up. Thirdly, the necessary characteristics of the subjects under test are defined. Finally, the set of graphs to test is specified.

5.1. Rating of Graph Drawing Algorithms

The analysis is done concerning four topics:

1. Quality of the drawing.
2. Maximum complexity of the algorithm (O).
3. Minimum complexity of the algorithm (Ω).

Measuring the complexity of an algorithm is comparatively easy as this is done by a theoretic analysis of the algorithm. When it comes to the quality of the drawing, however, an evaluation is not that easy. As the quality of different algorithms has to be compared, there has to be a measurement which delivers numbers representing the quality. These numbers define what is good and bad quality in a graph drawing; in other words whether a graph looks ”nice”.

To calculate the quality of a graph drawing, the following attributes are considered:

- Number of edge crossings.
- Size of the area needed for the drawing.
- Total edge length.
- Difference between the longest and shortest edge adjacent to one vertex.
- Difference between the longest and shortest edge of the whole graph.
- Minimum clearance between two vertices in the graph (gap).

All rated algorithms have to draw the same graphs to achieve a meaningful result. This is because different graphs may vary in their complexity. Furthermore, one graph might be easy to draw for an algorithm A and hard for B and for a different graph it might be the other way around. To avoid such cases, one algorithm has to draw many graphs before being evaluated.

5.2. Arrangements for the Test Objects

To achieve meaningful results, three arrangements are made concerning the subjects under test:

- Many test objects (graphs in this case).
- Carefully chosen test objects.
- Using the same test objects for all the algorithms.

Many test objects Using only one or very few test objects, the result is under no circumstances representative. The algorithm may coincidentally perform very well for one test object while it performs badly for most other cases as this object may be an ideal case for that algorithm. To relativise such cases, many different test objects are tested. Thus, it is very unlikely that the tested algorithm performs well (or badly) by chance only for the tested cases.

Carefully chosen test objects The test objects have to be chosen carefully as it makes no sense if the same constellation of nodes is tested again and again. The different objects should cover as many different cases as possible. The problem is to find those cases that affect a difference in the behavior of the algorithms. This is not easy as it requires extensive knowledge of the workings of the algorithms. An even harder problem is the fact that there are many different algorithms which could be tested. It would not be possible to examine all of them and create test objects based on all algorithms. Therefore, the graph structure is considered, recurring components are identified and a large number of combinations of these components will be used to assemble the test objects to get representative results.

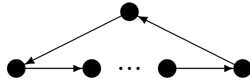
Using the same test objects for all algorithms Like for all (non-destructive) measurements the test objects have to stay the same for all algorithms tested. Changing the objects between testing the algorithms would make it impossible to to draw significant conclusions in the end.

5.3. Characteristics of the Graphs used as Test Objects

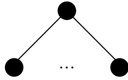
5.3.1. Shape of the Graph

Looking at the LAUTS structure, it becomes clear that on an abstract level only three components have to be considered. In this context, “abstract” means that except for the root vertex of the graph the type of the node is not considered. This means all vertices are equal. In contrast, the edges will be considered as they play a role for the type of the graph. The components are:

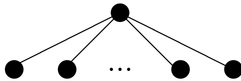
- **Sequence**



- **Decision**



- **Others**



Each graph consists of those three components. The set of graphs that are to be tested can be divided into two subsets: the smaller one consisting purely of one of the components and the bigger one containing graphs consisting of a mixture of the components. Pure graphs will point out if an algorithm has problems with one of the components while the mixed graphs present a more realistic scenario as a graph with only one component is very unlikely to occur in reality.

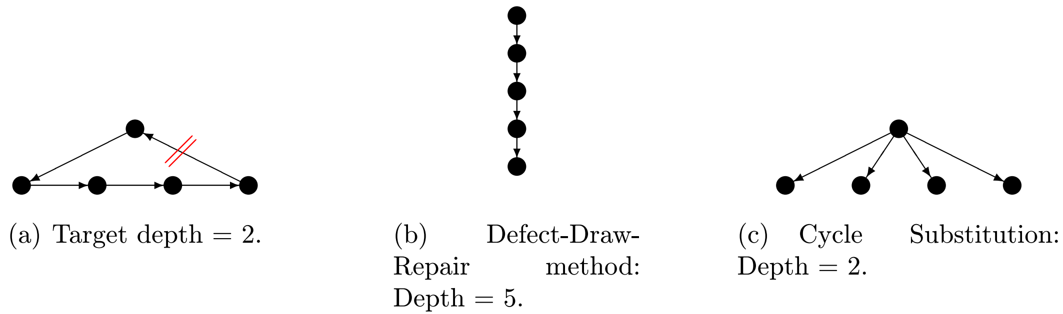
5.3.2. Size of the Graph

In addition to the shape of the graph, its size affects the performance of graph drawing algorithms. Therefore, graphs of different sizes will be selected for testing. Before doing so, however, one question has to be clarified: "What is the size of a graph ?"

Size is not a single value but it is a combination of:

- Number of nodes ($n(G) := |V(G)|$)
- Number of edges ($m(G) := |R(G)|$)
- Depth of the spanning tree of the graph ($max(n(P_{(G)}))$)

The first two items are simple. However, the third one should be considered in more detail. As defined in section 3.2, the LAUTS graph is a general directed graph. The problem with this type of graph is that there might be several spanning trees with different depths. In the same section, it is pointed out that the LAUTS graph is not a tree due to the Sequence. This means that the Sequence-Nodes have to be changed during construction of the spanning tree. This can be done with the Defect-Draw-Repair method described in 4.1 or the Cycle Substitution specified in 4.3. The problem which arises if the Sequence has more than one child node is that it has an unintended influence on the depth of the tree. Depending on the employed Transformer, the depth of the spanning tree will be different (as illustrated in figure 5.1).



(a) Target depth = 2.

(b) Defect-Draw-Repair method: Depth = 5.

(c) Cycle Substitution: Depth = 2.

Figure 5.1.: Influence of a Sequence to tree-depth.

Just removing the vertices in between the first and the last child of the Sequence is no solution either because these nodes may contain child nodes themselves which have to be considered when measuring the depth of the graph.

This means if the measurement of tree depth should be done automatically based on graph theory, the Sequence has to be transformed back to the style it had in the LAUTS structure. This is done by Cycle Substitution. This change is only done for the measurement and not permanently. Only thus can the depth be measured correctly.

5.4. Set of Graphs to Test

As the graphs need very much space, they can be found at least partly in *Appendix C* and completely on the CD. As mentioned before, the graphs are divided into pure component graphs and mixed component graphs.

Pure Component Graphs: As Sequence and Decision are Control-Flow operators, they may neither be the root nor one of the leafs of the graph. Due to this, a pure component graph for these two components has to have LAUTS Elements as root and leaf nodes.

For each of the components, there are two graphs of different sizes built:

- A minimal graph containing only one of the components (six nodes)
- A big graph containing a lot of components (roughly 10,000 nodes)

Mixed Component Graphs: As graphs containing only one of the components have already been covered, the mixed component graphs contain a mixture of all three components. Except for one special case, all graphs are randomly created.

This special case is that the graph is an acyclic path. This means each element of the graph contains exactly one other element except the last one. For the Control-Flow Operators only Random, RandomPutBack and RandomAll can be used. This is

because Sequence is not possible as the path is acyclic and Decision has to have exactly two following nodes.

For mixed component graphs, two different sizes are considered:

- A graph with about 100 nodes.
- A graph with roughly 10,000 nodes.

To avoid using a test object which represents a special case of an algorithm, eight different mixed component graphs are used (in addition to the two graphs for the special case). The difference made during the creation of those eight graphs is the maximum of the outer degree of the vertices. The eight graphs are divided into four pairs. Each pair has the same maximal outer degree. In one pair there is one graph with about 500 and one with about 10,000 vertices. The following maximal outer degrees are used: 3, 5, 20, 50. This makes 16 graphs altogether. For a detailed description, see *Appendix B*.

6. Graph Drawing Algorithms

6.1. Leaf-First-Layering

6.1.1. Overview

The *Leaf-First-Layering* algorithm is a very simple algorithm which uses a bottom up approach to draw rooted trees. It produces layered orthogonal poly-line drawings. Furthermore, it guarantees the minimal distance between two vertices and creates a very compact drawing concerning space. The time complexity is linear.

6.1.2. Mode of Operation

The drawing is a three-step process:

- Preprocessing
- Vertex placement
- Edge orthogonalization

Two constants have to be defined in advance for the algorithm:

- **Vertex separation:** Defines the minimal horizontal spacing between two nodes on the same layer (set to 14 for the tests).
- **Layer separation:** Defines the vertical spacing between two neighbouring layers (set to 14 for the tests).

1. **Preprocessing:** The preprocessing step has three major tasks:

   a) Order the leaving edges of each vertex
   b) Identify leafs of the graph
   c) Identify parent of each vertex
   d) Set the depth of each node

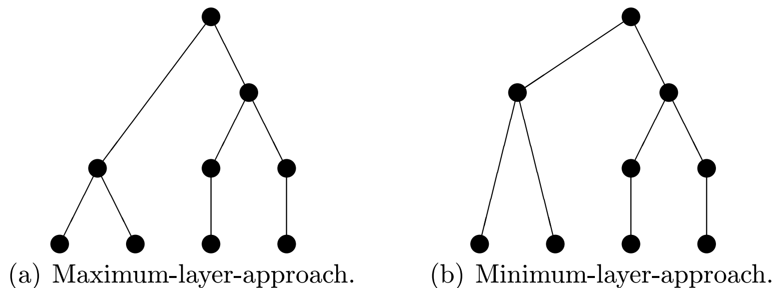
   a) First of all the children of each vertex have to be ordered counterclockwise. Therefore each vertex v contains a list of its child nodes $out(v)$ which contains the end-nodes of its leaving edges in that order. This means if v is on layer l and the vertices following v ($w_0..w_k$) are on layer $l+1$, w_0 will be the leftmost

and w_k the rightmost vertex. In other words, the order is from left to right. This is done by a *depth first* strategy. Looking at the LAUTS structure, this step is important to avoid edge crossing inside a Sequence. Fortunately, the Transformers (described in chapter 4) already take care of that and the resulting order is counterclockwise. This fact makes this step unnecessary, at least for the described Transformers. Due to this fact, the algorithm described and shown as pseudo-code below is adjusted as it leaves this step out.

b) The identification of the leafs can be done on the same run as the ordering. The leafs are stored in a list (*leafList*). It contains all the leaf nodes ordered in counterclockwise sequence.

c) Identifying the parent of each node is also done during the run ordering the vertices. The parent of v is addressed as $parent(v)$. As by definition of a rooted tree, each vertex has exactly one parent except the root which has no parent.

d) The last information retrieved in preprocessing step is the layer of each vertex. This can also be done at the same run. There is a list for each layer which contains the nodes belonging to it. A layer list is addressed by $layerList(x)$, where x is the number of the layer. It starts at 0 (for the root) and has a maximum value of $depth - 1$ where $depth$ is the depth of the tree. The process begins at the leaf nodes and is done upwards to the root (bottom up). This has an important influence on the resulting drawing as it defines if the layer of a vertex is maximal or minimal. The algorithm measured in this thesis uses the maximal-layer approach. The result in the drawing is that the inner nodes are as close to the leaf-layer as possible. Figure 6.1 (a) shows a draft for the results of a maximum-layer- and (b) the one for a minimum-layer-approach.



(a) Maximum-layer-approach. (b) Minimum-layer-approach.

Figure 6.1.: Minimum- and maximum-layer-approach for Leaf-First-Layering.

```
depth = 0;
procedure preprocessing(v:vertex,layer:int)
```

```
begin
  if(depth <= layer)    // Find out max depth
    then depth+1;

  if(out(v).length=0) // if v is a leaf
  then leafList + v;     // add it to leafList
  else                // if v is an inner node
    layerList(layer)+v;   // assign the  vertex to the layerList
    foreach(u of out(v))  // call perprocessing recursively ...
      preprocessing(u,layer+1);  // for all child nodes
end;
```

2. **Vertex placement:** The placement of vertices is done in two steps:

   a) Leaf placement
   b) Inner node placement

   The steps in detail:

   a) The first step places all the leaf nodes to the lowest layer ($depth - 1$). As they are ordered from left to right, they are simply placed one after the other separated by $vertexSeparation$ in x-direction. The y-value of each vertex is defined by the layer it belongs to. As this is the lowest one: $y = (depth - 1) * layerSeparation$. The lowest layer is $depth - 1$ because depth starts at 1 while layers start at 0.
   b) In a second step, all inner nodes of the tree are placed. This is done by the depth first approach too, which means that the placement is done bottom up. Each node $v \in V(G)$ is placed at the center of its child nodes. For the calculation of the center only the outer child nodes have to be considered. The x-coordinate is: $x(v) = w_0(v) + \frac{x(w_k(v))-x(w_0(v))}{2}$ with k = number of sub nodes of v minus one. The y-coordinate is defined by the layer of v: $y(v) = layer(v) * layerSeparation$.

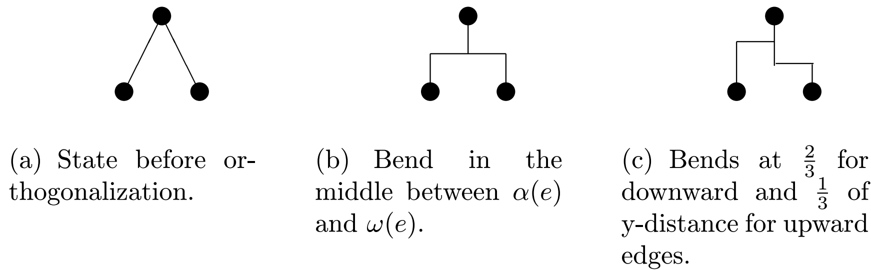
```
layerSeparation = 14;
nodeSeparation = 14;
procedure layout()
begin
  i = 0;

  // Place leaf nodes
  foreach(v of leafList)
    y(v)=depth*layerSeparation;
    x(v)=i*nodeSeparation;
    i++;
```

```
  // Place inner nodes
  foreach(i from depth-2 to 0)
    foreach(v of layerList(i))
      y(v)=i*layerSeparation;
      x(v)=x(out(v).first)+(x(out(v).first-x(out(v).last))/2;
  i--;
end;
```

3. **Orthogonalize:** This last step converts the drawing from a single-line to a polyline drawing. To be more precise, the drawing is converted to an orthogonal polyline drawing as the edges are right-angled to each other. Therefore, each edge (e) of the graph is considered and two bends are added per edge. The list of all edges of the graph is referred to as *list*. The x-coordinate of both bends is simple as it is $x(\alpha(e))$ for the first and $x(\omega(e))$ for the second one.

   Since this step is independent from the graph type, it considers edges with: $y(\alpha(e)) < y(\omega(e))$ (downward edges) as well as edges of the other direction: $y(\alpha(e)) > y(\omega(e))$ (upward edges). The y-coordinates for both bends are equal by definition as it has to be orthogonal. But if y is chosen exactly in the middle between $\alpha(e)$ and $\omega(e)$, it would not be visible if an edge is moving upward or downward (see figure 6.2(b)).



(a) State before orthogonalization.

(b) Bend in the middle between $\alpha(e)$ and $\omega(e)$.

(c) Bends at $\frac{2}{3}$ for downward and $\frac{1}{3}$ of y-distance for upward edges.

Figure 6.2.: Placement of bends in upward and downward edges for Leaf-First-Layering.

```
procedure orthogonalize(list:connectionList)
begin
  foreach(c:connection of list)
    if(y(c.begin) < y(c.end))
      x(bend1) = x(c.begin);
      y(bend1) = y(c.begin) + layerSeparation*2/3;
      x(bend2) = x(c.end);
      y(bend2) = y(c.begin) + layerSeparation*2/3;
    else
      x(bend1) = x(c.begin);
```

```
        y(bend1) = y(c.end) + layerSeparation/3;
        x(bend2) = x(c.end);
        y(bend2) = y(c.end) + layerSeparation/3;
end;
```

6.1.3. Complexity

The preprocessing step is executed one time and considers each vertex exactly once which produces a complexity of $O(n)$ and Ω(n).

This is true for the second step, the vertex placement, also. The sequence in which the vertices are treated is different but still each vertex is considered once. Again the complexity is $O(n)$ and Ω(n).

The orthogonalization needs some more consideration. This is because here the connections are handled. For a (spanning) tree, this would be easy as each vertex except the root of a tree has an outer degree of one (6.2). The root has an outer degree of 0 (see formula 6.1). As loops and cycles are not possible in a tree this means that the number of edges (m) is the number of vertices (n) minus one (the root) (6.5).

$$|r \in V \text{ with } g_G^+ = 0| = 1 \quad (6.1)$$
$$g_G^+(v) = 1 \forall v \in V \backslash r \quad (6.2)$$
$$W = \{v \in V | g_G^+(v) = 1\} \quad (6.3)$$
$$n = |V| \quad (6.4)$$
$$\Rightarrow m = |W| = n - 1 \quad (6.5)$$

Unfortunately, int this case, the graph is not a tree anymore as all connections (including those removed by the Transformer) have to be orthogonalized. The worst case to be considered is the fully connected graph which means that every vertex is connected to every other vertex. This results in a number of edges (m) against vertices(n) from $m = (n-1) \cdot \frac{n}{2}$ which is in fact the progression: 0,1,3,6,10,15,21,28,... . In consequence, the complexity is $O((n-1)\frac{n}{2})$. Ω(n) is true for the minimum complexity as the ideal case would be a tree. This is the case if the LAUTS structure does not contain any Sequences.

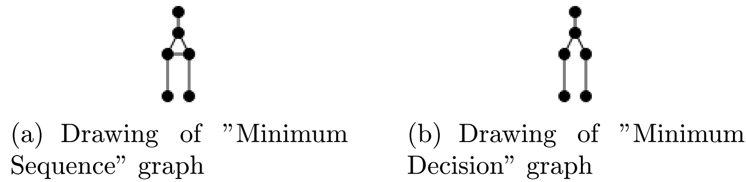
In summary, the complexity results in:

$$O \quad (2n + (n-1)\frac{n}{2}) \quad (6.6)$$
$$\Omega \quad (3n)$$

6.1.4. Results

Due to space limitations, only Minimum Sequence and Minimum Decision are presented here as an image to illustrate the results of the Leaf-First-Layering algorithm exemplarily.

All the other drawings can be found at least partly in Appendix C.1 and completely on the CD in directory `Drawings/LeafFirstLayering`. Figure 6.3 shows the resulting drawings.



(a) Drawing of "Minimum Sequence" graph

(b) Drawing of "Minimum Decision" graph

Figure 6.3.: Exemplary results of the Leaf-First-Layering algorithm.

6.1.5. Evaluation

	Crossings	Area	Length	Diff @ G	Diff at v	Gap
1_(10139)	1344	73094x252	798225	19370	19370	14
1_(563)	55	4158x168	34165	1535	1311	14
2_(471)	1510	6342x56	116220	3157	3157	14
2_(7311)	16640	98140x70	2204180	45982	45982	14
3_(670)	467	8428x98	69631	3945	3945	14
3_(9551)	9745	119126x98	1792394	57611	57611	14
4_(564)	19	2142x336	24358	618	618	14
4_(9787)	685	36932x504	573347	20478	20478	14
Min Decision	0	14,56	101	14	12	14
Min Other	0	14,56	101	14	12	14
Min Sequence	0	14,56	115	14	14	14
Special(100)	0	0x1344	1456	14	14	14
Special(10000)	0	0x146300	146300	14	14	14
Sequence	0	58226x126	776440	25402	25402	14
Decision	0	150192x294	1728190	46237	46237	14
Other	0	58198x140	552572	15536	15170	14

Table 6.1.: Result of Leaf-First-Layering.

The results of the Leaf-First-Layering algorithm (cp. table 6.1) show that the clearance is always exactly the desired value.

Considering the number of edge crossings, it turns out that it works fine for all pure graphs independently from their size. However, graphs of mixed elements cause edge crossings. The reason for that is definitely not the size of the graph as 4_9787 which has much more vertices but less edge crossings than 2_7311. Comparing the types of elements of those two graphs, 4_9787 has many Control-Flow Operators and few LAUTS

Elements while 2_7311 has mostly LAUTS Elements and only a small amount of Control-Flow Operators. But this alone can not be the explanation for this phenomenon. Two possible explanations exist: Either 2_7311 has more crossings because a LAUTS Element has only one child vertex whilst Control-Flow Operators may have more. Or the reason for the crossings lies in the number of leaf vertices which is directly related to the maximum number of child nodes (maximum outer degree) per vertex.

The first possibility does not seem to be plausible because if the algorithm has a problem with vertices having only one child it would have been visible at the pure-element graphs before.

The edge crossings do also not seem to be a stochastical effect. The number of child nodes of a graph seems to be correlated to the number of crossings. 2_471 has by far the most crossings in the group of graphs with some hundreds of nodes. Similarly 2_7311 has by far the most edge crossings in the group of graphs with some thousands of nodes. Both have a maximum outer degree of 50. The graphs with only a maximum of three (4_564 and 4_9787) have on their side also in both groups by far the smallest number of edge crossings. Summing up, it could be said that the algorithm seems to have problems with graphs containing vertices with a high outer degree.

The area needed for the drawing is relatively low. All the drawings are very wide in comparison to their height. As the required space is smaller compared to the Dominance-Straight-Line algorithm (see 6.2), the total edge length is also several times smaller. A possible reason for that is given in section 6.3.5 on page 55.

6.2. Dominance-Straight-Line

6.2.1. Overview

The Dominance Drawing algorithm proposed by [4] is used for planar digraphs. It creates a drawing with straight line connections, detects and displays symmetries and the geometric description of the transitive closure by the dominance relation between the points (coordinates) and vertices.

The problem concerning directed graphs compared with trees is that one vertex may be reachable by more than one path (see *figure 6.4*). This has to be taken into account in the algorithm traversing the tree. Otherwise, vertices might be considered multiple times instead of only once or the algorithm might run into an infinite loop. For this problem, many solutions are possible, for example simply a list which stores the vertices which were already considered. The algorithm presented here uses a different approach. It only considers the vertex if it was reached by the first incoming connection. This assumes that the connections are in some kind of order. It does not matter which order this is. The only thing that matters is that it does not change during the time the algorithm is working.

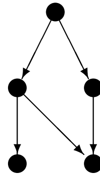


Figure 6.4.: The lower right vertex is reachable via two connections.

6.2.2. Mode of Operation

The algorithm is a three-step process. The first step, *Preprocessing*, transforms the data structure representing the graph into the right format, the second one, *Preliminary Layout*, places the vertices to a distinct point (x- and y- coordinate) and the third, *Compaction*, makes the drawing compact.

1. **Preprocessing:** The data structure for the graph to draw is transformed so that each vertex v has a list of outgoing edges sorted according to their clockwise sequence around v. It is possible to move inside this list by *next(e)* (for next vertex) and *pred(e)* (for preceding vertex). If an invalid position of the list is addressed, the result is *nil*[1]. *firstout(v)* and *lastout(v)* referes to the leftmost respectively rightmost edge leaving v. The other way around *firstin(v)* and *lastin(v)* refer to the first respectively last incoming edge of v. At least one edge $e = (u, v)$ refers to v as *head(e)*.

2. **Preliminary Layout:** In this step coordinates are assigned to each vertex. First, all the x- and then the y-coordinates are determined by incrementing a counter and setting its value as the coordinate value. This is done for x- and y-coordinates separately.

   The following pseudo code, which is similar to the description of the algorithm in [4], chapter 4.7, clarifies this step.

   First of all, the setting of x-coordinates:

```
count = 0;
GRID = 15; // Distance between the coordinates.
procedure LabelX(v:vertex)
begin
  X(v)=0;
  count++;
  e=firstout(v);
```

[1]nil = not in list.

```
  repeat
    w=head(e);
    if(e=lastin(w)) // This is used to not consider one node twice.
      then LabelX(w);
    e=next(e);
  until e=nil;
end;
```

Then the setting of the y-coordinates:

```
count = 0;
GRID = 15; // Distance between the coordinates.
procedure LabelY(v:vertex);
begin
  Y(v)=0;
  count++;
  e = firstout(v);
  repeat
    w = head(e);
    if(e=firstin(w)) // This is used to not consider one node twice.
      then LabelY(w)
    e=pred(e);
  until e=nil;
end;
```

3. **Compaction:** This last step makes the graph compact and assigns the final coordinates to the vertices. This is done according to the following scheme.

   Let u and v be a pair of vertices with consecutive x-coordinates. In case that $\exists e(u, v)$, x is not incremented except $g_G^+(u) = 1 \wedge g_G^-(v) = 1$. In the remaining cases, the x-coordinate is incremented. The assignment of the y-coordinates is done in the same way. The following pseudo code is also similar to the one presented in [4], chapter 4.7.

```
u = the vertex with X(u) = 0;
x(u) = 0;
procedure assignX()
begin
  while nextX(u)!=0
    v = nextX(u);
    if(Y(u)>Y(v) or (firstout(u) = lastout(u) and firstin(v) = lastin(v))
      then x(v) = x(u) + 1;
      else x(v) = x(u);
    u = v;
```

```
end;

u = vertex with Y(u)=0;
y(u) = 0;
while nextY(u) != 0
  v = nextY(u);
  if( X(u) > X(v) or (firstout(u) = lastout(u) and firstin(v)  = lastin(v)
    then y(v) = y(u) + 1;
    else y(v) = y(u);
  u = v;
end;
```

6.2.3. Complexity

The preprocessing step does not contribute to the complexity as the graph is already in a right shape. The calculation of preliminary layout itself is done in two procedures: one for the x and one for the y-coordinates. As the functionality could be seen to be the same the complexity is the same too and will be considered together here. The complexity is simply $O(n)$ and $\Omega(n)$ per procedure because each vertex is addressed exactly once. For the whole preliminary layout step, this makes a complexity of $O(2n)$ and $\Omega(2n)$.

The compaction step is again mostly the same for x- and y-coordinates which allows considering both procedures at once, too. First of all, there have to be two lists of vertices sorted by increasing x- respectively y-coordinates. Therefore a *quick sort* algorithm is used. Its complexity is $O(n^2)$ and $\Omega(n \cdot log(n))$ (cp. [11]). As the sorted lists are traversed exactly once each, the complexity is $O(n)$ and $\Omega(n)$ for each of the compaction step procedures. This means that the compaction step (without the sorting) has $O(2n)$ and $\Omega(2n)$.

In summary, this results in the following complexity:

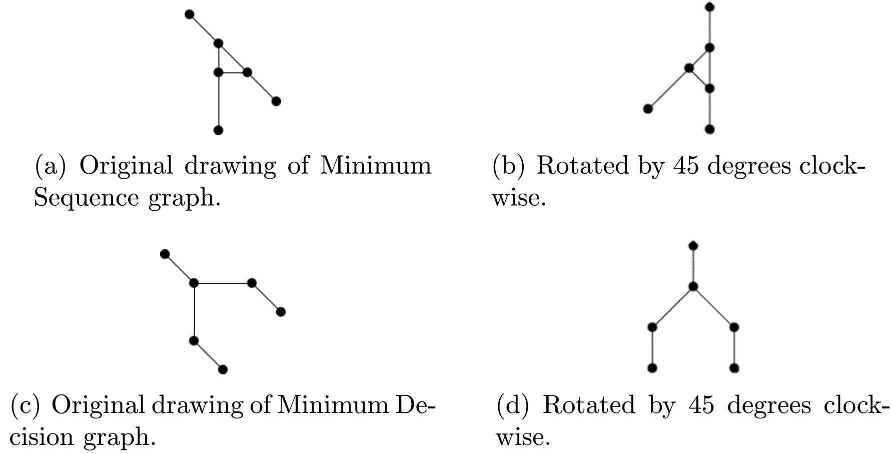
$$\begin{aligned} &O \quad (4n + 2n^2) \\ &\Omega \quad (4n + 2n \cdot log(n)) \end{aligned} \tag{6.7}$$

6.2.4. Results

In this chapter the drawing of the Minimum Sequence-graph is used to exemplary illustrate the result of the Dominance-Straight-Line algorithm. Figure 6.5(a) shows the original outcome of the algorithm. To better match the aesthetic criteria, 6.5(b) shows the outcome rotated clockwise by 45° . The other drawings can be found in *Appendix C.2* and in more detail on the appended CD in directory `Drawings/DominanceStraightLine`.

6.2.5. Evaluation

As clearly can be seen in the table, the algorithm produces planar drawings in all cases. Furthermore, the clearance is absolutely stable at the desired value. Looking at the



(a) Original drawing of Minimum Sequence graph.

(b) Rotated by 45 degrees clockwise.

(c) Original drawing of Minimum Decision graph.

(d) Rotated by 45 degrees clockwise.

Figure 6.5.: Exemplary results of the Dominance-Straight-Line algorithm.

	Crossings	Area	Length	Diff @ G	Diff at v	Gap
1_(10139)	0	74031x74031	1374675	59740	59735	14
1_(563)	0	4101x4101	49168	2740	2736	14
2_(471)	0	4541x4541	232988	4515	4515	14
2_(7311)	0	70241x70241	4924043	66640	66635	14
3_(670)	0	6111x6111	119167	6000	5995	14
3_(9551)	0	86521x86521	3476549	86500	86496	14
4_(564)	0	4101x4101	43551	3410	3406	14
4_(9787)	0	71481x71481	1350468	51638	51638	14
Min Decision	0	41x41	82	6	6	14
Min Other	0	41x41	82	6	6	14
Min Sequence	0	41x41	111	14	14	14
Special100	0	951x951	1457	0	0	14
Special 10000	0	104491x104491	147771	0	0	14
Sequence	0	86111x86111	1461827	82716	82716	14
Decision	0	107291x107291	1965484	75510	75510	14
Other	0	85941x85941	1657962	62566	62566	14

Table 6.2.: Result of the Dominance-Straight-Line.

required area, it turns out that the drawings are all quadratic. Unfortunately, the drawings need several times the space the other arithmetic algorithms need. Even if not the circumscribing rectangle but the area needed for the transitive closure is considered (which is roughly half of the space), the Dominance-Straight-Line algorithm still needs much more space.

The total edge length is also high in comparison to Improved Walker(6.3) and Leaf-First-Layering(6.1). One characteristic visible in all layered drawings is visible here, too.

The length differences between two edges of the whole graph and at one vertex are very tiny or even equal. This effect occurs as the tree gets broader from root to the leafs on the one hand and because superior vertices are placed in the barry center of their child vertices. As all the vertices of one layer are placed on one line, the length of the edges from those vertices to their superior node(s) differs. The differences increase more the closer the layer is to the layer 0 (which contains the root-vertex). Figure 6.6 illustrates that fact. An opportunity to have less edge length differences would be to use centric layers instead of straight lines.

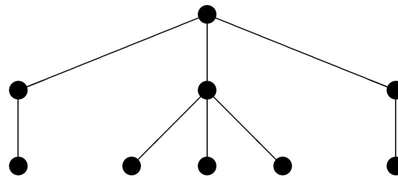


Figure 6.6.: The greatest difference in edge lengths at one node is at connections starting at the root node.

6.3. Improved Walker

6.3.1. Overview

The algorithm called Improved Walker has been improved many times. Its origin is the Wetherell-Shannon algorithm [19] introduced in 1979. It is a linear time algorithm for binary trees. This was improved by Reingold and Tilford [10] in 1981. It still runs in linear time and adds some additional aesthetic features as example symmetry. In 1990, Walker extended this algorithm to handle rooted ordered trees of unbound degree [17]. Unfortunately, this algorithm has a quadratic runtime [2]. The adjustment presented in [2] corrects this problem. Regrettably, it seems that these changes were at the expense of planarity. As the results show, planarity is not given in all cases anymore.

The basic idea is to do the drawing bottom up by a depth-first strategy. After the placement of each node, it is figured out if there is a conflict with the subtree neighbouring to the left and to the right. If so, the subtree is not really moved. Instead, the necessary movement is saved in a variable at the node just placed. Moving the tree at that point would raise the complexity above linear. To figure out if two subtrees intersect only the contours of both are considered instead of comparing each node of both trees which would raise the complexity to $O(n^2)$. How this works in detail is described in the next section.

6.3.2. Mode of Operation

The algorithm works in three steps. First of all, some preprocessing initializing the variables has to be done. Then the two active steps called *walks* are executed. The *first*

walk calculates preliminary coordinates and the *second walk* produces the final ones. Therefore, more information has to be stored per vertex. In addition to the x- and y-coordinate, these are:

- Modifier referenced as *mod(v)*: Stores one part of the number of pixels (*modsum*). The node including its subtree has to be moved in the second walk. The value *mod* at one node is only one part of the real number of pixels the node has moved. The real number is the sum of all superior vertices' *mod*-value added to the *mod*-values of the actual node:

  Let P be the path from the root to vertex to move (v_k) with the vertices of P numbered from v_0 as root to v_k with $k = n(P) - 1$;

  $modsum = \sum_{k=0}^{n(P)-1} mod(v_k)$

- Preliminary coordinate as *pre(v)*: Stores the preliminary coordinates of a vertex.

- A pointer to a preceding vertex called *ancestor*: This variable is a pointer to one superior vertex ($ancestor \in P$). This is needed to memorize superior vertices while traversing the subtree of *ancestor*. Without remembering those vertices, the algorithm would have to track back which means that the complexity becomes higher.

- A second pointer to a vertex called *thread*: This variable is one of the most important ones for keeping the complexity linear. It connects the vertices belonging to the contour of the subtree even if they reside in different subtrees. Figure 6.8 visualizes this.

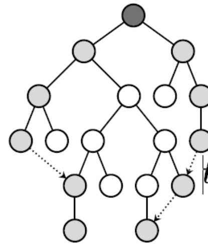


Figure 6.7.: Illustration of thread-pointer in the Improved-Walker algorithm. The connections established via the thread-pointer, are shown with dotted lines. The root of the subtree is shown in dark gray and the vertices belonging to the contour are colored light gray. ([2], p. 4).

The Improved Walker algorithm consists of three steps:

1. Preprocessing

2. First walk

3. Second walk

The three steps in detail:

1. **Preprocessing:** The preprocessing step defines and initializes the variables. Only the name of the procedure has been changed in contrast to the pseudo-code presented in [2]. The parameter T is the graph to draw. The pseudo-code is adopted from the one presented in [2].

```
procedure preprocessing(T)
begin
  foreach(v:vertex of T)
    mod(v)=0;
    thread(v)=0;
    ancestor(v)=v;
  r = root(T)

  firstWalk(r);
  secondWalk(r, -pre(r));
```

2. **First walk:** This step calculates the preliminary coordinates for the vertices. As stated before, the approach is bottom up which means that the step *first walk* is done recursively first for all the vertices below v and then for v. The major task is done inside the procedure *apportion* which is kind of the heart of the algorithm. The sequence of execution for non-leaf-vertices is as follows:

   a) Remember the leftmost child of v as *defaultAncestor*
   b) Call first *firstWalk(w)* and then *apportion(w,defaultAncestor)* for all the children w of v.
   c) Execute the shifts for v.
   d) Finally set the preliminary x-coordinate for v.

   Inside the procedure *apportion*, the combination of the actual subtree with the previous subtrees is done. As mentioned before, the threads play an important role in this step for traversing the inside- and outside-contours of the subtrees up to the highest common level. The vertices are referred to as follows:

   - v_+^i means the inside right subtree
   - v_+^o means the outside right subtree
   - v_-^i means the inside left subtree
   - v_-^o means the outside left subtree

   Furthermore the following variables are used to sum up the modifiers along the contour:

   - s_+^i means the sum of the modifiers of the inside right subtree
   - s_+^o means the sum of the modifiers of the outside right subtree

- s^i_- means the sum of the modifiers of the inside left subtree
- s^o_- means the sum of the modifiers of the outside left subtree

If there are two conflicting nodes of the inside contours (see figure 6.8), the procedure *ancestor* is called, which figures out the left one of the *greatest distinct ancestors*. Afterwards the procedure *moveSubtree* is called to prepare the separation of the subtrees. The real movement is done by the function *executeShifts*. As the final coordinates are calculated in the next step, only the preliminary coordinates and modifiers of the direct[2] child nodes have to be shifted.

Two more procedures are needed: *nextLeft(v)* and *nextRight(v)* which traverse the left respectively right contour of the subtree/subforest of v.

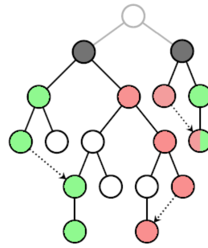


Figure 6.8.: Inner (red) and outer (green) contours of two subgraphs. This image is a modification of figure 1 from [2] on page 4.

```
procedure firstWalk(v:vertex)
begin
if(v is a leaf)
then
  pre(v)=0;
  if(v has a left sibling w)
    then let pre(v) = pre(w) + distance;
else
  defaultAncestor  = leftmost child of v;
  foreach(child w of v from left to right)
    firstWalk(w);
    apportion(w,defaultAncestor);
  executeShifts(v);
  midpoint = ((pre(leftmost child of v) + pre(rightmost child of v))/2
  if(v has a left sibling w)
  then
    pre(v)=pre(w)+distance;
    mod(v)=pre(v)-midpoint;
  else
```

[2]Direct means that only the child nodes (w) of v are shifted but not the child nodes of w on their part.

```
    pre(v)=midpoint;
end;

procedure apportion(v:vertex, defaultAncestor:vertex)
begin
  v^i_+=v^o_+=v;
  v^i_-=w;
  v^o_m= leftmost sibling of v^i_+;
  s^i_+=mod(v^i_+);
  s^o_+=mod(v^i_+);
  s^i_-=mod(v^i_-);
  s^o_-=mod(v^i_-);
  while(nextRight(v^i_-)≠0 and nextLeft(v^i_+)≠0)
    v^i_-=nextRight(v^i_-);
    v^i_+=nextRight(v^i_+);
    v^o_-=nextRight(v^o_-);
    v^o_+=nextRight(v^o_+);
    ancestor(v^o_+)=v;
    shift=(pre(v^i_-) + s^i_-)-(pre(v^i_+)+s^i_+)+distance;
    if(shift > 0)
      moveSubtree(ancestor(v^i_-,v,defaultAncestor),v,shift);
      s^i_+=s^i_++shift;
      s^o_+=s^o_++shift;
    s^i_-=s^i_-+mod(v^i_-);
    s^i_+=s^i_++mod(v^i_+);
    s^o_-=s^o_-+mod(v^o_-);
    s^o_+=s^o_++mod(v^o_+);
  if(nextRight(v^i_-)≠0 and nextRight(v^o_+)=0)
    thread(v^o_+)=nextRight(v^i_-);
    mod(v^o_+)=mod(v^o_p)+s^i_--s^o_+;
  if(nextLeft(v^i_+)≠0 and nextLeft(v^o_-)=0)
    thread(v^o_-)=nextLeft(v^i_+);
    mod(v^o_-=mod(v^o_-)+s^i_+-s^o_-;
    defaultAncestor=v;
end;

procedure nextLeft(v:vertex)
begin
  if(v has child)
    then return the leftmost child of v;
    else return thread(v);
end;

procedure nextRight(v:vertex)
begin
```

```
  if(v has child)
    then return the rightmost child of v;
    else return thread(v);
end;

procedure moveSubtree(w₋:vertex, w₊, shift)
begin
  subtrees=number(w₊)-number(w₋);
  change(w₊)=change(w₊)-shift / subtrees;
  shift(w₊)=shift(w₊)+shift;
  change(w₋)=change(w₋)+shift / subtrees;
  pre(w₊)=pre(w₊)+shift;
  mod(w₊)=mod(w₊)+shift;
end;

procedure executeShifts(v)
begin
  shift=0;
  change=0;
  foreach(w:child of v from right to left)
    pre(w)=pre(w)+shift;
    mod(w)=mod(w)+shift;
    chage=change+change(w);
    shift=shift+shift(w)+change;
end;

procedure ancestor(vⁱ₋:vertex, v:vertex,defaultAncestor:vertex
begin
  if(ancestor(vⁱ₋ is sibling of v)
    return ancestor(vⁱ₋);
  else
    return defaultAncestor;
end;
```

3. **Second walk:** The *second walk* computes the final coordinates by recursively summing up the modifiers.

```
procedure secondWalk(v:vertex, m:vertex)
begin
x(v) = pre(v) + m;
y(v) = level(v);
foreach(w:child of v)
  secondWalk(w,m+mod(v));
end;
```

6.3.3. Complexity

The complexity of the preprocessing step is simply $O(n)$ and $\Omega(n)$ as it addresses each vertex exactly once to set its initial values.

The *first walk* is more complex. The point of interest is the call of *apportion* and *executeShifts*. The rest of this step is simple traversing the tree once ($O(n)$ and $\Omega(n)$. First of all, *apportion* is considered. It calls four other procedures: *nextRight*, *nextLeft*, *ancestor* and *moveSubtree*. Fortunately, these procedures are uncritical as they have neither loops nor further procedure calls which means that they are linear. The only thing left is the while-loop of *apportion* procedure. It goes down the inner contour of the two subgraphs until one or both are down to the lowest level. This is done each time two subtrees are connected. The worst case scenario for that would be a tree with a depth of one. This means $g_G^+ = n-1$. That is because there the apportion procedure has to be called n-2 times. One comparison has to be made for every pair of neighbouring vertices. This results in a complexity of $O(n-2)$. Such a graph with n=9 is shown in figure 6.9.

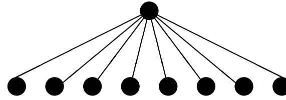


Figure 6.9.: The worst case for the *apportion*-procedure of Improved Walker.

The minimal complexity has to be considered separately this time. In an ideal case there is no need to compare two subtrees. That is the case if the graph is a path. Figure 6.10 illustrates such a case. As no comparison has to be made at all minimum complexity is simply $\Omega(0)$.

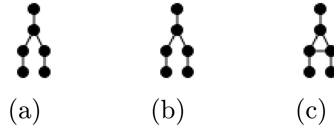
Figure 6.10.: The ideal case for the *apportion*-procedure of Improved Walker.

The complexity of the *second walk* is simply $O(n)$ and $\Omega(n)$ as it traverses the complete tree once.

$$\begin{aligned} &O \quad (3n + n - 2) \\ &\Omega \quad (3n) \end{aligned} \tag{6.8}$$

6.3.4. Results

As already stated before, only the small drawings are presented here. The others can be found at least partly in *Appendix C.3*. The complete drawings are on the CD in directory `Drawings/ImprovedWalker`. Figure 6.11 shows the three smallest graphs. (a) is the Minimum Decision, (b) is Minimum Other and (c) is Minimum Sequence.



(a) (b) (c)

Figure 6.11.: The drawings of the smallest graphs done by Improved Walker.

	Crossings	**Area**	**Length**	**Diff @ G**	**Diff at v**	**Gap**
1_(10139)	545	57750x238	571342	15199	15199	0
1_(563)	5	3289x154	23832	1171	1013	1
2_(471)	0	5949x42	111117	2793	2793	7
2_(7311)	439	93939x56	2119550	44064	44064	1
3_(670)	49	7521x84	61189	3526	3526	0
3_(9551)	2369	102318x84	1539857	45810	45810	0
4_(564)	0	1649x322	14538	479	479	14
4_(9787)	0	27758x490	346210	16753	16753	14
Min Decision	0	14x42	73	2	2	14
Min Other	0	14x42	73	2	2	14
Min Sequence	0	14x42	87	2	2	14
Special(100)	0	0x1330	1442	0	0	14
Special(10000)	0	0x146286	146286	0	0	14
Sequence	0	58226x112	659961	25402	25402	14
Decision	0	115942x280	1021576	25235	25235	14
Other	0	58198x126	494360	15536	15170	14

Table 6.3.: Result of the Improved Walker algorithm.

6.3.5. Evaluation

As the results show, the clearance is not guaranteed. Even worse it goes down to zero which means overlapping vertices. These problems do not appear at the simple and pure-element graph types. As with the Leaf-First-Layering algorithm discussed before, the number of crossings seems to be related to the outer degree of the vertices. Unlike Leaf-First-Layering where a clear relation is visible, it is only visible at two of the test objects

for Improved Walker. At 4_564 and 4_9787, there are no major structural differences in comparison to the other graphs except for the outer degree. For these two graphs, the outer degree is only three while it is higher for the other graphs. A direct relation is not visible as the number of edge crossings is highest for 2_7311 and 2_471 but the number of crossings is lowest there. As a result, it could be said that the maximum outer degree of a vertex decides if the algorithm is able to draw a planar graph or not but it does not give any information about the existence or the magnitude of edge crossings.

With regard to the area needed for the drawing, it can be said that the drawings are very compact. Improved Walker requires even less space than Leaf-First-Layering. The difference is quite small compared to the drawings of Dominance-Straight-Line(6.2), too. As already discussed before, the total edge length, total edge length difference and at one vertex are related to the area needed for the graph. In comparison to Leaf-First-Layering, the lengths of Improved Walker are smaller. At this point, one has to be mentioned that the differences in required area and edge lengths can not be taken as absolute values so easily as graphs with less crossings normally have longer edges as shown in figure 6.12. An algorithm with edge crossings can produce more compact drawings than one creating planar drawings.

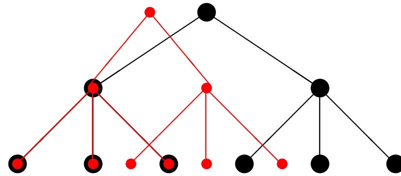


Figure 6.12.: Required area and edge lengths are less at graphs with crossings.

6.4. Magnetic Spring Model - Centric

6.4.1. Overview

The Magnetic Spring Model is an algorithm proposed by many papers and books one of which is [4]. Another common name is “Rings and Springs”. It is one of the algorithms simulating physics, in this case, magnetic particles (vertices) inside a magnetic field with edges as springs between the particles. Figure 6.13 shows a draft. There are different magnetic fields possible. Some common ones are shown in figure 6.14.

For this algorithm, a centric field was chosen with its origin in the origin of the coordinate system (top left). As the field is not limited, there has to be a mechanism which holds it near to the origin. This is done by placing the root at the origin and not letting it move away from that position.

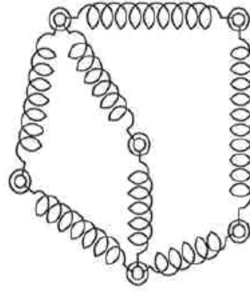
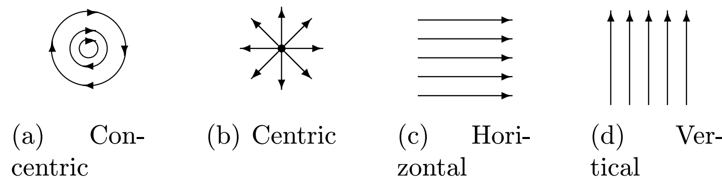


Figure 6.13.: Illustration of the Magnetic Spring Model ([12], p. 44).



(a) Concentric (b) Centric (c) Horizontal (d) Vertical

Figure 6.14.: Common types of magnetic fields.

6.4.2. Mode of Operation

This algorithm is divided into two steps. First of all, the preprocessing step places the vertices on their initial positions. Secondly, the physical system starts simulating the forces and moves the nodes several times. Considering all the vertices once is called *simulation step* or *iteration*. The number of iterations can be defined in advance or be dependent on properties of the drawing. An example for the later approach would be the culminated distance of all vertices moved. However, the algorithm presented here will use a fixed number of iterations. There has to be made a trade off between time and quality. Ideally the system will end up in a *locally minimum energy configuration* (cp. [4], p. 303). This is reached if there are no more changes between the iterations. For big graphs, this might take very long. Therefore, the number of iterations is limited.

1. **Preprocessing:** In this first step, all vertices except the root are distributed randomly around a circle with its origin at: ($x = radius$,$y = radius$). The *radius* has to be previously defined. This ensures that all the vertices are (and as the magnetic field is centric stay) in the third quadrant (cp. figure 6.15).

   The root is placed at the origin (0,0).

   ```
   k = 5; // k is a factor for the size of the circle
   springLength = 10;
   procedure preprocessing(root:vertex, nodes:listOfNodes)
   begin
     radius = k*nodes.length/2*PI;
   ```

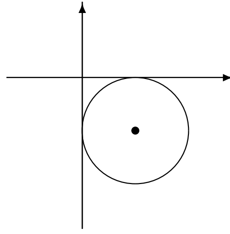


Figure 6.15.: Preprocessing for Magnetic Spring Model.

```
    step = 2*PI/nodes.length;
    for(i from 0 to nodes.length)
      v = nodes(i);
      // calculate x and y
      y(v) = SIN(i*step)*radius+radius+1;
      x(v) = COS(i*step)*radius+radius+1;
      i++;
  end;
```

2. **Simulation:** When all vertices are at their initial place, the simulation starts. Simulating is a relatively easy process consisting of two steps:
   a) Calculate forces
   b) Move vertices

   It is important to move the vertices after the forces of all vertices were calculated. Moving the vertices in advance will change the forces of other (not yet calculated) vertices which is not intended.

   a) **Calculate forces:** The force of a vertex is made up of two kinds of forces: The force of the magnetic field and the force of the springs at this vertex.
      i. **Magnetic force:** There is only one magnetic force per vertex as a homogenous field is used. Its value is equal everywhere (*magenticFieldStrength*). Its direction is the elongation of the vector between the fields origin (o) and the node (v).
      ii. **Spring forces:** There can be more than one force caused by the springs. In fact, each node which is neither root nor leaf has at least two forces because of springs.
      The value of the force is the distance between $\alpha(e)$ and $\omega(e)$ of edge e minus the normal length (d) of the spring (*springLength*) divided by two ($\frac{|\alpha(e)-\omega(e)|-d}{2}$). The normal length of the spring may invert the direction of the force if the vertices are too close[3].
      The direction of the force is the elongation of the vector from $\alpha(e)$ to $\omega(e)$. As stated in the paragraph above, the direction might be inverted.

[3]Too close means that the distance between $\alpha(e)$ and $\omega(e)$ is smaller than the normal length of the spring.

The forces have to be summed up altogether. The resulting force ($force(v)$) is the one which affects the node (v).

```
magneticFieldStrength = 5;
springLength = 10;

procedure calculateForces(v:vertex)
begin
  if(v = root)
  then
    x(v) = 0;
    y(v) = 0;
    foreach(c:Connection of out(v))
      calculateForces(c.end);
  else
    // Calculate the magnetic force
    alpha = ARCCOS(y(v) / (SQUARE_ROOT(x(v)*x(v) + y(v)*y(v)));
    y(force(v)) = SIN(alpha)*magneticFieldStrength;
    x(force(v)) = COS(alpha)*magneticFieldStrength;

    // Calculate spring forces for outgoing edges
    foreach(c:out(v) of v)
      dx = x(c.begin) - x(v);
      dy = y(c.begin) - y(v);
      z = SQUARE_ROOT(dx*dx + dy*dy);
      beta = ARCCOS(dx/z);
      xSpringLength = COS(beta)*springLength;
      ySpringLength = SIN(beta)*springLength;

      x(force(v)) = x(force(v)) + x(v)-x(c.end) + xSpringLength;
      y(force(v)) = y(force(v)) + y(v)-y(c.end) + ySpringLength;
      calculateForces(c.end);
end;
```

b) **Move vertices:** The last step of an iteration is the movement of the vertices. Therefore, each vertex is moved by the vector given by the force calculated for this vertex.

```
procedure move(v:vertex)
begin
  x(v) = x(v) + x(force(v));
  y(v) = y(v) + y(force(v));
  foreach(c:Connection of out(v))
    move(c.end);
end;
```

```
procedure iterate(root:vertex)
```

```
begin
  calculateForces(root);
  move(root);
end;
```

6.4.3. Complexity

The preprocessing step calculates the initial position for each vertex once. This makes a complexity of $O(n)$ and $\Omega(n)$.

For the simulation step, this is not so easy as it has to be run multiple times. The number of executions of the simulation step is referred as m. The two steps of the simulation: *calculate forces* and *move vertices* are done separately. There has to be a calculation done for each connection (spring forces) and for each vertex (magnetic force). Fortunately, both forces can be assessed by traversing the tree only once. This makes a complexity of $O(n)$ and $\Omega(n)$.

The second part of the simulation is the movement of the vertices. As each node has to be considered exactly once, the complexity is also $O(n)$ and $\Omega(n)$.

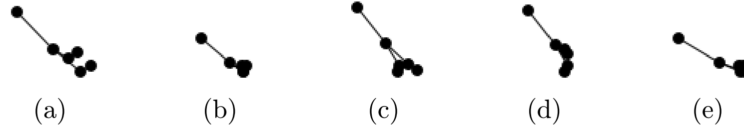
Unlike the other algorithms, the Magnetic Spring Model algorithm is not arithmetic. It is a simulation and in almost all cases, it is not possible to find the solution in only one step (e.g. $m = 50$ for the drawings made in this thesis). This introduces a variable (m) containing the number of iterations into complexity which is described for the algorithm as follows:

$$\begin{aligned} O &\quad (3mn) \\ \Omega &\quad (3mn) \end{aligned} \tag{6.9}$$

6.4.4. Results

As with the other drawings, those produced by the Magnetic Spring Model are too big to show them here. All drawings are shown at least as parts in *Appendix C.4* and completely on the CD in the directory `Drawings/MagneticSpringModel`. In difference to the other algorithms, the Magnetic Spring Model produces non-deterministic outputs. The reason for this is that the vertices are placed randomly on the circle in the preprocessing step. To illustrate this, five different drawings of the Min Decision-graph are shown in figure 6.16.

For a meaningful evaluation of the results, it would not be enough to do only one test run for each test object. The risk that the randomly created initial state is a best- or worst-case by chance makes it necessary to do more test runs. In this thesis, five testruns are executed for each test object. The table presented here (6.4) shows the averages for the single values calculated during the tests. The tables with the results can be found in *Appendix C.4*.



(a) (b) (c) (d) (e)

Figure 6.16.: Five drawings of the same graph with Magnetic Spring Model - Centric.

	Crossings	Area	Length	Diff @ G	Diff @ v	Gap
1_(10139)	333714	72604x89740	65516659	13820	13688	0
1_(563)	3269	1974x2229	250419	702	674	0
2_(471)	11218	923x895	8665	677	671	0
2_(7311)	886532	1501x1502	3660292	755	691	0
3_(670)	11638	541x552	58871	253	246	0
3_(9551)	1734140	3907x3608	13040541	2600	2513	0
4_(564)	1146	3996x5429	147577	627	532	0
4_(9787)	198796	92533x111712	42748538	12298	11890	0
Min Decision	1	38x34	71	25	17	4
Min Other	1	35x26	61	24	18	2
Min Sequence	1	30x28	65	24	17	3
Special(100)	12	749x871	4215	84	22	4
Special(10000)	179945	117995x14256	44502886	13395	4163	6
Sequence	989250	14392x15933	58038576	8208	8144	0
Decision	1002739	212329x197891	209204064	29437	8461	0
Other	566893	20320x45056	62841221	10977	3117	0

Table 6.4.: Average result of Magnetic Spring Model with a centric magnetic field.

6.4.5. Evaluation

Looking at the clearance, it is turns out that only very small graphs have clearance at all and even there not in all of the five test runs. It could be said that a graph drawn with the Magnetic Spring Model algorithm has a clearance above zero only by chance.

For the number of edge crossings, the results are even worse on average, not even one of the tested graphs is drawn planar. The correlation between number of vertices (which is strongly related to the number of edges) and the number of edge crossings is explicitly visible.

Having a look at the area needed for the drawing, it becomes clear that it is weakly related to the number of vertices. There are exceptions from this however, for example, 2_7311 only needs a small area. In return, the number of crossings is much higher. Another observation which can be made is that the shape of the required area is not consistent. Some drawings are roughly quadratic, others are higher than wide or the other way around. However, the differences between width and height are not so enormous as with the Improved Walker drawings. This means that width and height have no

regular relation but are at least relatively close together.

The total length of the edges shows a two-folded result. For the three minimal graphs, the edge length is significantly shorter. For most of the other graphs, the edges are much longer, sometimes by the factor ten. In contrast to that, the total edge length difference as well as at one vertex is much smaller than with the other algorithms. One reason for this is the fact that the edges are seen as springs and so the vertices are pulled together. That is something which never happens with the other algorithms in this way. Secondly, the vertices are not arranged on layers which allows the nodes to be arranged around the ancestor vertex rather than on a passant line (the straight line layer). This revokes the effect generating the big edge length differences described in section 6.2.5.

Considering the differences between the total edge length difference against the difference at one vertex, it becomes clear that there are sometimes big differences between those two values. This marks another distinction to the other algorithms. The reason for this is, amongst other things, the freedom of placement. Again the effect described in section 6.2.5 does not apply for the Magnetic Spring Model algorithm.

Finally it has to be emphasised this algorithm takes a tremendous amount of time compared to the other algorithms presented in this thesis. Furthermore, it has many parameters which might be changed to get better or worse results: For instance the force of the springs or the strength of the magnetic field, the radius of the circle to which the vertices are placed in the preprocessing step or the number of executions of the simulation step. All these parameters make this algorithm harder to use than the arithmetic ones.

7. Analysis and Perspectives

In this chapter, the tested graph drawing algorithms will be compared and their evaluation will be summarised concerning the specific graph given by the LAUTS structure. Additionally, the last section will highlight some perspectives and further work which can be done to improve the algorithms.

7.1. Comparison of the Algorithms

7.1.1. Transformers

There were two types of algorithms under test in this thesis. First of all, the Transformers which were used to transform the graph from one type into another one. They are used as additives to be able to apply algorithms to the graph which were originally not able to handle that type of graph. They could be divided into three types depending on the input- and output-type of graph:

- General graph without self loops into a tree (Defect-Draw-Repair, Cycle Substitution)
- General graph without self loops into an acyclic graph (Cycle Removal).
- n ary tree to binary tree (Bi-Treeing).

To be able to get meaningful results for the graph drawing algorithms, it is necessary to decide on one or a combination of Transformers which are used for all algorithms evaluated. Using different Transformers would distort the results and make them incomparable.

Bi-Treeing This Transformer was actually introduced to be able to apply the graph drawing algorithm described in [17] (Walker 90) as this algorithm is only able to handle binary trees. Before Bi-Treeing can be applied, the graph has to be converted into a tree by another Transformer.

Walker90 had a non-linear time constraint which was improved by the algorithm proposed by [2] called Improved Walker which additionally is able to handle n ary trees thus, Bi-Treeing became obsolete. However, it is still presented in this thesis as it might be useful for further improvements of the graph drawing algorithms. This topic is handled in the next section, 7.2.1.

The major disadvantage of this Transformer is the fact that it increases the depth of the tree by introducing dummy nodes.

Cycle Removal The name of this Transformer proposed by [4] is misleading as it does not remove something from the graph but it inverts one edge per cycle. This method has two big advantages. First of all, there is only one edge which has to be changed during the transformation phase and so also only one in correction phase. The second advantage is that this edge is only changed and not removed.

In contrast to the Defect-Draw-Repair method and Cycle Substitution, Cycle Removal allows this edge to be considered during the drawing process. This has an advantage for routing the edges as all of them are considered at once. Removing an edge in the transformation phase means that it has to be routed after the correction phase of the Transformer was executed and the previously removed edge was re-added to the graph.

The major disadvantage of Cycle Removal is that this Transformer only produces an acyclic graph which means that this method can only be used for the Magnetic Spring Model - Centric as all other algorithms require the graph to be a tree.

Defect-Draw-Repair This Transformer is capable to transform a general loop-less graph into a tree. In fact, the resulting graph is even a rooted tree as there exists exactly one vertex with $g_G^- = 0$ which is the definition of a tree's root vertex.

A major advantage of the Defect-Draw-Repair method is that it does only little changes to the graph and has a static complexity of $O(1)$[1] due to that. Only one edge per cycle has to be removed in the transformation phase. Therefore, only one edge has to be added in the correction phase. This makes the Defect-Draw-Repair method superior to Cycle Substitution which has a higher complexity.

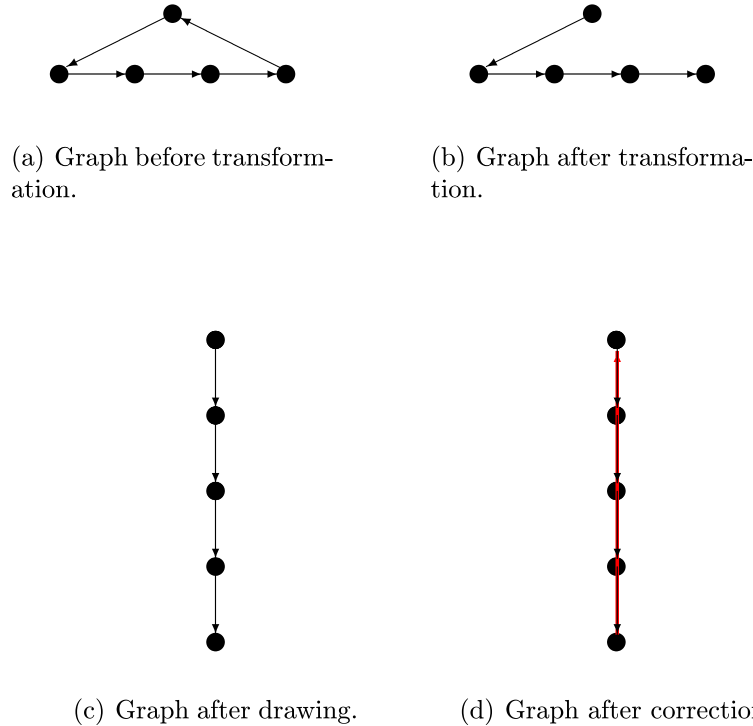
Unfortunately, the drawings which result from all graph drawing algorithms except Magnetic Spring Model - Centric using Defect-Draw-Repair method have a disfigurement at Sequences. The cycle sub-vertices which should be on one layer are underneath each other, each one on an own layer. Figure 7.1 illustrates the problem.

An additional problem visible at 7.1(d) is that the edge removed at the transformation phase and added again after the drawing in correction phase was not considered during the drawing process, which can produce edge crossings. In fact, it is rather unlikely that no edge crossings are produced. In this case, none of the cycle sub nodes has child vertices. That is a worst case as the number of edge crossings is maximal there (n crossings where n is the number of cycle sub nodes).

Especially the placement of one Sequence's child nodes to different layers, leads to the decision not to take this Transformer for testing the graph drawing algorithms.

Cycle Substitution Like the Defect-Draw-Repair method, Cycle Substitution transforms a general loop-less graph into a rooted tree. Both break up the cycles. The difference is the way how this is done. While the Defect-Draw-Repair method does only changes to one edge, Cycle Substitution changes all except one. So those methods could be seen as contrary approaches at least in the face of the complexity.

[1]The complexity is one as only one edge has to be changed. It does not consider the cycle detection.



(a) Graph before transformation.

(b) Graph after transformation.

(c) Graph after drawing.

(d) Graph after correction.

Figure 7.1.: Problem of drawings done with the Defect-Draw-Repair method. (a) shows the graph before the transformation, (c) shows the graph after the transformation and (d) shows the graph after the correction.

The disadvantage of this Transformer is that there are many changes done to the graph. However, the complexity is still linear: $O(n-1)$. As stated before, only one edge is left unchanged. This has of course an effect on the runtime of the Transformer. The more cycle sub vertices there are, the bigger is the difference between Defect-Draw-Repair method and Cycle Substitution relating to the runtime.

The major advantage of Cycle Substitution is that the graph drawing algorithms produce drawings out of the transformed graph which much better match the aesthetic criteria defined in the style guide than those done with the Defect-Draw-Repair method. Furthermore, the problem of producing edge crossings is solved. There is neither a case where two edges of the transformed graph are crossing nor one where the edges of the corrected graph do if the graph drawing algorithm produces planar drawings.

Summary Before the graph drawing algorithms can be tested, one of the Transformers has to be selected which is used for all algorithms to be able to compare them to each other.

Bi-Treeing is not necessary as n ary rooted trees are enough for all the drawing algorithms under test. Cycle Removal produces only acyclic graphs while trees would

be necessary for three of the four algorithms. This means the decision had to be made between the Defect-Draw-Repair method and Cycle Substitution. As Cycle Substitution in combination with the three arithmetic graph drawing algorithms produces better[2] results, Cycle Substitution was chosen as the Transformer used to prepare the graphs for drawing.

7.1.2. Graph Drawing Algorithms

Magnetic Spring Model - Centric The Magnetic Spring Model with a centric magnetic field does not perform well in drawing trees. The algorithm is very simple and its claims are low as it is able to draw even acyclic graphs while all other algorithms must have rooted trees. Thus the major advantage of this algorithm is that it can handle less structured graphs.

The disadvantages, however, are clearly visible looking at the results: a huge number of edge crossings, the fact that in bigger graphs no global structure is discernable and last but not least the fact that the resulting drawings are indeterministic let this algorithm not be competitive to the others.

One additional fact is the number of parameters which can be changed, such as magnetic field strength and forces of the springs. Choosing the wrong parameters may cause the drawing to "explode" or "collapse". Even the complexity depends on one of these parameters. The number of iteration affects the complexity directly. A better application for this algorithm are less structured graphs.

All these facts disqualify this algorithm.

Dominance-Straight-Line Analysing the drawings of this algorithm, it becomes clear that they look very decent, even though they have to be rotated by 45 degrees clockwise to become visible as horizontally layered drawings. Furthermore, this algorithm performs best in regard to edge crossings and in ensuring the clearance of vertices. There was not even one edge crossing in the test cases. The clearance was not only adhered to, it was even reached exactly (on the lowest layer). Unfortunately, Dominance-Straight-Line needs much space for its drawings even after a 45 degree clockwise rotation which implies higher edge lengths.

Another unattractive fact is that there are sorted lists necessary which in worst case raise the complexity to quadratic.

All these facts recommend this algorithm for smaller or sub-graphs but not for universal use.

Leaf-First-Layering Even though this algorithm is rather simple, it produces good results in reasonable time. Additionally, it fulfills with its orthogonal lines one of the "nice to have" claims of the style guide. As Dominance-Straight-Line and Magnetic-Spring-Model - Centric are not universally usable, Leaf-First-Layering only competes with the Improved Walker algorithm.

[2]Better means that the resulting drawing fits better to the aesthetic criteria defined in the style guide.

The only measured value where Leaf-First-Layering is superior to Improved Walker is the clearance of the vertices. As with Dominance-Straight-Line, Leaf-First-Layering delivers exactly the desired value. All other resulting values are slightly worse than those of Improved Walker.

Looking at figure 7.2, it becomes clear that Leaf-First-Layering is obviously inferior to Improved Walker in view of complexity.

Altogether, it can be stated that Leaf-First-Layering produces better results than Dominance-Straight-Line except for the number of line crossings but it is not as good as Improved Walker except for the clearance.

Improved Walker This graph drawing algorithm is the fastest of the four considered here. Its results for the number of line crossings and clearance are not as good as those of Dominance-Straight-Line but therefore, the required space as well as the line lengths are much better.

All in all, Improved Walker is a algorithm which produces good drawings in reasonable time. This makes this graph drawing algorithm the first choice for drawing the graphs in LAUTS.

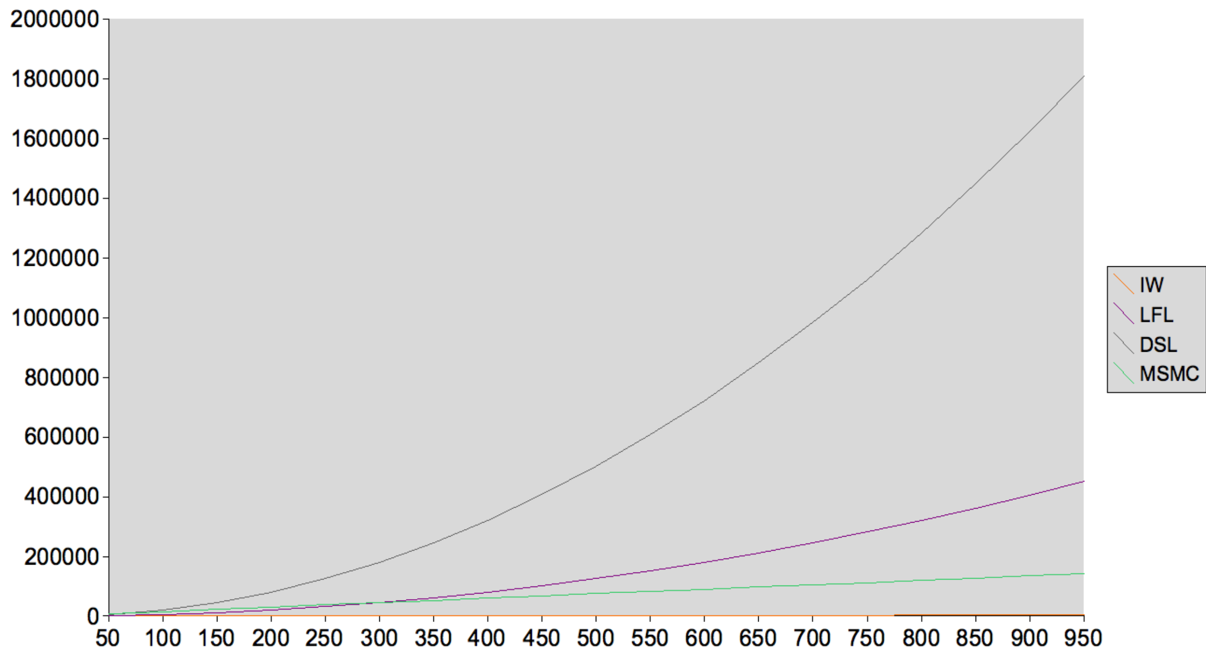


Figure 7.2.: Complexity of graph drawing algorithms. IW = Improved Walker, LFL = Leaf-First-Layering, DSL = Dominance-Straight-Line, MSMC = Magnetic-Spring-Model - Centric.

7.2. Further Work and Perspectives

So far, not all problems could be solved and not all ideas could be tested. The goal of this work was to find a suitable algorithm for drawing the graphs used by LAUTS. The Improved Walker algorithm in combination with the Cycle Substitution Transformer are a good team for that task. However, there might be some additional modifications to the algorithm which produce even better drawings.

7.2.1. Perspectives

[2] states that there are no edge crossings in drawings done by Improved Walker. This could not been confirmed by this thesis as the presented results show a counterexample. Previous versions of this algorithm were used to draw binary trees. It might be interesting to add the Bi-Treeing Transformer after Cycle Substitution was executed. Maybe with this trick it would be possible to obtain drawings without edge crossings.

Another approach which might be interesting is to apply a simulation like the Magnetic-Spring-Model - Centric to a graph drawn by an other algorithm. For example, if the connections are handled as springs after Leaf-First-Layering, the total edge length as well as the edge length differences could be reduced. An extrem example would be a vertex which is a direct child of the root in a tree of depth four. This would mean that the vertex is placed on the lowest level and has an edge across all layers to the root. Considering the edges as springs now could pull this vertex up and produce an even better drawing. Of course, one would have to make sure that no new edge crossings are produced and that the clearance is kept.

7.2.2. Further Work

One parameter was explicitly excluded from this thesis: the size of vertices. In the real diagram used in LAUTS, vertices have different sizes. Depending on the algorithm used to draw the diagram, it is challenging to include differently sized vertices. While this does not matter for Magnetic-Spring-Model - Centric, it might become more complex for the Improved Walker algorithm.

Part III.
Appendix

A. Tree Diagram Style Guide

This part of the document describes the way a diagram should look like. It clarifies the open questions about the appearance of the diagram which is not given by the structure of the graph.

In contrast to the rules given by the graph structure, these style rules are not mandatory. They are used to define rules for the graph layout algorithms. This means that the graph should look as similar to the rules defined in the style guide as possible but it does not have to look exactly like the definition.

The rules concerning the positioning of the nodes are defined in a first step. Afterwards, the rules for the connection are given.

A.1. Placement of Vertices

A.1.1. Vertical Vertex Layout

In section 3.1, the structure of LAUTS is given as it is realized by the data model of the software. It is obvious that it is hierarchical. This should be visible in the diagram too, although the graph to draw is not a tree any more. Nevertheless, the diagram should have at least a (rooted) tree-like appearance.

In detail, this means that the nodes are arranged in layers. A node A which is contained by a node B is drawn one layer below B. Thus, all nodes contained by B are drawn on the same layer. This implies that all sub nodes of the nodes on one layer are arranged one line below their foregoing nodes. Figure A.1 shows the arrangement by layers.

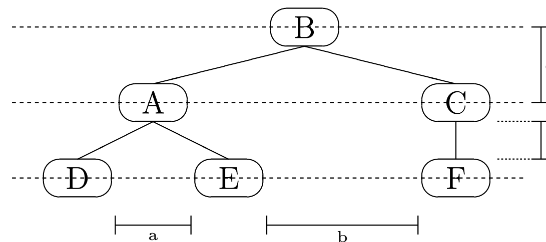


Figure A.1.: Layer-oriented appearance.

A.1.2. Horizontal Vertex Layout

For the horizontal layout of the diagram the layers are used as well. A previous node is placed at the median center of its child nodes. Furthermore, the child nodes of a node are placed right beside each other. This will avoid edge crossing within inter-layer-edges. An illustration is shown in figure A.1.

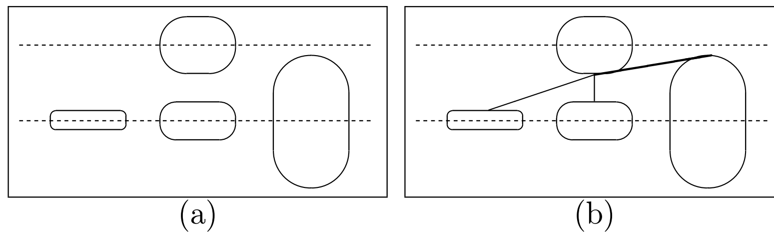
A.2. Spacing

To prevent overlapping of nodes, a minimum distance in vertical as well as in horizontal direction is required.

A.2.1. Vertical Spacing

The vertical spacing could be done in two ways. The first is that the spacing between the layers (d) is dependent on the size of the vertices in the layers. The second is that d is the same for all layers. To avoid overlapping, (d) depends on c which defines the minimal distance between the lower bound of the largest vertex of the upper layer and the upper bound of the largest vertex of the lower layer.

With regard to height optimization, it might make sense to arrange the vertices on the layers in such a way that the bigger vertices on one layer are on the opposite of the smaller vertices on the other layer. Figure A.2(a) visualizes this. Unfortunately, the effect shown in figure A.2(b) is not desirable.



(a) (b)

Figure A.2.: Height optimization problem.

A.2.2. Horizontal Spacing

There are two types of horizontal spacing. One between sub nodes of the same parent node (a) and one between the sub nodes different parent nodes (b). While a is a fixed value, b is variable. This is because b not only depends on the actual layer but also on the layers above. Because of this, there is only a minimum defined for b.

A.3. Edges

Edges are split into two categories:

- inter-layer edges
- inner-layer edges

Depending on the category of edge, there are different rules.

A.3.1. Inter-Layer Edges

These edges are used most frequently. They connect two nodes which are on different layers (parent - child relationship). An inter-layer edge should always “fall” from its start to the end point. That means if the starting point is considered to be the point of origin, the edge has to take course through the third or fourth quadrant of the coordinate system. The starting point is always the center of the lower bound of the node on the higher layer. The ending point of the connection is the center top of the lower layer node. Depending on the element represented by the higher node, the edges have different appearance (see *table A.1*).

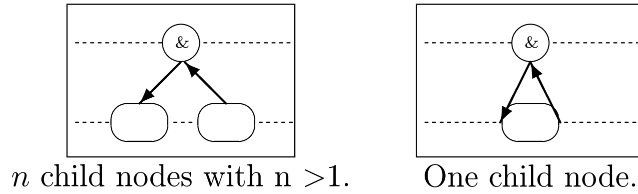
Except when the higher layer node is of type Sequence, there is exactly one connection between a parent node (higher layer node) and its child nodes (lower layer nodes). For Sequence nodes, there are exactly two connections between the Sequence (higher level) node and its child nodes: one connecting to the first and one connecting from the last child node. As visible in figure A.3, in respect of the number of connections, it does not matter how many child nodes there are.

Parent Element	Top	Bottom
Data Elements	—	—
Random	⇐	⇒
RandomPutBack	◀—	—▶
RandomAll	⇐	⇒
Decision	—	—
Sequence	—	—▶

Table A.1.: Appearance of edges subject to the preceding node.

A.3.2. Inner-Layer Edges

This type of edge is only used if the higher layer node is of type Sequence. They direct and connect the vertices representing the Sequence’s child nodes which by definition are all on the same layer. For better readability, the vertices have exactly the sequence as the elements they represent from left to right side. This ensures that all edges are running in the same direction (from left to right) and so by definition edge crossings



n child nodes with n >1. One child node.

Figure A.3.: Inter-layer edges with a Sequence as cycle root.

between inner-layer edges are not possible by definition. The appearance of the edge is the same as already defined for Sequence in table A.1.

A.3.3. Orthogonal Edges

As straight line edges do not look very professional, the edges should be routed orthogonal. The horizontal part of such an orthogonal line is in between two layers even if the layer of the nodes differs more than 1. This avoids an overlap between an edge and a node.

B. Set of Graphs used as Test Objects

B.1. Details of the Test Objects

B.1.1. Minimal Decision

Name	Frequency
TestSuites	0
TestCases	1
Functions	3
Actions	1
Reactions	0
Sequences	0
Randoms	0
RandomAlls	0
RandomPutBacks	0
Decision	1
Number of vertices	6
Depth	4
Leafs	2
Maximal outer degree	2

B.1.2. Minimal Sequence

Name	Frequency
TestSuites	0
TestCases	2
Functions	1
Actions	2
Reactions	0
Sequences	1
Randoms	0
RandomAlls	0
RandomPutBacks	0
Decision	0
Number of vertices	6
Depth	4
Leafs	2
Maximal outer degree	2

B.1.3. Minimal Other

Name	Frequency
TestSuites	0
TestCases	2
Functions	1
Actions	2
Reactions	0
Sequences	0
Randoms	1
RandomAlls	0
RandomPutBacks	0
Decision	0
Number of vertices	6
Depth	4
Leafs	2
Maximal outer degree	2

B.1.4. Pure Decision

Name	Frequency
TestSuites	0
TestCases	1
Functions	0
Actions	10729
Reactions	0
Sequences	0
Randoms	0
RandomAlls	0
RandomPutBacks	0
Decision	10727
Number of vertices	21458
Depth	21
Leafs	10729
Maximal outer degree	2

B.1.5. Pure Sequence

Name	Sequence
TestSuites	0
TestCases	1
Functions	0
Actions	4160
Reactions	0
Sequences	5848
Randoms	0
RandomAlls	0
RandomPutBacks	0
Decision	0
Number of vertices	10009
Depth	9
Leafs	4160
Maximal outer degree	7

B.1.6. Pure Other

Name	Frequency
TestSuites	0
TestCases	1
Functions	0
Actions	4158
Reactions	0
Sequences	0
Randoms	1995
RandomAlls	1920
RandomPutBacks	1935
Decision	0
Number of vertices	10010
Depth	10
Leafs	4158
Maximal outer degree	7

B.1.7. Special Case with 96 Vertices - Special_(100)

Name	Frequency
TestSuites	0
TestCases	13
Functions	20
Actions	8
Reactions	0
Sequences	8
Randoms	12
RandomAlls	24
RandomPutBacks	11
Decision	0
Number of vertices	6
Depth	96
Leafs	1
Maximal outer degree	1

B.1.8. Special Case with 10,450 Vertices - Special_(10000)

Name	Frequency
TestSuites	0
TestCases	1430
Functions	2090
Actions	880
Reactions	0
Sequences	0
Randoms	2200
RandomAlls	2640
RandomPutBacks	1210
Decision	0
Number of vertices	10450
Depth	10450
Leafs	1
Maximal outer degree	1

B.1.9. Small Graph with $g_G^+ = 5$ - 1_(563)

Name	Frequency
TestSuites	0
TestCases	97
Functions	110
Actions	103
Reactions	41
Sequences	55
Randoms	48
RandomAlls	44
RandomPutBacks	49
Decision	16
Number of vertices	563
Depth	12
Leafs	298
Maximal outer degree	5

B.1.10. Huge Graph with $g_G^+ = 5$ - 1_(10139)

Name	Frequency
TestSuites	1
TestCases	1884
Functions	1895
Actions	1839
Reactions	637
Sequences	919
Randoms	927
RandomAlls	886
RandomPutBacks	902
Decision	249
Number of vertices	10139
Depth	18
Leafs	5222
Maximal outer degree	5

B.1.11. Small Graph with $g_G^+ = 50$ - 2_(471)

Name	Frequency
TestSuites	0
TestCases	152
Functions	146
Actions	152
Reactions	5
Sequences	5
Randoms	2
RandomAlls	6
RandomPutBacks	3
Decision	0
Number of vertices	471
Depth	4
Leafs	454
Maximal outer degree	50

B.1.12. Huge Graph with $g_G^+ = 50$ - 2_(7311)

Name	Frequency
TestSuites	1
TestCases	2325
Functions	2286
Actions	2329
Reactions	74
Sequences	81
Randoms	71
RandomAlls	61
RandomPutBacks	82
Decision	1
Number of vertices	7311
Depth	5
Leafs	7011
Maximal outer degree	50

B.1.13. Small Graph with $g_G^+ = 20$ - 3_(670)

Name	Frequency
TestSuites	0
TestCases	231
Functions	194
Actions	178
Reactions	24
Sequences	15
Randoms	15
RandomAlls	14
RandomPutBacks	17
Decision	0
Number of vertices	670
Depth	7
Leafs	603
Maximal outer degree	20

B.1.14. Huge Graph with $g_G^+ = 20$ - 3_(9551)

Name	Frequency
TestSuites	0
TestCases	2808
Functions	2799
Actions	2744
Reactions	229
Sequences	242
Randoms	242
RandomAlls	255
RandomPutBacks	210
Decision	22
Number of vertices	9551
Depth	7
Leafs	8510
Maximal outer degree	20

B.1.15. Small Graph with $g_G^+ = 3$ - 4_(564)

Name	Frequency
TestSuites	1
TestCases	79
Functions	77
Actions	74
Reactions	43
Sequences	66
Randoms	66
RandomAlls	57
RandomPutBacks	75
Decision	26
Number of vertices	564
Depth	24
Leafs	154
Maximal outer degree	3

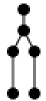
B.1.16. Huge Graph with $g_G^+ = 3$ - 4_(9787)

Name	Frequency
TestSuites	1
TestCases	1275
Functions	1325
Actions	1354
Reactions	705
Sequences	1101
Randoms	1154
RandomAlls	1204
RandomPutBacks	1149
Decision	439
Number of vertices	9787
Depth	36
Leafs	2639
Maximal outer degree	3

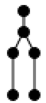
C. Results of Drawings

C.1. Leaf-First-Layering

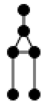
C.1.1. Min Decision



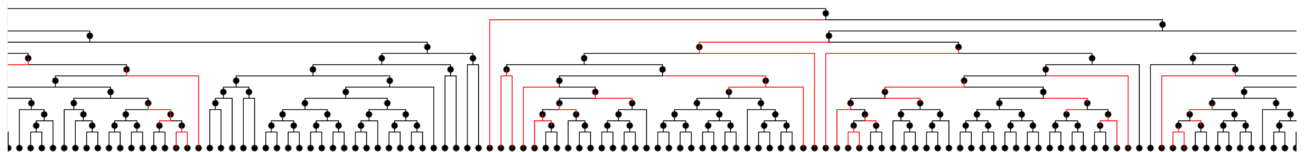
C.1.2. Min Other



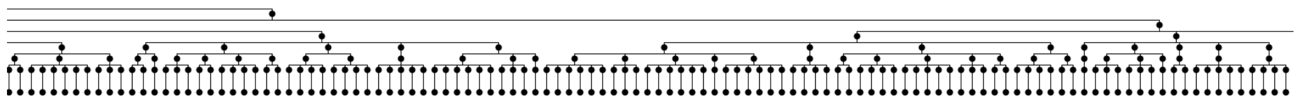
C.1.3. Min Sequence



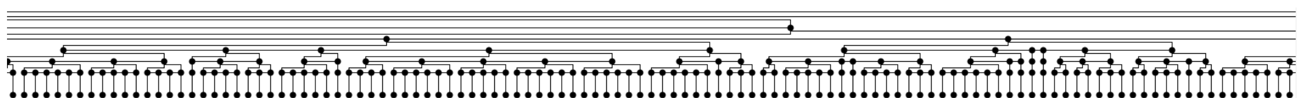
C.1.4. Pure Decision



C.1.5. Pure Other



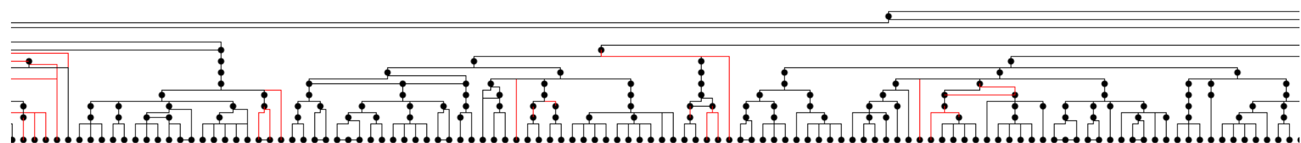
C.1.6. Pure Sequence



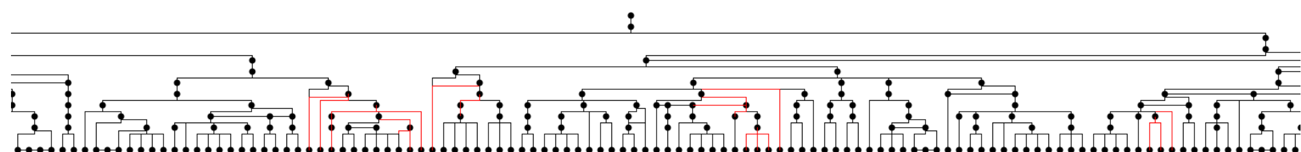
C.1.7. Special(100)

C.1.8. Special(10000)

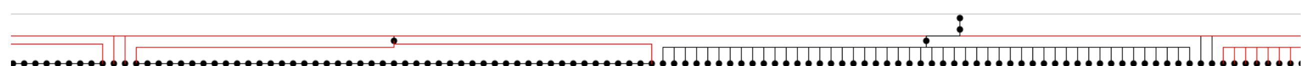
C.1.9. 1_(10139)



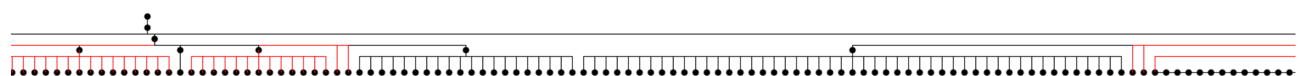
C.1.10. 1_(563)



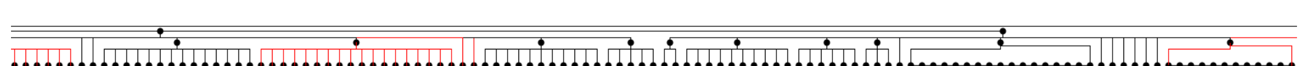
C.1.11. 2_(471)



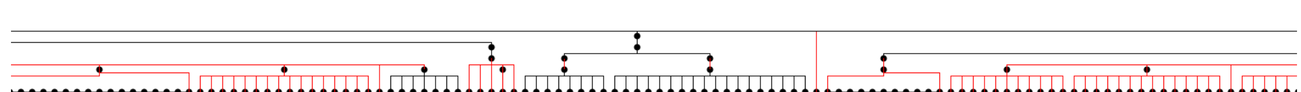
C.1.12. 2_(7311)



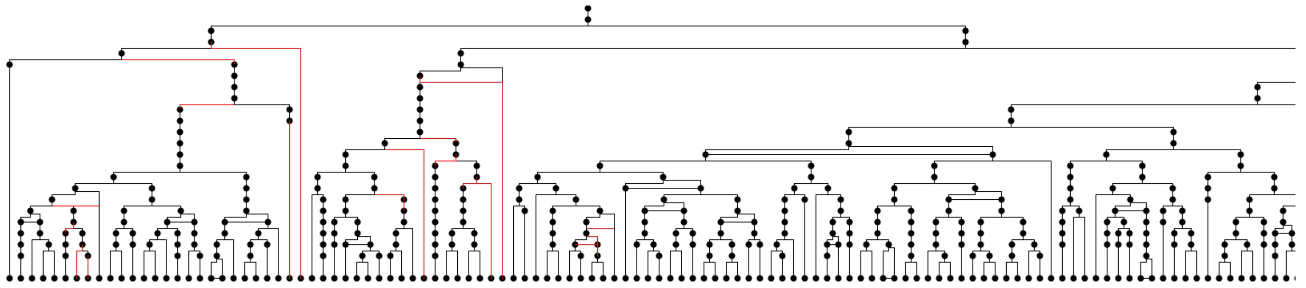
C.1.13. 3_(670)



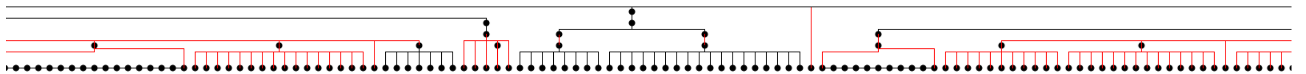
C.1.14. 3_(9551)



C.1.15. 4_(564)

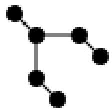


C.1.16. 4_(9787)

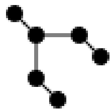


C.2. Dominance Straight Line

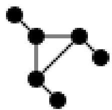
C.2.1. Min Decision



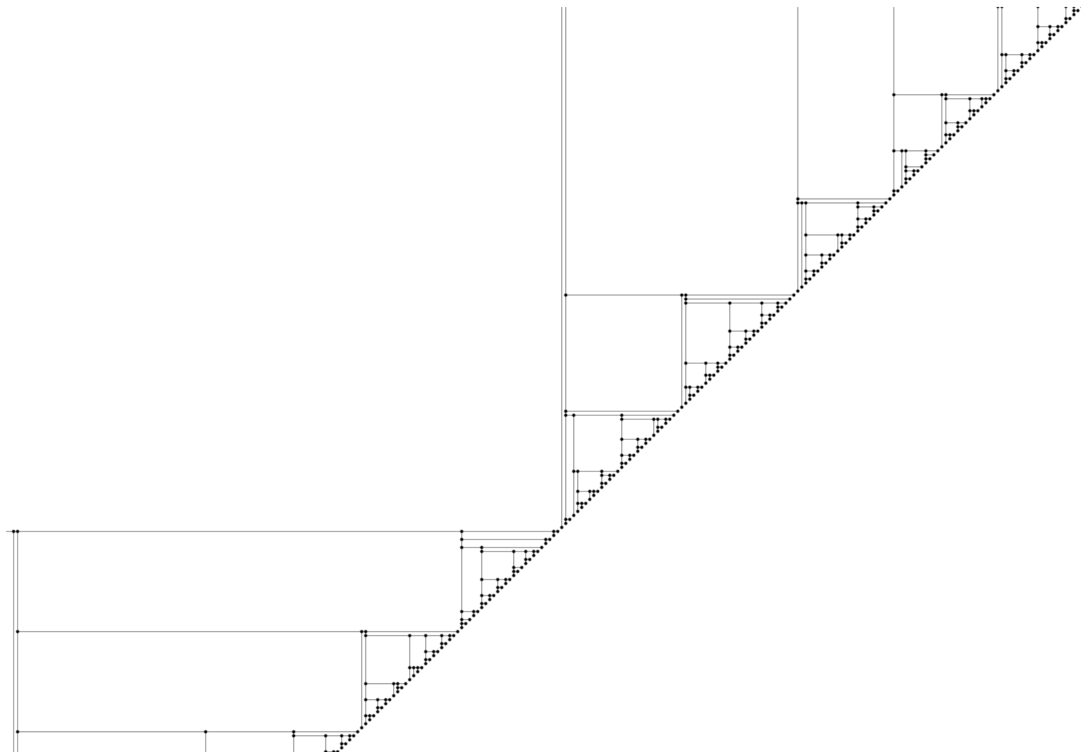
C.2.2. Min Other



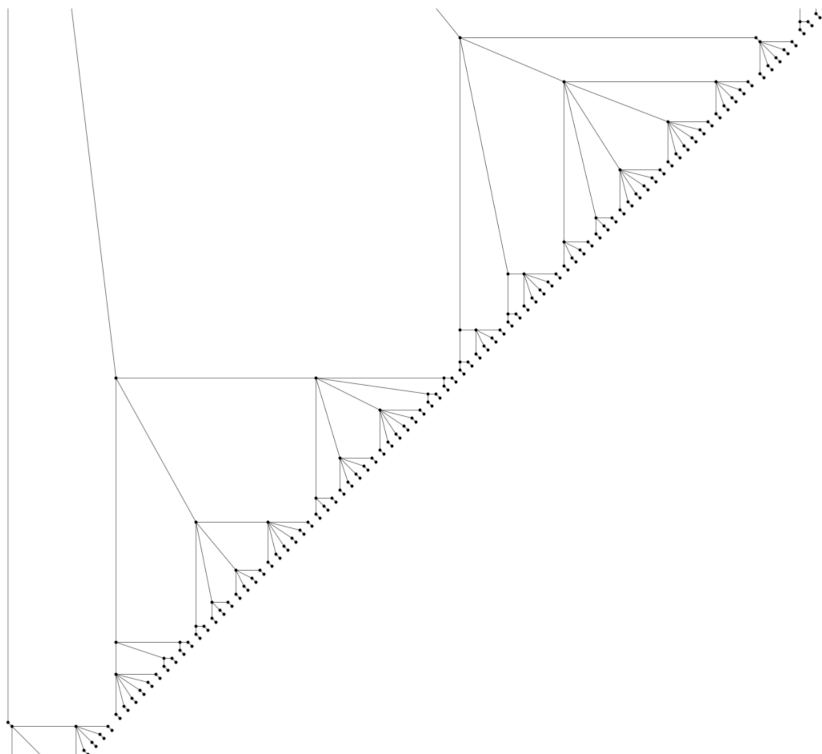
C.2.3. Min Sequence



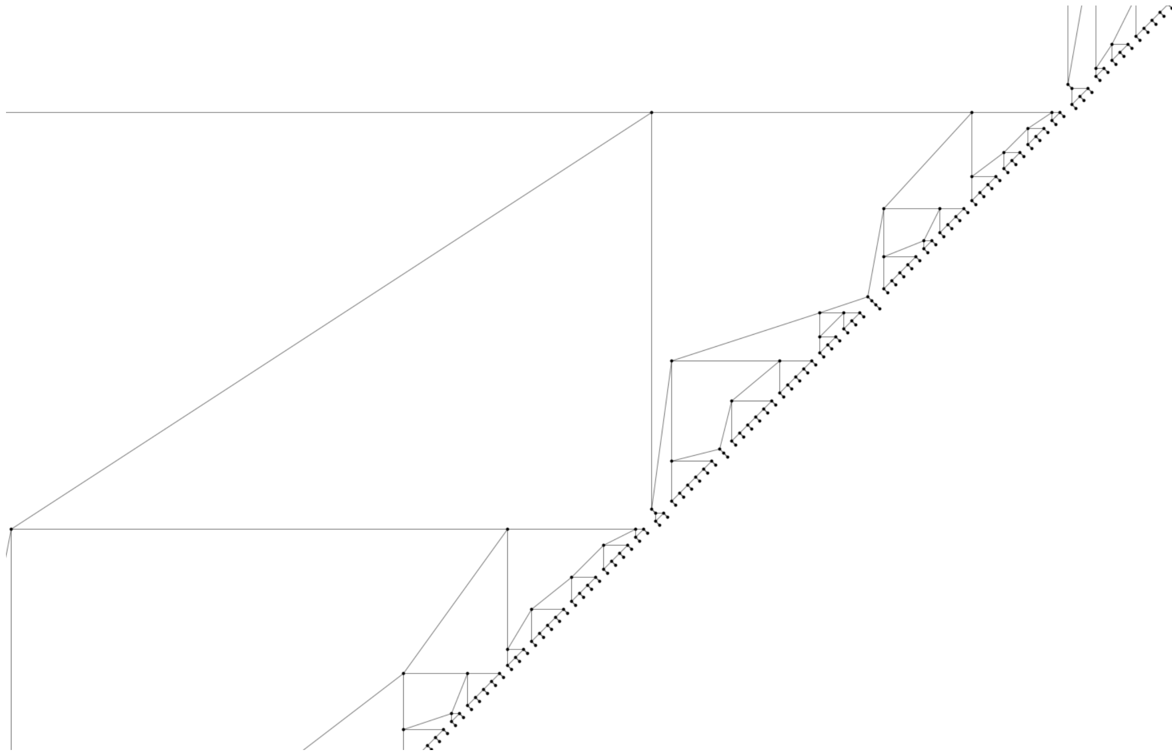
C.2.4. Pure Decision



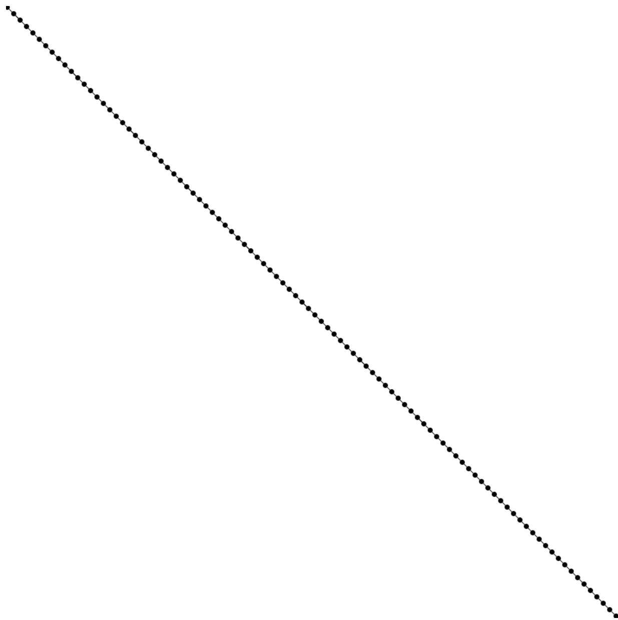
C.2.5. Pure Other



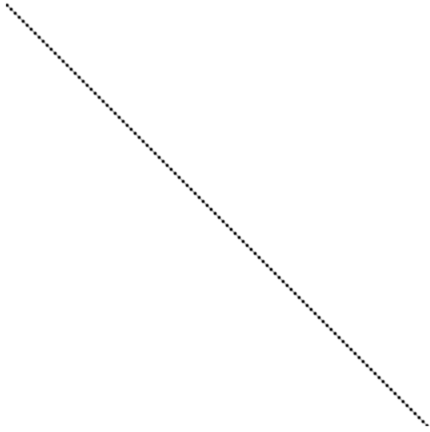
C.2.6. Pure Sequence



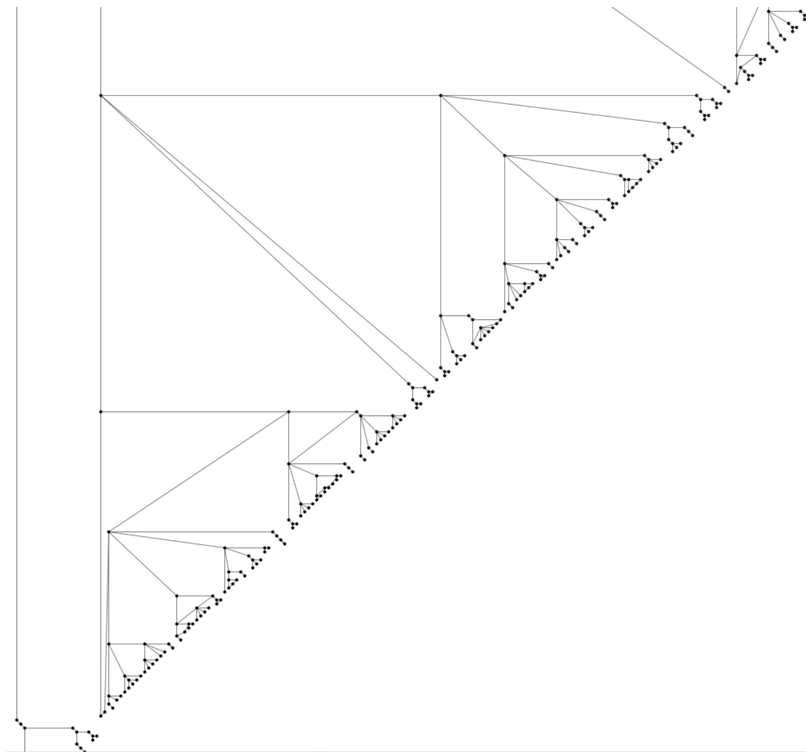
C.2.7. Special(100)



C.2.8. Special(10000)

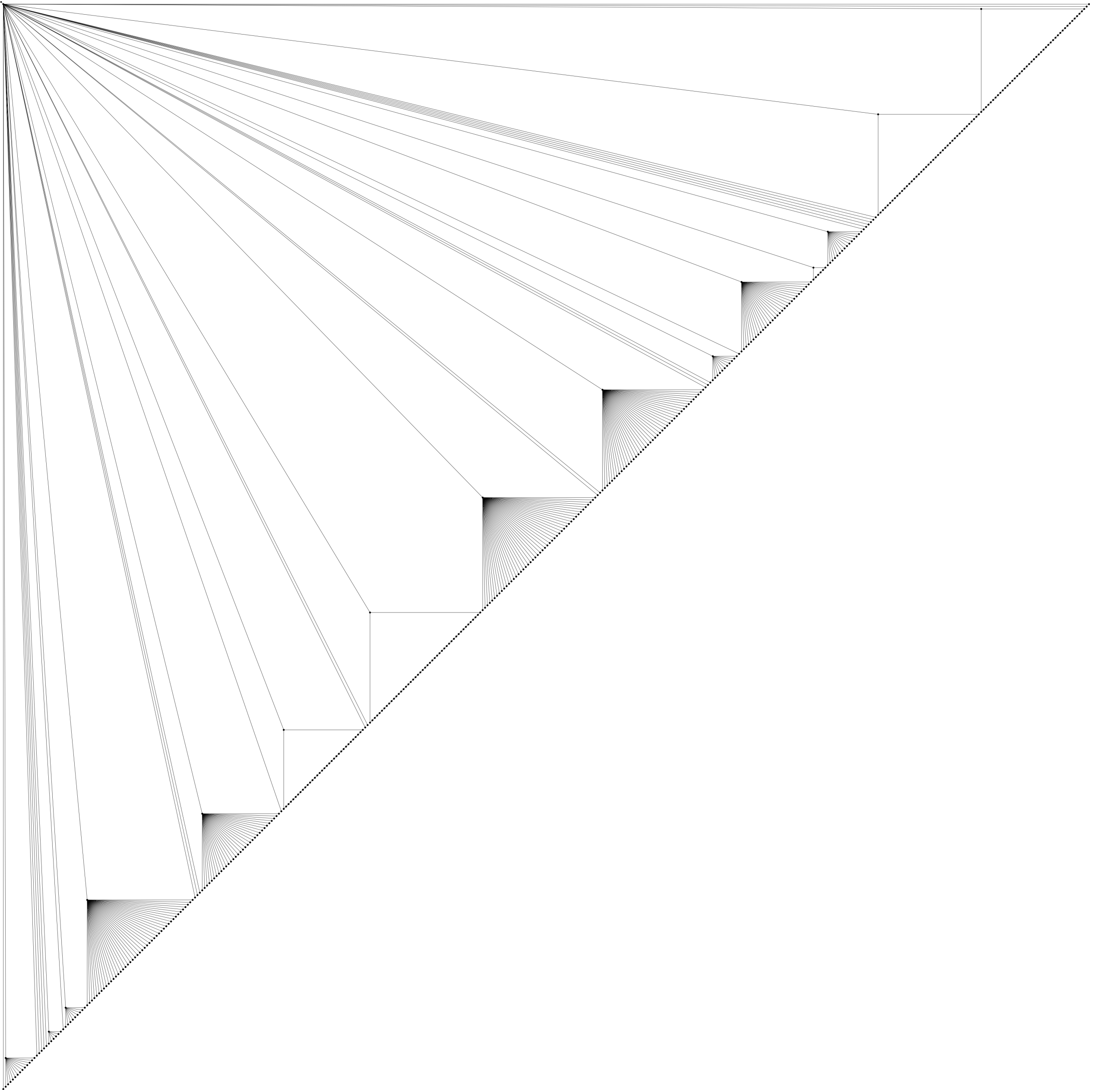


C.2.9. 1_(10139)

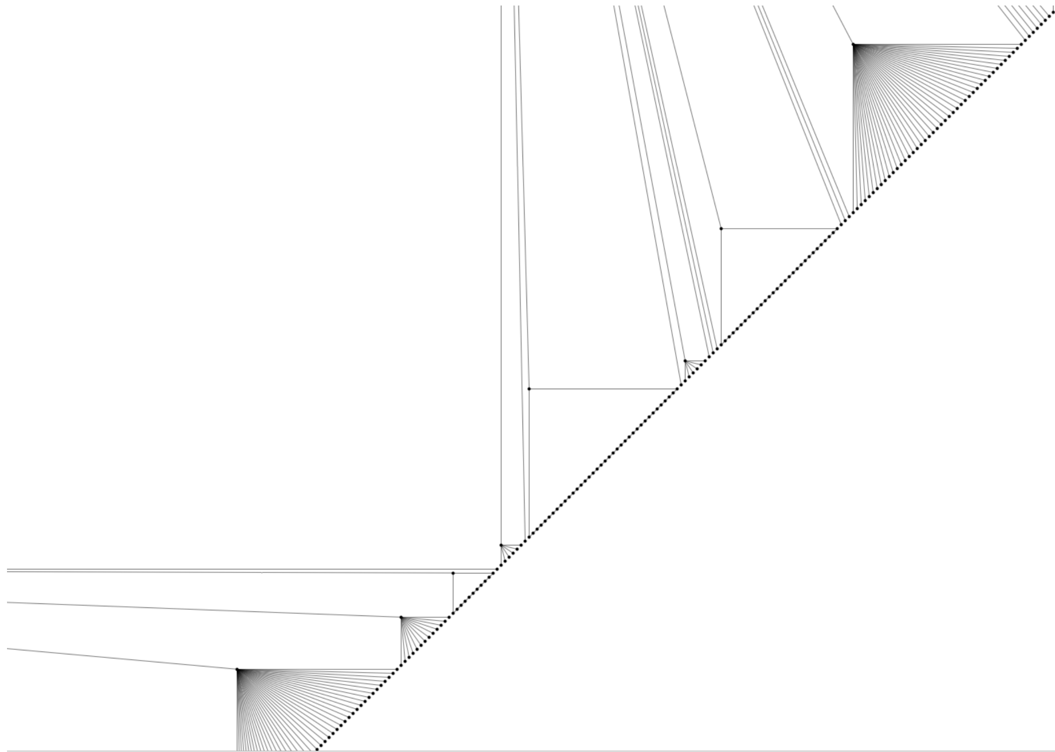


C.2.10. 1_(563)

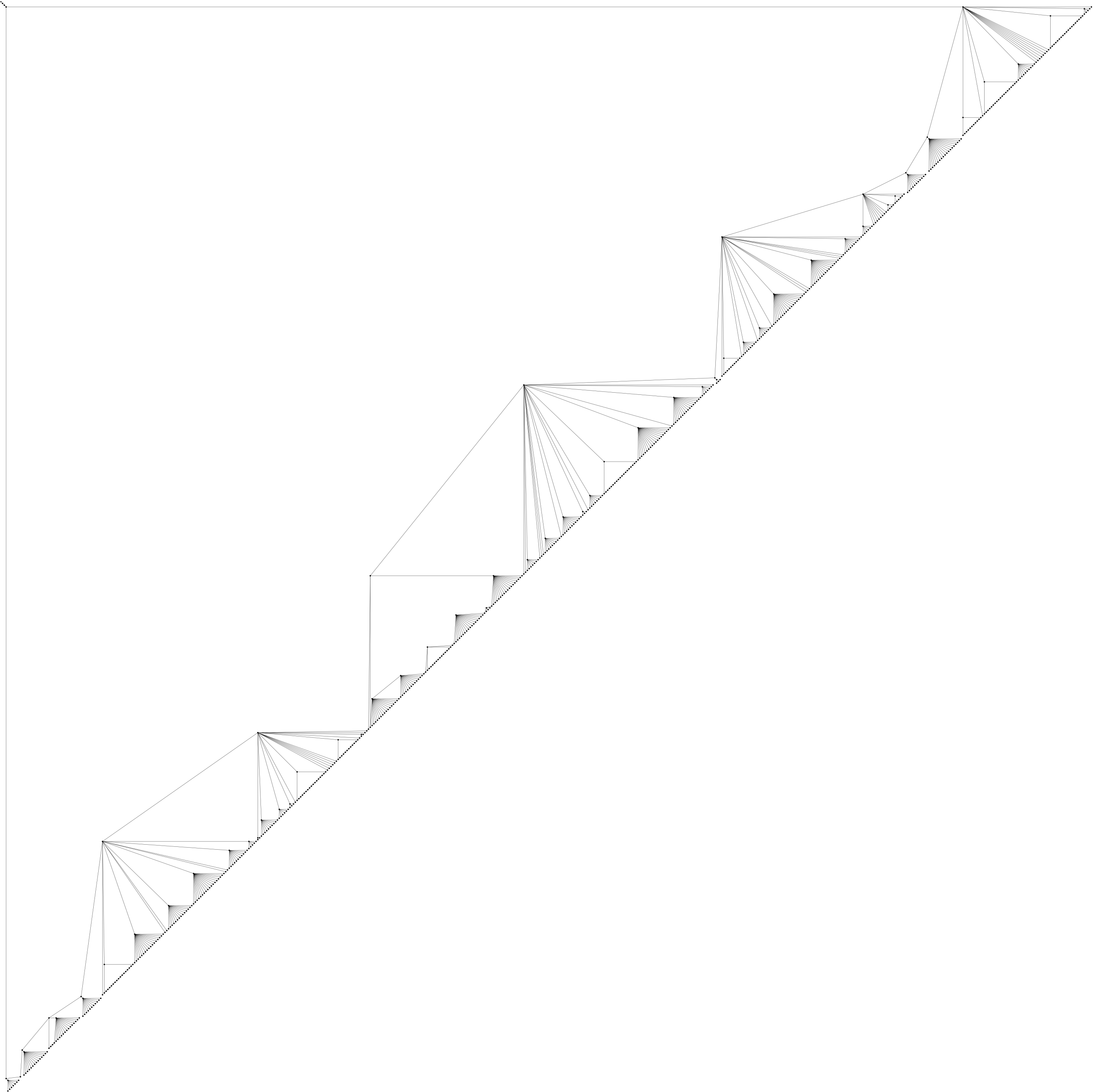
C.2.11. 2_(471)



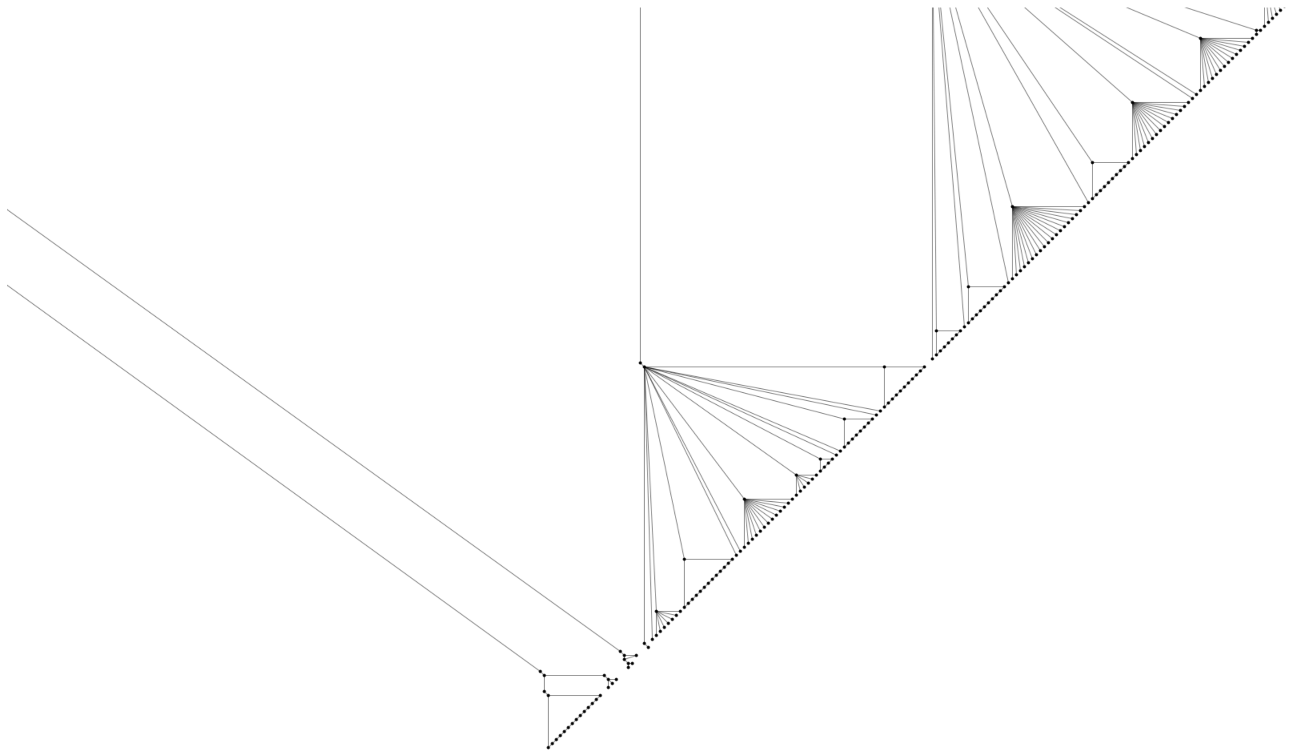
C.2.12. $2_{-}(7311)$



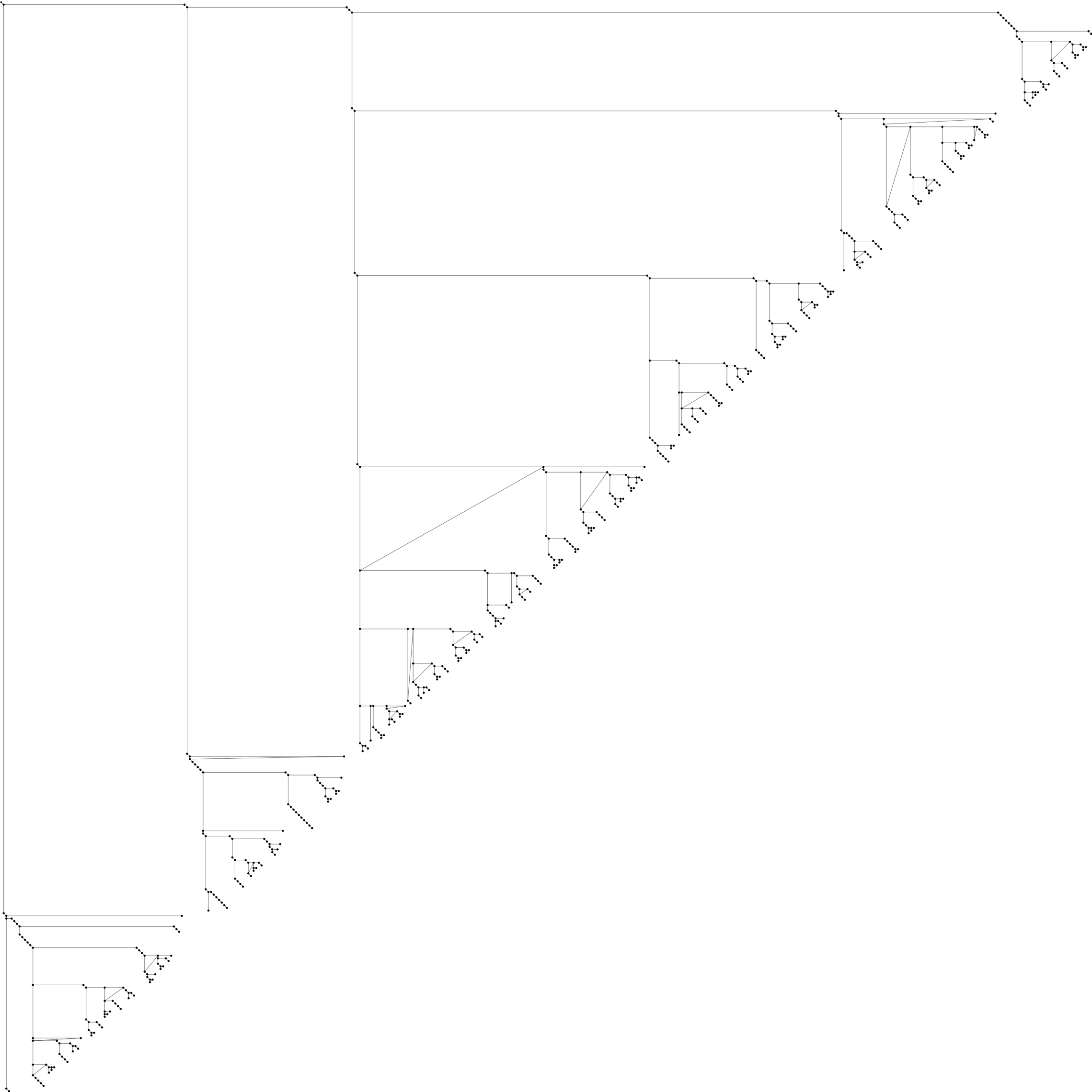
C.2.13. 3_(670)



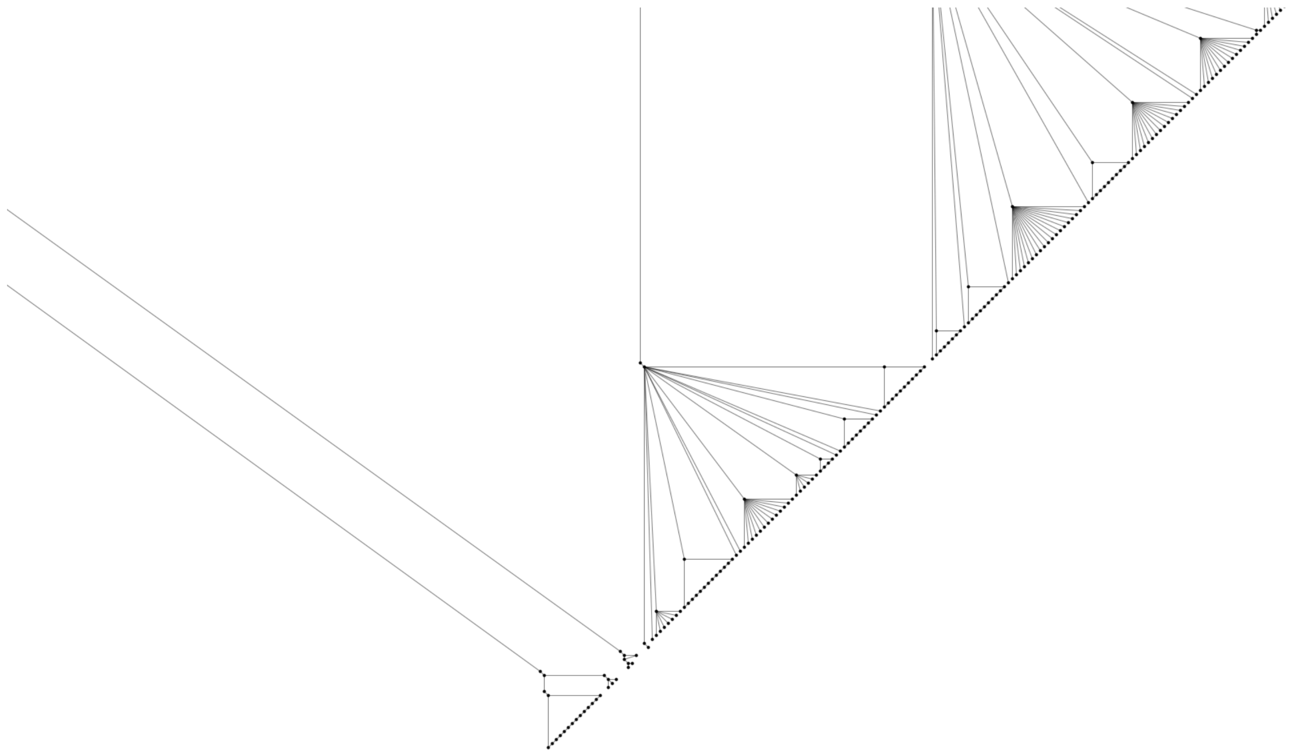
C.2.14. 3_(9551)



C.2.15. 4_(564)

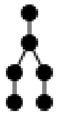


C.2.16. 4_(9787)

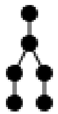


C.3. Improved Walker

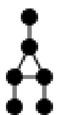
C.3.1. Min Decision



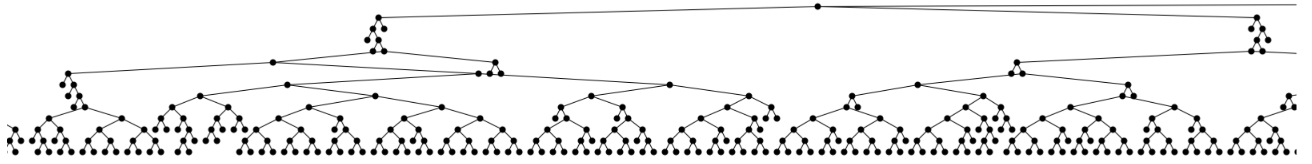
C.3.2. Min Other



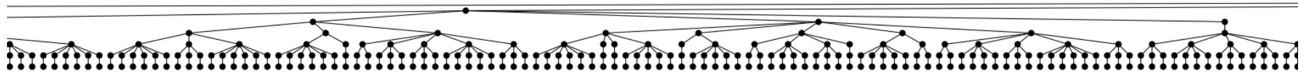
C.3.3. Min Sequence



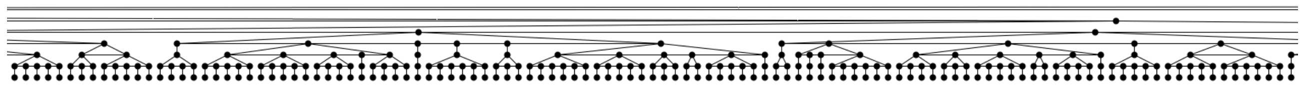
C.3.4. Pure Decision



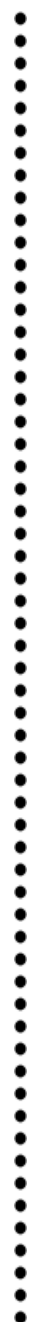
C.3.5. Pure Other



C.3.6. Pure Sequence

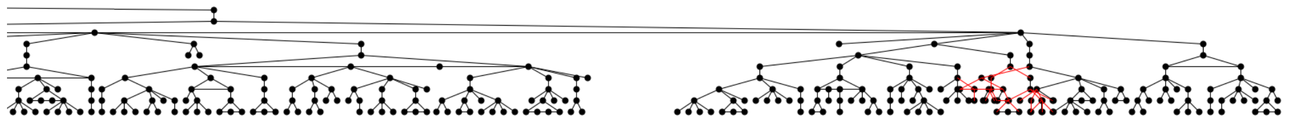


C.3.7. Special(100)

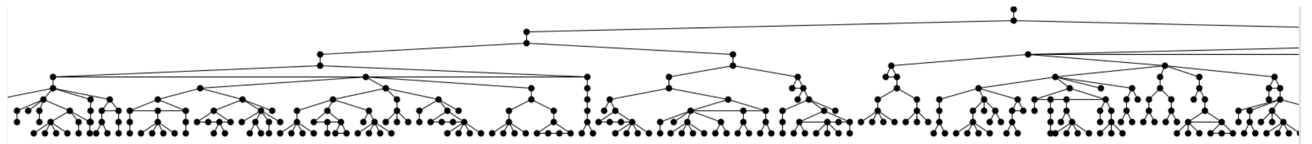


C.3.8. Special(10000)

C.3.9. 1_(10139)



C.3.10. 1_(563)



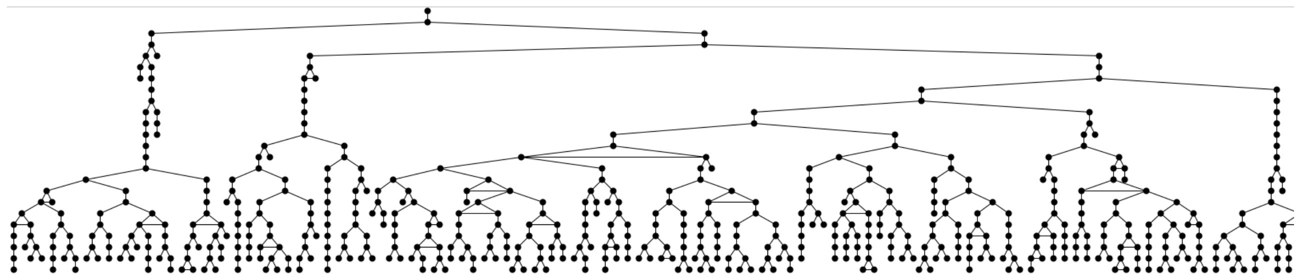
C.3.11. 2_(471)

C.3.12. 2_(7311)

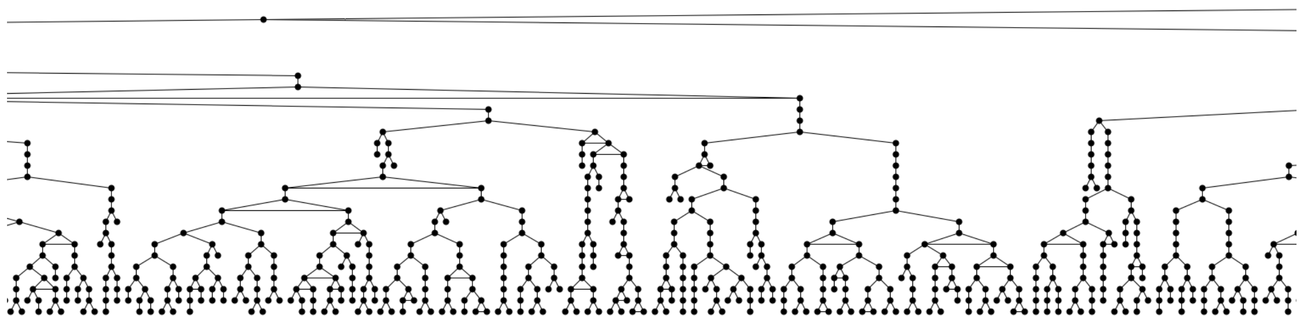
C.3.13. 3_(670)

C.3.14. 3_(9551)

C.3.15. 4_(564)



C.3.16. 4_(9787)



C.4. Magnetic Spring Model - Centric

C.4.1. Min Decision

C.4.2. Min Other

C.4.3. Min Sequence

	Crossings	Area	Length	Diff @ G	Diff at v	Gap
1_(10139)	337183	60774x95617	73564866	13075	13072	0
1_(563)	2345	2747x1948	227015	736	702	0
2_(471)	16347	918x912	8837	686	679	0
2_(7311)	842449	1288x1544	1089623	744	693	0
3_(670)	10177	545x587	24386	267	267	0
3_(9551)	3183044	3713x3441	11785836	2678	2667	0
4_(564)	1169	2811x6449	167837	773	680	0
4_(9787)	194773	107724x90923	44228324	11712	11259	0
Min Decision	0	49x40	85	27	23	8
Min Other	3	26x22	53	20	14	0
Min Sequence	0	26x38	73	26	20	4
Special(100)	7	1174x546	4090	104	20	4
Special(10000)	178687	111180x113986	44639417	13553	4192	9
Sequence	1541598	9351x17853	48592031	8197	3106	0
Decision	973220	285498x163256	211985483	33170	8728	0
Other	771038	12691x23858	55445001	10201	3295	0

Table C.1.: Result of first run of Magnetic Spring Model with a centric magnetic field.

	Crossings	Area	Length	Diff @ G	Diff at v	Gap
1_(10139)	302141	96933x58587	63276038	14525	14394	0
1_(563)	2915	2678x1720	296196	851	850	0
2_(471)	12123	925x892	9394	682	673	0
2_(7311)	1188912	1688x1633	2033260	823	704	0
3_(670)	9802	513x602	90239	240	226	0
3_(9551)	1005872	5562x3891	17927803	3265	3234	0
4_(564)	1099	4124x4627	108275	609	405	0
4_(9787)	228025	88945x109964	43273031	10569	10537	0
Min Decision	0	26x41	63	26	19	4
Min Other	1	36x22	60	21	14	2
Min Sequence	3	26x30	57	30	20	2
Special(100)	5	892x910	3809	78	18	4
Special(10000)	180305	132958x104621	44827449	13520	4010	4
Sequence	723843	12459x18515	61263852	8208	8144	0
Decision	1110118	208627x182590	181701861	28396	7537	0
Other	515414	21687x15775	51463033	9486	3099	0

Table C.2.: Result of second run of Magnetic Spring Model with a centric magnetic field.

	Crossings	**Area**	**Length**	**Diff @ G**	**Diff at v**	**Gap**
1_(10139)	312287	79304x94120	75951369	13668	13512	0
1_(563)	3564	1194x3201	298859	729	703	0
2_(471)	13932	892x887	8451	671	667	0
2_(7311)	742644	1508x1318	2160128	685	661	0
3_(670)	13838	615x465	59731	243	238	0
3_(9551)	2047651	3825x3651	13050471	2590	2582	0
4_(564)	1212	4089x5831	139575	551	515	0
4_(9787)	176586	90034x122747	40491518	14742	14729	0
Min Decision	0	40x43	80	26	13	4
Min Other	1	46x22	69	27	21	4
Min Sequence	3	26x22	53	24	14	0
Special(100)	21	453x884	4257	68	22	5
Special(10000)	181225	114095x112998	45143982	13221	4174	6
Sequence	900785	16943x17712	70712589	9536	9534	0
Decision	777496	199633x230570	214933122	28316	8914	0
Other	434289	30340x20566	78467521	12808	3267	0

Table C.3.: Result of third run of Magnetic Spring Model with a centric magnetic field.

	Crossings	**Area**	**Length**	**Diff @ G**	**Diff at v**	**Gap**
1_(10139)	263580	71575x107087	47479235	14392	14271	0
1_(563)	3844	1313x1718	176481	503	459	0
2_(471)	16347	894x890	8233	673	667	0
2_(7311)	797423	1736x1773	2371806	835	717	0
3_(670)	11980	532x628	37166	274	272	0
3_(9551)	1133224	3306x4707	16238257	2965	2582	0
4_(564)	1105	5593x4016	166150	576	451	0
4_(9787)	191662	90563x109966	42964723	12082	11759	0
Min Decision	3	30x22	55	21	14	0
Min Other	3	26x22	53	20	14	0
Min Sequence	0	26x30	60	20	14	4
Special(100)	13	718x1043	4961	119	27	2
Special(10000)	180133	121988x108203	43843385	12095	4233	9
Sequence	832542	20345x14644	72408674	10024	9995	0
Decision	1082446	143278x251782	230452096	29662	9150	0
Other	531872	18556x27684	64159928	11198	3106	0

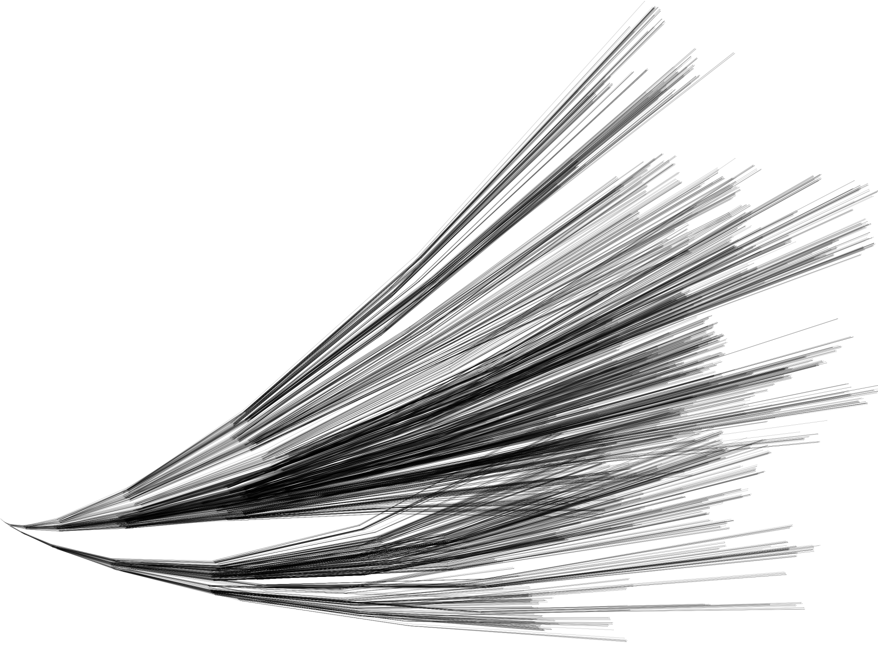
Table C.4.: Result of fourth run of Magnetic Spring Model with a centric magnetic field.

	Crossings	Area	Length	Diff @ G	Diff at v	Gap
1_(10139)	453380	54435x93290	67311788	13438	13191	0
1_(563)	3678	1938x2557	253544	691	657	0
2_(471)	11381	896x892	8411	675	670	0
2_(7311)	861234	1283x1241	10646644	687	681	0
3_(670)	12395	499x476	82831	240	226	0
3_(9551)	1300909	3131x2352	6200340	1500	1498	0
4_(564)	1145	3361x6224	155446	624	609	0
4_(9787)	202935	85401x124959	42785093	12383	11166	0
Min Decision	1	44x22	72	27	16	3
Min Other	0	32x40	64	30	25	4
Min Sequence	1	48x22	81	22	17	4
Special(100)	14	506x971	3957	71	24	4
Special(10000)	179373	109753x131471	44060195	14585	4208	0
Sequence	947483	12862x10941	37215735	6392	6390	0
Decision	1070415	224609x161256	206947756	27640	7975	0
Other	581853	18324x31241	64670622	11194	2820	0

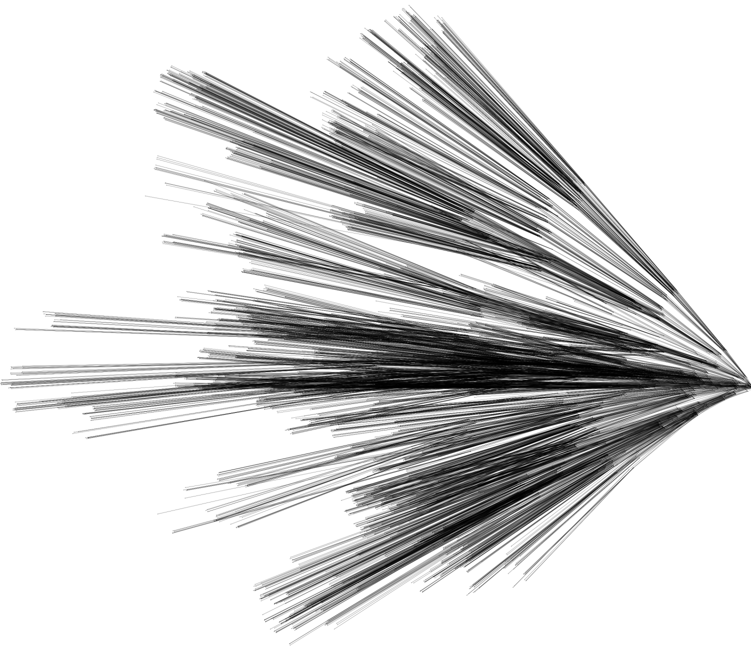
Table C.5.: Result of fifth run of Magnetic Spring Model with a centric magnetic field.

C.4.4. Pure Decision

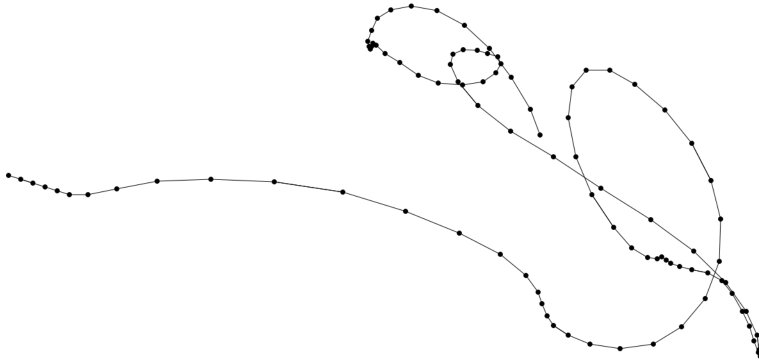
C.4.5. Pure Other



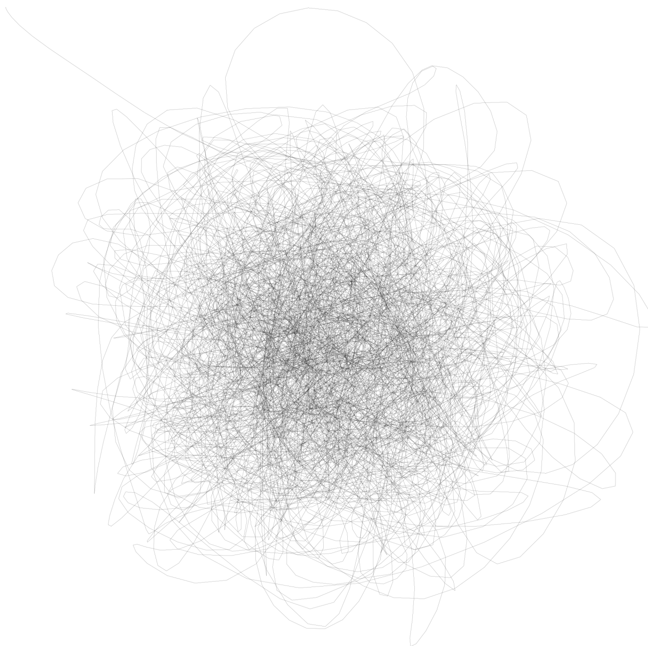
C.4.6. Pure Sequence



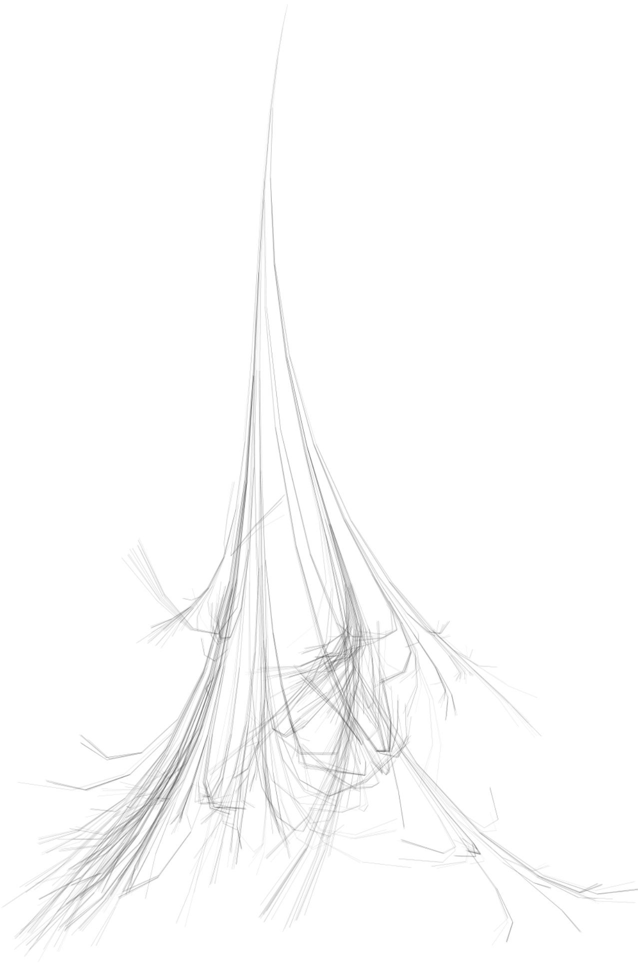
C.4.7. Special(100)



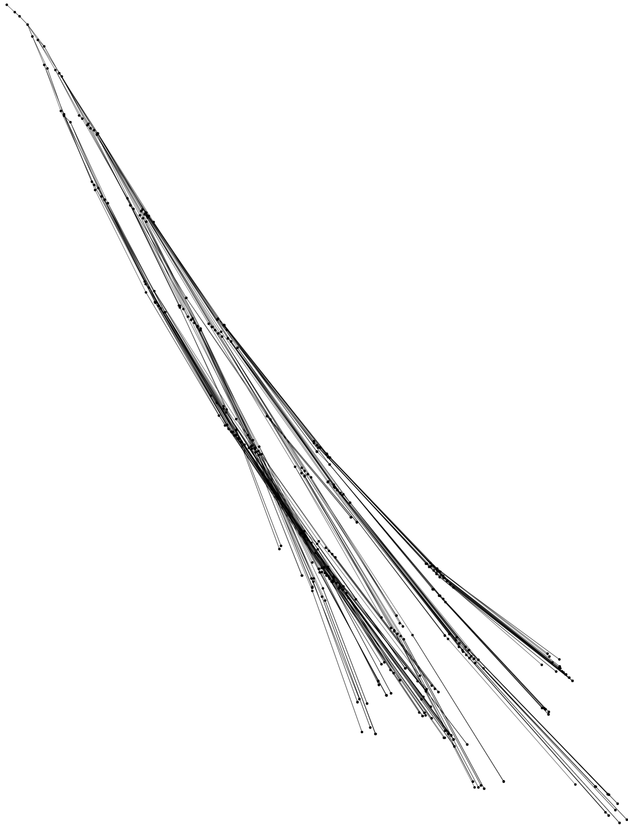
C.4.8. Special(10000)



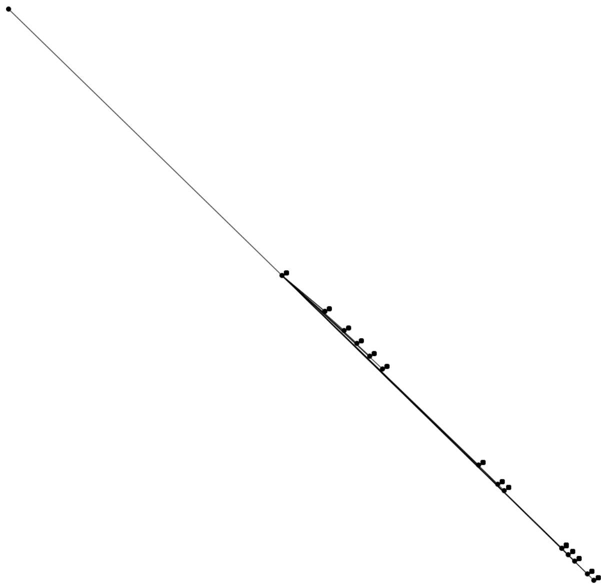
C.4.9. 1_(10139)



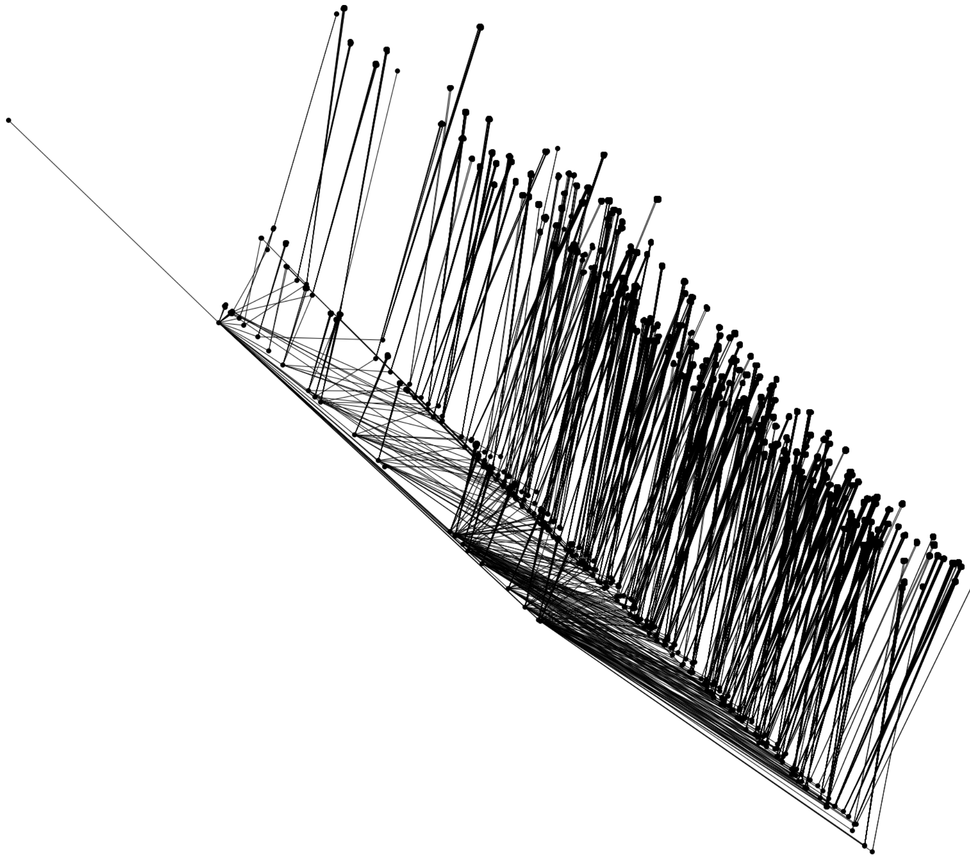
C.4.10. 1_(563)



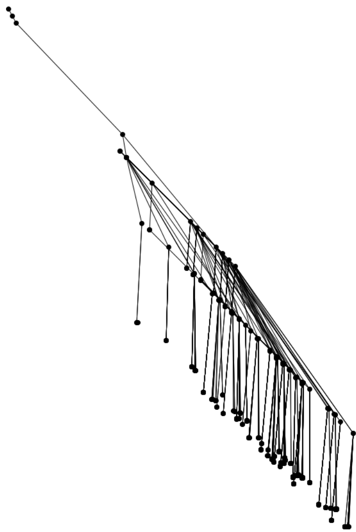
C.4.11. 2_(471)



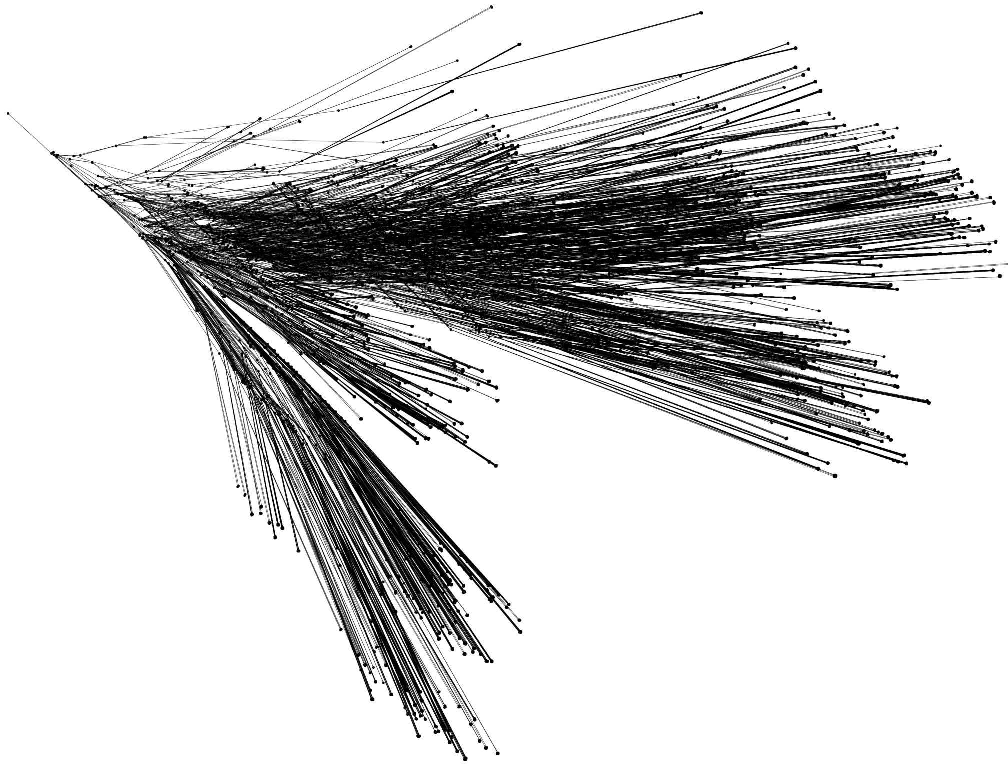
C.4.12. 2_(7311)



C.4.13. 3_(670)

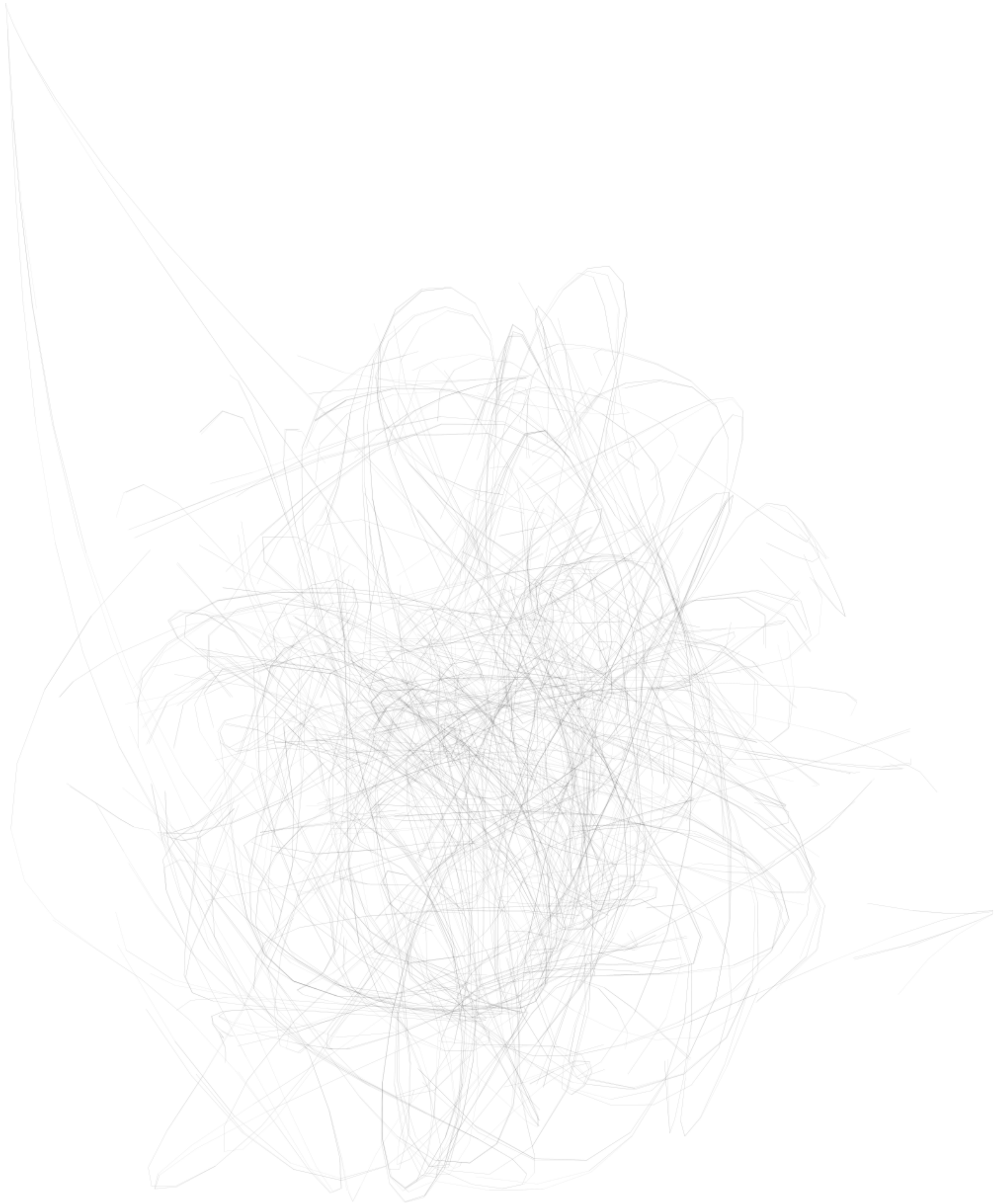


C.4.14. 3_(9551)



C.4.15. 4_(564)

C.4.16. $4_{-}(9787)$



D. Definitions

Directed Graph (digraph) A directed graph is a quadruple

$$\begin{aligned} &G = (V, R, \alpha, \omega) \quad \text{with} \\ &V \neq \emptyset, V \cap R = \emptyset, \alpha : R \to V, \omega : R \to V \end{aligned} \tag{D.1}$$

G = (directed) graph
V = Set of vertices
R = Set of edges
α and ω are mappings
α(r) = beginning edge
ω(r) = ending edge
$n :=\ |V(G)|$ = number of vertices
$m := |R(G)|$ = number of edges
$\delta^+_G(v) := \{r \in R : \alpha(r) = v\}$ = outgoing edges of v
$\delta^-_G(v) := \{r \in R : \omega(r) = v\}$ = incoming edges of v
$N^+_G(v) := \{\omega(r) : r \in \delta^+_G(v)\}$ = set of vertices following v
$N^+_G(v) := \{\alpha(r) : r \in \delta^-_G(v)\}$ = set of vertices preceding v
$g^+_G(v) := |\delta^+_G(v)|$ = outer degree of v
$g^-_G(v) := |\delta^-_G(v)|$ = inner degree of v
$g_G(v) := g^+_G(v) + g^-_G(v)$ = degree of v
$\Delta G := max\{g_G(v) : v \in V\}$

Undirected graph An undirected graph is a triple

$$\begin{aligned} G \quad &= (V, E, \gamma) \quad \text{with} \\ V \quad &\neq \emptyset, V \cap E = \emptyset, \\ \gamma : \quad &E \to \{X : X \subseteq V \ \text{ with } \ 1 \leq |X| \leq 2\} \end{aligned} \tag{D.2}$$

$$\begin{aligned} G &= \text{Graph} \\ V &= \text{Set of vertices} \\ E &= \text{Set of edges} \\ \gamma &= \text{Set of one or two vertices, defining an edge} \\ \delta(v) &:= \{e \in E : v \in \gamma(e)\} \text{ incident edges of } v \\ N_G(v) &:= \{u \in V : {}^{\gamma(e)=\{u,v\}}_{\text{for } e \in E}\} \text{ vertices adjacent to } v \\ g_G(v) &:= \sum_{e \in E : v \in \gamma(e)} (3 - |\gamma(e)|) \text{ degree of } v \\ \Delta(G) &:= max\{g_G(v) : v \in V\} \text{ maximum degree of } G \end{aligned} \tag{D.3}$$

Incidence Two edges r and r' are called *incident* if

$$\begin{aligned} \exists \quad v \in V; \quad r, r' \in R \quad | \quad \alpha(r) &= \alpha(r') \quad \text{or} \\ \alpha(r) &= \omega(r') \quad \text{or} \\ \omega(r) &= \alpha(r') \quad \text{or} \\ \omega(r) &= \omega(r') \end{aligned} \tag{D.4}$$

Adjacence Two edges v and v' are adjacent if

$$\begin{aligned} \exists \quad r \in R; \quad v, v' \in V \quad | \quad \alpha(r) = v \quad \text{and} \quad \omega(r) &= v' \quad \text{or} \\ \alpha(r) = v' \quad \text{and} \quad \omega(r) &= v \end{aligned} \tag{D.5}$$

Isomorphic Two graphs $G(V, R, \alpha, \omega)$ and $G'(V', R', \alpha', \omega')$ are isomorphic ($G \cong G'$) if there are two bijective mappings:

$$\begin{aligned} \sigma : V \to V'; \\ \tau : R \to R'; \\ r \in R \quad | \\ (i) \quad \alpha(\tau(r)) &= \sigma(\alpha(r)) \\ (ii) \quad \omega(\tau(r)) &= \sigma(\omega(r)) \end{aligned} \tag{D.6}$$

Partial graph / Superior graph A graph $G' = (V', R', \alpha', \omega')$ is called *partial graph* ($G' \sqsubseteq G$) of $G = (V, R, \alpha, \omega)$ if

$$\begin{aligned} (i) \qquad & V' \subseteq V \quad \text{and} \quad R' \subseteq R \\ (ii) \qquad & \alpha|_{R'} = \alpha' \quad \text{and} \quad \omega|_{R'} = \omega' \end{aligned} \tag{D.7}$$

(Self) loop An edge is called a loop if

$$r \in R | \alpha(r) = \omega(r) \tag{D.8}$$

Path A path P in a graph G is:

$$\begin{aligned} P &= (v_0, r_1, v_1, ..., r_k, v_k) \text{ with} \\ k &\geq 0, v_0, ..., v_k \in V(G), r_1, ..., r_k \in R(G), \\ \alpha(r_i) &= v_{i-1}, \omega(r_i) = v_i \text{ for } i = 1, ..., k \end{aligned} \tag{D.9}$$

Beginning and ending vertex are defined as follows:

$$\begin{aligned} \alpha P &:= v_0 \\ \omega P &:= v_k \end{aligned} \tag{D.10}$$

Cyclic graph The partial graph G' is a cycle if

$$\begin{aligned} P &= v_0, ..., v_{k-1} \text{ with } k \leq 3 \quad \text{then} \\ G' &:= P + r \text{ with } \alpha(r) \; = \; v_{k-1}, \omega(v_0) \end{aligned} \tag{D.11}$$

A cycle can also be defined via its vertices:

$$G' := (v_0, ..., v_{k-1}, v_0) \text{ with } k = n(G') \tag{D.12}$$

Acyclic graph A graph is acyclic if

$$\nexists P | \alpha P \; = \omega P \tag{D.13}$$

Parallel edge Edges are considered parallel if

$$r, r' \in R, r \neq r' \quad | \quad \alpha(r) = \alpha(r') \quad \text{and} \quad \omega(r) = \omega(r') \tag{D.14}$$

Antiparallel / inverse edge Edges are antiparallel or inverse if

$$r, r' \in R, r \neq R' \quad | \quad \alpha(r) = \omega(r') \quad \text{and} \quad \omega(r) = \alpha(r') \tag{D.15}$$

Simple graph A graph is called simple if it has no loops and no parallel edges.

Reachability An edge w is reachable from vertex v if

$$\exists P | \alpha(P) = v \; \wedge \; \omega(P) = w \tag{D.16}$$

$E_G(v)$ means all the vertices reachable from v.

Connected Two vertices are v,w are called connected ($v \leftrightarrow w$) if

$$\begin{aligned} v, w &\in V(G) \\ w &\in E_G(v) \wedge v \in E_G(w) \end{aligned} \tag{D.17}$$

Connected graph A graph is connected if

$$\forall v, w \in V \ \exists \ w \in E_G(v) \ \wedge \ v \in E_G(w) \tag{D.18}$$

Transitive Graph A graph $G = (V, R, \alpha, \omega)$ is transitive if:

$$\begin{aligned} r_{uv}, r_{vw} &\in R \text{ with} \\ \omega(r_{uv}) &= \alpha(r_{vw}) \text{ then} \\ \exists r \in R &\mid \alpha(r) = \alpha(r_{uv}) \wedge \omega(r) = \omega(r_{vw}) \end{aligned} \tag{D.19}$$

Transitive Closure A Graph $G^* = (V, R^*)$ is the transitive closure of a graph without parallel edges G=(V,R) if:

$$\begin{aligned} (i) &\quad G \sqsubseteq G^* \\ (ii) &\quad G^* \text{ is transitive} \\ (iii) &\quad if G' \text{ is transitive and } G \sqsubseteq G' \text{ then } G^* \sqsubseteq G' \end{aligned} \tag{D.20}$$

From these conditions, it follows that the transitive closure G^* from a graph without parallel edges G is distinctly defined (see [8] pages 79 - 83 for the proof).

Forest A graph G is called *forest* if it is simple and acyclic.

Tree A graph G is called *tree* if it is connected. A vertex v with $g_G^+(v) = 0$ is called a *leaf*.

Spanning tree A partial graph $G'(V, E', \gamma)$ is called spanning tree of a graph $G(V, E, \gamma)$, if G' is a tree.

Rooted tree A graph G is called a *rooted tree* if it is a tree and has a special vertex $r \in V$ which is called *root*. The root is the only vertex of the tree which has the inner degree $g_G^- = 0$. Every other vertex has $g_G^- = 1$. ([7], p. 47)

Binary tree A *binary* tree is a rooted tree G with $g_G^+(v) \leq 2 \quad \forall v \in V$.

Drawing A drawing Γ is a function mapping each $v \in V$ to a definite point $\Gamma(v)$ and each edge (u, v) to a connection $\Gamma(u, v)$ with the starting point $\alpha(\Gamma_{(v)})$and the ending point $\omega(\Gamma_{(v)})$ ([4]).

Planar A graph is planar if no two distinct edges intersect ([4]).

Cycle root A cycle root (c) is the ending vertex of the shortest path (Q) out of the set of paths between the graphs root (s) and the vertices of the cycle (C).

$$\begin{aligned} (i) &\quad C \sqsubseteq G|\alpha C = \omega C \\ (ii) &\quad s \in V(G) \wedge g_G^-(s) = 0 \\ (iii) &\quad O = \{P \sqsubseteq G|\alpha P = s \wedge \omega P \in V(C)\} \\ (iv) &\quad c = \omega Q \text{ with } Q \in O|min\{|V(Q)|\} \end{aligned} \tag{D.21}$$

Cycle vertices Cycle vertices (v) refer to all vertices of a cycle (C).

$$v \in V(C) \tag{D.22}$$

Cycle sub vertices Every vertex (v) except the cycle root (c) of a cycle (C) is part the set "cycle sub vertices" (F).

$$\begin{aligned} F(C) &\subset V(C) \\ v &\in F(C) \end{aligned} \tag{D.23}$$

E. CD

F. Affidavit

Affidavit

I confirm that I have produced this thesis by myself and without using resources other than those mentioned. All ideas taken directly or indirectly from other publications are indicated. At my best knowledge, there was no thesis publicized or submitted to any examination authority before which is equal or similar to this thesis.

Erklaerung

Ich erklaere hiermit, dass ich die vorliegende Arbeit selbstaendig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt uebernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde nach meiner besten Kenntnis bisher in gleicher oder aehnlicher Form keiner anderen Pruefungsbehoerde vorgelegt und auch noch nicht veroeffentlicht.

Hof, den 19.06.2007

Bibliography

[1] Batini, C., Nardelli, E. and Tamassia, R.: A Layout Algorithm for Data-Flow Diagrams, IEEE Trans. Softw. Eng., SE-12, no. 4, 538-546, 1986.

[2] Buchheim[1], Christoph, Jnger[1], Michael, Sebastian Leipert[2]: Improving Walker's Algorithm to Run in Linear Time, [1]University of Cologne, [2] cesar research center Bonn, Germany, 2002.

[3] Carpano, M. J.: Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysts, IEEE Trans. Syst. Man Cybern., SMC-10, no 11, pages 705-715, 1980.

[4] Di Battista, Giuseppe, Eades, Peter, Tamassia, Roberto, Tollis, Ioannis G.: Graph Drawing, Algorithms for the visualization of Graphs, 1999.

[5] Kamada, T.: On visualization of abstract objects and relations, A Dissertation Submitted to the Graduate School of the University of Tokyo, 1988.

[6] Kaufmann, Michael and Wagner, Dorothea (Eds.): Drawing Graphs, Methods and Models, Springer Verlag, 1998.

[7] Kaufmann, Michael and Wagner, Dorothea: Drawing Graphs: Methods and Models, Springer-Verlag Inc., 2001.

[8] Krumke, Sven Oliver, Noltemeier, Hartmut: Graphentheoretische Konzepte und Algorithmen, 1. Auflage, 2005.

[9] Laughton, Craig: http://gooseania.blogspot.com/2005/02/graphs-and-trees.html, 30.01.2007.

[10] Reingold, Edward M. and Tilford, John S. Tidier drawings of trees. IEEE Transactions on Software Engineering, 7(2): 223-228, 1981.

[11] Sedgewick, Robert: Algorithms in C++, Pearson Studium, 2002, page 325-327.

[12] Sugiyama, Kozo: Graph Drawing and Applications, For Software and Knowledge Engineers Vol 11, 2002.

[13] Sugiyama, K., Tagawa, S. and Toda, M.: Methods for Visual Understanding of Hierarchical Systems, IEEE Trans. Syst. Man Cybern., SMC-11, no. 2, pages 109-125, 1981.

[14] Tamassia, R. : On embedding a Graph in the Grid with the minimum Number of Bends, SIAM J. Comput., 16. no. 3, 421-444, 1987.

[15] Tamassia, R., Di Battista,G. and Batini, C.: Automatic Graph Drawing and Readability of Diagrams, IEEE Trans. Syst. Man Cybern., SMC-18, no. 1, page 61-78, 1988.

[16] US-Patent 5705791. http://www.patentstorm.us/patents/5705791-claims.html; 2006-06-05.

[17] Walker II, John Q. . A node-positioning algorithm for general trees. Software Practice and Experience, 20(7): 685-705, 1990.

[18] Warfield, J. : Crossing Theory and Hierarchy Mapping, IEEE Trans. Syst. Man Cybern., SMC-7, no. 7, pages 502-523, 1977.

[19] Wetherell, Charles and Shannon, Alfred. Tidy drawing of trees. IEEE Transactions on Software Engineering, 5(5): 514:514-520, 1979.

[20] Wikipedia (German); http://de.wikipedia.org/wiki/Diagramm; 01.03.2006.