UNIVERSITÄT ZU LÜBECK

Diplomarbeit

Configurable Graph Drawing Algorithms for the Ti*k*Z Graphics Description Language

Jannis Pohlmann

14. Oktober 2011

Institut für Theoretische Informatik
Prof. Dr. Till Tantau

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Zudem versichere ich, dass ich weder diese noch inhaltlich verwandte Arbeiten als Prüfungsleistung in anderen Fächern eingereicht habe oder einreichen werde.

Lübeck, den 14. Oktober 2011

_____

Abstract

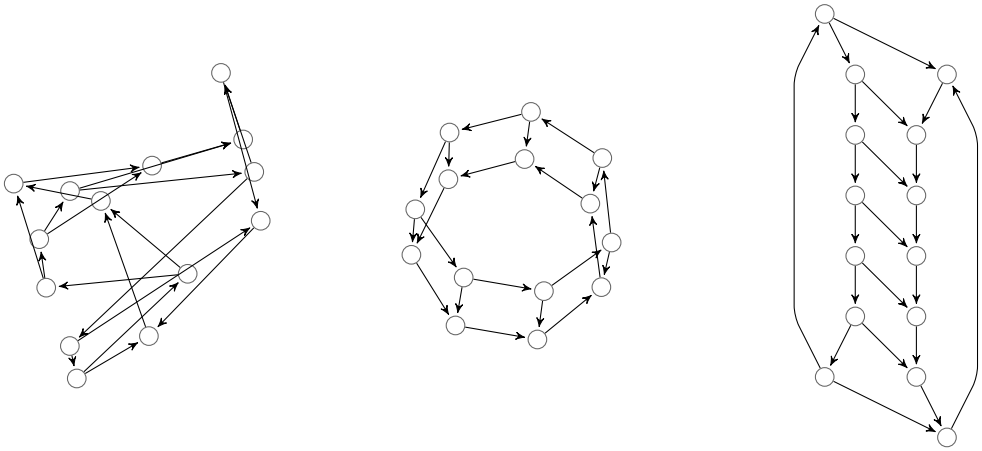
Graphs are key to illustrating and understanding many complex systems and processes. This thesis provides an introduction to the graph drawing features of Ti*k*Z, a versatile language for creating a wide range of vector graphics that includes a framework for automatic graph layout strategies written in the Lua programming language. Algorithms for two families of drawing methods—spring and spring-electrical layouts of general graphs as well as layered drawings of directed graphs—were developed specifically for Ti*k*Z. They are presented along with a discussion of their underlying concepts, implementation details, and options to select, configure, and extend them. The generated spring and spring-electrical layouts feature a high degree of symmetry, evenly distributed nodes and almost uniform edge lengths. The techniques developed for layered drawings are successful in highlighting the flow of directed graphs and producing layouts with uniform edge lengths and few crossings. By discussing many graph drawing features valuable for end-users and researchers alike, this thesis aims at enabling a comparison to related tools and providing the groundwork for future research and algorithm development.

Contents

1

Introduction

Consider the following three drawings:



These drawings show the same graph. Clearly, the two graphics on the right are more appealing than the one on the left and convey more information about the structure of the graph. What causes us to make this distinction between good and bad or informative and non-informative are aesthetic aspects such as symmetry, distribution of nodes, or edge crossings. Graphs are not only a formal structure to model relational data, but are also key to illustrating and understanding many complex systems and processes in the form of graph drawings. Three possible applications are the visualization and analysis of social networks, the documentation of component hierarchies in software engineering, and the illustration of the control flow of a machine or production plant. Due to their size and for reasons of convenience, drawings of graphs are

often generated automatically with the help of specialized software. The quality of the automatic layout techniques used in these tools and the compliance with common aesthetic criteria are critical for producing graph drawings that are visually appealing and easy to understand.

This is the central focus of *graph drawing*, an area of mathematics and computer science that addresses the problem of visualizing graphs by developing automatic layout strategies and evaluating these solutions based on aesthetic criteria. It combines elements from geometric graph theory and information visualization and plays an important role in data analysis as well as the production of research papers, books, lecture notes, and presentations where drawings of graphs are needed.

Graph drawing is an active field of research with its own annual conference, the *International Symposium on Graph Drawing* [5], first held in 1995. Since its early days in the 1960s, graph drawing has succeeded in establishing many concepts related to the visualization of graphs. In addition to the identification and evaluation of common aesthetic criteria such as the minimization of edge crossings or the maximization of symmetry, a number of standard strategies and algorithm frameworks for drawing graphs have been developed. They are often classified by their underlying concepts, the types of graphs they accept as their input, and the drawing conventions they implement. Popular classifications include methods based on the simulation of physical forces, methods for drawing trees or directed graphs, and methods that produce orthogonal, planar or layered drawings. An introduction to the most important drawing conventions, constraints, and aesthetic criteria as well as a detailed overview of popular techniques for drawing graphs is given by Di Battista et al. [12].

Graph drawing algorithms such as the ones discussed in [18, 21, 27, 31, 35 and 50] have gained recognition and are implemented in many well-known applications, including the graph drawing tool Graphviz[1] and the computational software Mathematica[2]. These tools typically provide a special language or syntax for describing graphs. They parse these descriptions and apply graph drawing algorithms in order to find good graph layouts. The resulting drawings are then either rendered to the screen or converted into output formats such as PDF, EPS or PNG, allowing them to be imported into other applications and used in documents and other graphics.

This thesis focuses on graph drawing with Ti*k*Z, a versatile language for creating a wide range of vector graphics. Like other tools, it provides a compact syntax for specifying graphs and techniques for turning these graph descriptions into proper drawings. The work presented here was preceded by extensive research of existing graph drawing literature. Algorithms for two families of drawing methods—force-based layouts of general graphs and layered drawings of

[1] Graphviz website: http://graphviz.org

[2] Mathematica website: http://www.wolfram.com/mathematica/

directed graphs—were investigated and implemented specifically for use in Ti*k*Z. Even though the examples used in this thesis mostly stem from various areas of computer science, the developed algorithms can be applied to many fields of science, education, data visualization, and process modeling. The central topic of the subsequent chapters is to give an introduction to the graph drawing features of Ti*k*Z and to discuss the implemented algorithms, including their implementation, features, configuration options, and performance. The results produced by these algorithms are evaluated with regards to their visual properties and common aesthetic criteria.

What sets Ti*k*Z apart from other graph drawing tools are three central features. First of all, its scope is not limited to graph drawing. On the contrary, it has many powerful features for creating complex drawings. Drawing graphs and combining these drawings with other graphic elements is only one of them. In contrast to solutions such as Graphviz or the Open Graph Drawing Framework[3], Ti*k*Z is neither a standalone software nor a framework on top of which other graph drawing tools are built. It can be used to produce graphics for importing into other applications. Its main purpose, however, is to allow the creation of drawings that are embedded into the source code of documents built with the TEX typesetting system. This approach has a number of advantages. Not only does it simplify the document production workflow by enabling users to store, edit, and build drawings together with the rest of their documents. It also allows drawings to inherit font properties and typesetting features from their containing documents and from TEX, which helps in producing content with a consistent and professional look. The third central feature of Ti*k*Z is that—thanks to its modular architecture—it can be extended with new language features, graphics elements, and graph drawing algorithms easily. Unlike with other tools, where such extensions are either not possible at all or require fiddling with compiled languages and build tools, adding new graph drawing algorithms to Ti*k*Z is as simple as dropping a script into the TEX directory tree that is written in the Lua programming language and implements a simple function. One thing to note here is that the performance of TEX and algorithms written in Lua is not comparable to the often optimized compiled code used in other tools. This can be a problem with many graph drawings or drawings of large graphs. Luckily, some variants of TEX provide workable solutions to address these issues.

Contributions

Support for algorithmic graph drawing in Ti*k*Z is provided by a dedicated subsystem called the *graph drawing engine* that takes a graph description, passes it over to a Lua framework where it is turned into an object-oriented representation of the graph and processed with the desired

[3] Open Graph Drawing Framework website: http://ogdf.net

graph drawing algorithm. The resulting layout is returned to Ti*k*Z for rendering and embedding into the output document.

The main contributions of this thesis are the implementation of algorithms for two popular drawing methods applicable to many graphs. These algorithms were implemented specifically for the Lua-based graph drawing engine of Ti*k*Z and are presented along with Ti*k*Z syntax additions for users to select and configure them. They are evaluated with regards to layout quality and performance, based on example graphs collected from literature and lecture notes.

The force-based algorithms developed for Ti*k*Z are based on the spring model by Kamada and Kawai [35] and the spring-electrical model by Eades [15], Fruchterman and Reingold [21]. Their underlying idea is to model the input graph as a system of spring forces or spring and electrical forces, depending on the model, and generate a layout for the graph by simulating the node movements caused by these forces until the system energy reaches an equilibrium state. Three such algorithms were implemented, including a spring and a spring-electrical algorithm based on the work by Hu [31] and a modified spring-electrical algorithm based on ideas by Walshaw [51]. These algorithms feature techniques for improving the generated layouts and reducing the computational cost such as a multilevel approach for escaping local energy minima and an algorithm for approximating electric forces based on the Barnes–Hut opening criterion [2]. Scaling issues found in the original algorithms are addressed and partially solved. The produced layouts are evenly distributed and feature an almost uniform distribution of edge lengths as well as a high degree of symmetry.

The second family of drawing methods investigated are algorithms for layered drawings of directed graphs based on the Sugiyama method introduced in [18 and 50]. This method intends to highlight the flow of directed graphs by arranging its nodes in layers from the top to the bottom. Generating layered drawings is commonly implemented in five steps: cycle removal, layer assignment, crossing minimization, coordinate assignment, and edge routing. Since all of these steps are hard to solve, a variety of interchangeable heuristics and accurate algorithms are presented throughout graph drawing literature. For this thesis, a modular framework for layered drawing algorithms was developed and added to Ti*k*Z. It is based on the Sugiyama method and provides an extension interface that allows new solutions for each of its steps to be written and selected in Ti*k*Z easily. A number of popular sub-algorithms were implemented for the different steps including the versatile techniques proposed by Gansner et al. [27]. In its default configuration, the modular algorithm produces layered drawings that are comparable to those generated by Grapvhiz and Mathematica. Its main advantage, however, is the simple extension interface which makes experimentation with new ideas easy and which I am confident other graph drawing researchers will find useful.

In addition to this, a considerable amount of work was put into completing and improving the graph drawing engine. Contributions to the Lua algorithm framework include a powerful class for performing vector operations; a class for lowering the efforts of writing algorithms based on iterative depth-first-search; helper classes like a Fibonacci heap, a priority queue, and a quadtree; standard algorithms like Dijkstra and Floyd-Warshall; approximative algorithms for detecting cycles and computing graph diameters as well flexible iterators for arrays and hash tables. Additions to the Ti*k*Z syntax range from a new macro for creating mesh graphs to options for fixing nodes at certain coordinates and for specifying node clusters. All of these features were merged into the Ti*k*Z repository and are available to graph drawing researchers intending to implement new graph drawing algorithms with Ti*k*Z.

Related Work

Graph drawing has been the topic of many books, research papers, and software projects. Early forms of geometric drawings of graph-like structures can be found in 13th century games, family trees in genealogy, and the square of opposition used in Aristotelian logic. Forms of visualization more akin to the abstract graph drawings of today came up along with early work on graph theory by Euler and Vandermonde in the 18th and 19th century. An interesting survey on the history of graph visualization was written by Kruja et al. [36].

The idea to model graphs as systems of spring and electric forces originates from work by Kamada and Kawai [35], Eades [15], Fruchterman and Reingold [21]. It was extended by a multilevel approach from graph partitioning [30] in algorithms developed by Walshaw [51 and 52] and Hu [31] in order to overcome problems with local energy minima and thereby yield graph layouts of better quality. In an attempt to increase the efficiency of their spring-electrical algorithms, Walshaw and Hu successfully applied an algorithm for approximating the forces in an N-body system. This technique was initially proposed by Barnes and Hut [2] and was later improved by Dubinski [14]. Alternative force-based approaches not based on springs include the binary stress model [34], multidimensional embeddings [53], and simulations of magnetic fields [12].

The concept of highlighting the flow of directed graphs by arranging nodes in layers was first brought up by Sugiyama et al. [50]. Their ideas, which are often referred to as the *Sugiyama method*, have inspired an entire family of similar algorithms. Detailed surveys on these algorithms are provided by Eades and Sugiyama [18], Bastert and Matuszewski [3] and Di Battista et al. [12]. One of the most frequently quoted and implemented layered drawing algorithm with support for optimal layer assignment was developed by Gansner et al. [27] based on the network simplex method described in [10]. Articles and book chapters on individual sub-problems of the

Sugiyama method such as the feedback arc set problem, optimal crossing minimization, and optimal layer assignment can be found in [4, 8, 9, 17, 20 and 33],

Most of the algorithms presented in graph drawing literature, including those implemented in this thesis, assume that nodes do not have a size that needs to be taken into account. This assumption, however, does not hold in many practical scenarios and ignoring the size of nodes can lead to undesired overlaps between pairs of nodes as well as nodes and edges. Possible solutions for this differ depending on the drawing method. In [29], Harel and Koren describe the challenges of supporting arbitrarily sized nodes in force-based algorithms and propose carefully constructed extensions to support such nodes. An alternative solution based on a proximity stress model is proposed by Gansner and Hu in [24]. An entire set of techniques for eliminating overlaps in a post-processing step of general-purpose drawing methods is presented in [41]. Friedrich and Schreiber discuss the problem of supporting arbitrarily sized nodes in layered drawing algorithms based on the Sugiyama method. They propose a special layer assignment algorithm that takes the size of nodes into account and allows users to choose between compact and less compact drawings [23].

In practical scenarios nodes and edges often have labels attached to them that need to be taken into consideration when producing graph layouts. The problem of optimal label placement is discussed in [39] and [37].

Many of the aesthetic criteria applied to graph drawings were initially established by developers of graph drawing algorithms without proof of their relevance with regards to the readability of graphs. Several empirical studies have been performed by Purchase et al. in an attempt to improve this situation and by providing evidence for common criteria such as minimization of edge crossings and display of symmetry [43, 44, 46 and 47].

As mentioned before, many of the algorithms developed over time have been incorporated into software products such as graph drawing tools and frameworks. Projects related to the work presented in this thesis include applications such as Graphviz or Mathematica, frameworks such as TULIP[4] or the Open Graph Drawing Framework, and graph editors such as Gravel[5] [1], yEd[6] or Ti*k*Zit[7]. Graphviz provides a compact graph syntax similar to that of Ti*k*Z and ships a set of command line applications to produce graph layouts using different algorithms and convert them into output formats such as PDF or PNG. It implements spring and spring-electrical algorithms based on [31] and a layered drawing algorithm based on work by Gansner et al. [27], who are

[4] TULIP website: http://tulip.labri.fr

[5] Gravel website: http://www.rbergmann.info/projekte/gravel.html

[6] yEd website: http://www.yworks.com/en/index.html

[7] Ti*k*Zit website: http://tikzit.sourceforge.net/

also the developers of Graphviz itself. Mathematica is a commercial software for performing various sorts of computations. It offers a graph syntax and graph drawing algorithms as one of its many features. Algorithms for spring and spring-electrical layouts, multidimensional embeddings, layered drawings, and drawings of trees are supported [53]. These are essentially the same as the ones implemented in Graphviz and in this thesis. The Open Graph Drawing Framework and TULIP are C++ frameworks that can be used to build applications for graph drawing and data visualization. Like Ti*k*Z they can be used to develop new algorithms. A related area of research that combines graph drawing algorithms with graph editing interfaces is interactive graph drawing. It has spawned graphical editors such as yEd and Gravel that allow users to adjust the layouts produced by graph drawing algorithms or create their own layouts from scratch. The Ti*k*Zit software was written specifically for creating and editing Ti*k*Z graphs. It can be used to build graphs rapidly but does not incorporate elements of automatic graph drawing.

Structure of the Thesis

The remaining chapters of the thesis is organized as follows.

Chapter 2 briefly introduces the TEX typesetting system and gives an overview of Ti*k*Z as well as the basic elements of its compact graph syntax. It further explains the central aspects of the graph drawing engine, including its architecture, the graph drawing process from a graph description up to the final drawing, and the Lua interface for writing graph drawing algorithms. The chapter closes by summing up the contributions made to Ti*k*Z in the course of the thesis.

Chapters 3 and 4 present the implemented graph drawing algorithms for force-based layouts and layered drawings. Both chapters start off with an introduction covering the basics of the respective drawing method, including the underlying concepts and objectives. Chapter 3 explains how spring and electrical forces can be applied to graph drawing and describes differences, similarities, common elements and various implementation details of the developed spring and spring-electrical algorithms. Chapter 4 presents the building blocks of the Sugiyama method and provides an extensive survey of various existing techniques for some of its steps. This is followed by a description of the modular framework for layered drawing algorithms developed as part of the thesis. Both chapters close with a detailed tutorial explaining the Ti*k*Z syntax added for users to select and configure the new algorithms. In addition to this, chapter 4 also explains the Lua interface for writing new algorithms for the individual steps of the Sugiyama method.

In chapter 5, the layouts produced by the developed algorithms for a representative collection of example graphs are compared and evaluated based on common aesthetic criteria. The time required to process these graphs is analyzed in two benchmarks. Possible causes for the slow performance of the implemented spring and spring-electrical algorithms are discussed, followed by a description of possible optimizations and workarounds.

The thesis closes with a summary of the achievements made in this thesis, highlights open problems and hints at possible areas of future research or development in chapter 6.

2

An Overview of Ti*k*Z and its Graph Drawing Features

Ti*k*Z (a German recursive acronym for "Ti*k*Z ist kein Zeichenprogram") is a powerful language for producing a wide range of vector graphics, including graphs, diagrams, charts and plots. It is applicable to a variety of scenarios and is frequently used in scientific publications due to its well-documented implementation for the TEX typesetting system. In TEX distributions, Ti*k*Z is shipped as a front-end language for the PGF (Portable Graphics Format) package that provides drawing primitives and takes care of converting them into one of the desired output formats such as PostScript or PDF. The PGF system is designed to be extensible, allowing new libraries for additional TEX commands, special drawing routines and markups to be added easily. It was originally created by Till Tantau and is developed as an open source project driven by volunteers[8].

One of the key applications of Ti*k*Z is drawing graphs. The traditional way to do this in Ti*k*Z was to define a set of nodes and edges and position the nodes manually. Unfortunately, the syntax for creating nodes and edges was relatively complicated and verbose compared to the graph languages of related tools. To solve this, Tantau developed a simplified graph syntax, much alike to the syntax of the popular Graphviz software. He also added a number of semi-automatic node placement strategies to overcome some of the problems with manual positioning. However, these strategies are useful only to some extend and were not enough to get rid of manual positioning entirely, which can quickly turn into a time-consuming task as graphs become larger. To solve this problem, a group of students added a framework for extending Ti*k*Z with automatic graph drawing algorithms written in the Lua programming language.

[8] The Ti*k*Z/PGF project on SourceForge.net: http://sf.net/projects/pgf

When I started working on the thesis, I took over from these students, who had just finished a first functional version of the Lua algorithm framework, called the *graph drawing engine*. I spent the first weeks completing and improving this framework and invested a considerable amount of time into making its interfaces consistent. In addition to this, a number of new classes and further modifications to framework became necessary while implementing the graph drawing algorithms presented in chapters 3 and 4.

In order to get an understanding of Ti*k*Z, its graph syntax, the design of the graph drawing engine and its Lua interface for graph drawing algorithms, this chapter provides a detailed introduction to all these aspects. In section 2.1, the basic functionality and organization of Ti*k*Z/PGF are explained, followed by an introduction to the simplified graph syntax in section 2.2. The graph drawing engine is presented in section 2.3. The final section of this chapter, section 2.4, sums up my own contributions to Ti*k*Z/PGF in the course of this thesis and briefly describes some of the modifications made. These modifications are partly a result of the initial cleanup and partly consist of side products created while working on the graph drawing algorithms presented in the subsequent chapters.

2.1 Ti*k*Z as a Front-End Language for the PGF Graphics Package

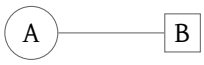
The TEX typesetting system was written by Donald Knuth and is frequently used for scientific publications, presentations, lecture notes and books. It features a powerful syntax, an interpreter and processing tools to convert TEX documents into a variety of output formats such as DVI, PostScript or PDF. Unlike documents created with WYSIWYG (What You See Is What You Get) word processing tools such as the one included in LibreOffice, TEX documents are written as source code and are structured and annotated using special elements called *macros*. Macros look bear similarities with functions known from programming languages. However, they differ from functions in that they produce only text. When a TEX document is processed, macros are expanded and replaced by the text they produce. Various tricks developed over time turn the TEX macro syntax into a powerful programming language that can be used to extend the basic functionality and core macros provided by TEX.

The PGF system is a macro package that implements a number of powerful programming concepts on top of the basic TEX macros and provides a powerful set of features to create complex graphics. It consists of three different layers. The *system layer* acts as an abstraction layer on top of output drivers—applications that are able to generate special output formats but that all have their own input syntax—and provides a single abstract interface for converting PGF graphics into different output formats. On top of this sits the *basic layer* that provides TEX macros to produce complex graphics and consists of a *core* system and a set of *libraries*, each of which

wraps special-purpose functionality e.g. for plotting data or for decorating objects. While the basic layer provides enough to create various kinds of complex drawings, it is tedious to use the macros all the time. Therefore, PGF has a *front-end layer* that provides its own set of high-level macros and a special graphics syntax that is more convenient to use. Providing a syntax similar to METAPOST, TikZ is one such front-end and, in fact, the only complete one at the time of writing.

TikZ provides a syntax for PGF drawing primitives and high-level graphics objects—such as graphs or decorated paths—that is significantly less verbose and cryptic than the corresponding basic layer macros. Take, for instance, the following example graphic that connects a circle with a rectangle, first with PGF basic layer macros and then with TikZ.

```
\startpgfpicture
  \pgfsetstrokecolor{black!60}
  \pgftransformshift{\pgfpoint{0cm}{0cm}}
  \pgfnode{circle}{center}{A}{a}{\pgfusepath{stroke}}
  \pgftransformshift{\pgfpoint{2cm}{0cm}}
  \pgfnode{rectangle}{center}{B}{b}{\pgfusepath{stroke}}
  \pgfpathmoveto{\pgfpointanchor{a}{east}}
  \pgfpathlineto{\pgfpointanchor{b}{west}}
  \pgfusepath{stroke}
\stoppgfpicture
```




```
\tikz[every node/.style={anchor=center,draw=black!60}] {
  \node [circle]    (a) at (0cm,0cm) { A };
  \node [rectangle] (b) at (2cm,0cm) { B };
  \draw (a) -- (b);
}
```

This example illustrates some of the basic features of TikZ, namely a special syntax for specifying nodes (`\node`), a special syntax for specifying paths (`--`), a key-value syntax for graphic parameters (`draw=black!60`), and grouping of parameters using styles (`every node/.style`). Notable other features of TikZ include a special syntax for edges, trees, actions on paths, coordinate transformation.

Its powerful set of mathematical and graphics macros, the simple TikZ syntax, the high rendering quality, and the vast amount of special-purpose libraries shipped with it—ranging from key-value management and loop constructs to object and path decorations as well as arbitrarily complex syntax additions—make PGF one of the most versatile graphics packages for the TEX world. Embedding the source code of TikZ graphics directly into TEX documents has the

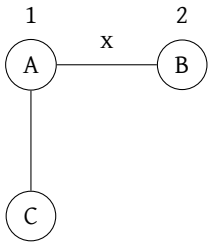
advantage of inheriting a syntax for mathematical formulas as well as font and typesetting features without the need to manually update and export graphics every time the document font changes, as is often necessary when using external graphics tools. This, along with the well-written manual[9] comprising more than 800 pages of extensive documentation—including many examples and tutorials—, may be another reason why TikZ and PGF have become popular tools in many areas of academia.

In the rest of this thesis, the term TikZ is used as a synonym for the entire graphics package.

2.2 A Special Syntax for Specifying Graphs

The basic graphic elements required for defining and drawing graphs are nodes and edges, with optional labels attached to them. The traditional way to create these elements in TikZ is via the syntax used in the initial TikZ/PGF example:

```
\tikz[every node/.style={draw,circle}] {
  \node (a) at (0, 0) [label=above:1] { A };
  \node (b) at (2, 0) [label=above:2] { B };
  \node (c) at (0,-2) { C };

  \draw (a) edge node[above,draw=none] { x } (b);
  \draw (a) edge (c);
}
```



While this syntax is easy to get used to and the naming of macros and options is fairly predictable and intuitive, creating complex graphs like this quickly becomes a tedious task. The syntax is verbose compared to tools like Graphviz, where creating nodes and edges is as simple as writing `A -> B`. The resulting source code is often complex compared to that required to draw graphs with tools like Graphviz, where creating nodes and edges is as simpl as writing `A -> B`. This may easily result in graph descriptions that are inflexible and hard to understand.

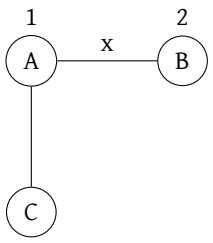
For this reason, the development version of TikZ and PGF features a simplified graph syntax that allows to specify graphs more quickly with significantly less syntactical overhead. This syntax, which already existed before this thesis was started, comes in the form of a special graphics instruction called `\graph`, is provided by a dedicated library `graphs` that can be loaded with the following instruction.

[9] Link to the TikZ/PGF manual: http://mirrors.ctan.org/graphics/pgf/base/doc/generic/pgf/pgfmanual.pdf

```
\usetikzlibrary{graphs} % LaTeX
\usetikzlibrary[graphs] % ConTeXt
```

The graph syntax is inspired by the Graphviz graph language and supports features such as a simplified syntax for nodes and edges, node chains to connect nodes and edges more easily, node groups to connect groups of nodes and apply the same options to multiple nodes at once, color classes to create logical sets of nodes, and special macros for creating standard graphs such as complete graphs or cycles. Using this syntax the above graph example can be written as follows:

```
\tikz \graph [nodes={draw,circle},
              grow right=2cm,
              branch down=2cm]
{
  A ["1"] --["x"] B ["2"],
  A -- C
};
```

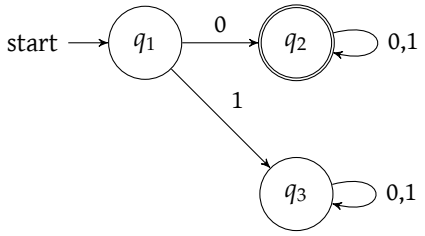


2.2.1 Syntax for Nodes and Edges

Nodes may have a name and are created using a syntax that matches one of the following patterns. Regular nodes are created with ⟨*node name*⟩. When using this notation, the node name is not only used as the text to display inside a node but is also the name via which the node can be referenced when creating edges. In situations where it is preferred to have a node with different reference name and text, the ⟨*node name*⟩__⟨*node text*⟩ syntax can be used. Here, the node name is used for referencing while the node text is used as the text displayed inside the node. Notes created with either of these formats may be assigned optional options that are appended in brackets: ⟨*node name*⟩__⟨*node text*⟩ [⟨*options*⟩]. Anonymous nodes that cannot be referenced internally may be created by omitting the ⟨*node name*⟩ part. Due to the way parsing works, certain special characters may not be used in the node name, including commas, semicolons, hyphens, braces, dots, parentheses. In the ⟨*node text*⟩ they are only supported if wrapped inside { and }.

Edges are created between nodes or node groups using one of the supported graph operators such as `--` for undirected edges, `->`, `<-` and `->` for directed edges and `-!-` for deleting edges previously created. Like nodes, edges may be configured using optional parameters, as is shown in the finite state machine example below.

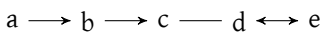
```
\tikz \graph [no placement] {
  q1__$q_1$ [at={(0,0)},state,initial],
  q2__$q_2$ [at={(2,0)},state,accepting],
  q3__$q_3$ [at={(2,-2)},state],

  q1 -> ["0"] q2,
  q1 -> ["1"] q3,
  q2 -> ["{0,1}",loop right] q2,
  q3 -> ["{0,1}",loop right] q3,
};
```



2.2.2 Node Chains and Chain Groups

Two useful features that reduce the code required to build complex graphs are node chains and groups. As the name suggests, node chains allow a sequence of pairwise connected nodes to be specified using a slightly simplified syntax:

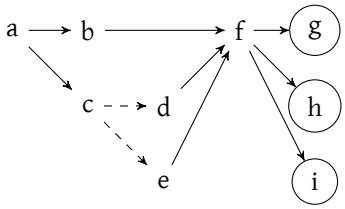
```
\tikz \graph { a -> b -> c -- d <-> e };
```



Each of the nodes and edges in a chain may be specified using the syntax explained in section 2.2.1.

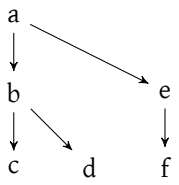
Chain groups make creating graphs easier by allowing two things: create multiple edges with a single statement and apply options to several individual nodes at once. Groups of nodes may be placed anywhere in a graph definition where individual nodes are allowed. The example below demonstrates how node groups may be used to connect multiple nodes at once and assign options to nodes in two different groups:

```
\tikz \graph {
  a -> {
    [edges={dashed}] b, c -> { d, e }
  } -> f -> {
    [nodes={draw,circle}] g, h, i
  },
};
```



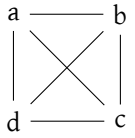
This example already shows that chain groups may be nested, making compact definitions of hierarchical graphs possible, which is a feature that is particularly useful when creating trees:

```
\tikz \graph [grow down,branch right] {
  a -> {
    b -> { c, d },
    e -> f,
  }
};
```



Chain groups also support so-called graph operators, that is, algorithms that receive the nodes of a group as input and create edges between them automatically or semi-automatically. One such operator provided by the `graphs.standard` library is the `clique` operator, which connects all nodes in a group:

```
\tikz \graph [clockwise=4,phase=135] {
  { [clique] a, b, c, d }
};
```



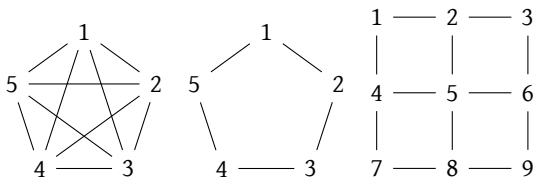
Node chains, chain groups and operators are powerful tools for creating graphs. The above examples only highlight a small set of their features. The list of operators provided by Ti*k*Z out of the box is long and can be extend by custom ones easily.

2.2.3 Graph Macros

Often, graphs are composed of nodes connected by standard patterns such as cliques, circles or meshes. This is the basic idea behind the graph macros provided by Ti*k*Z. These macros allow standard subgraphs to be created without having to create nodes and connect nodes individually. Some of these standard graphs will be used frequently in the remaining chapters of this thesis and it is thus helpful to know their syntax.

A graph macro consists of a name and has options that define how many nodes are created and how they are connected. The following example illustrates how a complete subgraph (`K_5`), a circle with five nodes (`C_5`) and a small mesh are created using graph macros:

```
\tikz \graph [clockwise=5]
  { subgraph K_n[n=5] };
\tikz \graph [clockwise=5]
  { subgraph C_n[n=5] };
\tikz \graph [grid placement]
  { subgraph Grid_n[n=9] };
```



2.3 The Graph Drawing Engine and its Lua Interface

The graph syntax is a notable improvement compared to the previous verbose notation. However, it does not eliminate the other main challenge of drawing graphs: arranging nodes so that a graph is visually appealing and easy to understand. Along with the new graph syntax, Ti*k*Z introduces a number of semi-automatic placement strategies like the ones used in examples section 2.2. Supported placements include `Cartesian placement`, `circular placement`, `grid placement`, and `manual placement`. If no other options are passed to the `\graph` macro, Ti*k*Z defaults to `Cartesian placement`, which starts with the first node and arranges the others by moving either along the `grow` axis or `branch` axis, depending on whether a node belongs to the same node chain or a different chain group. The `circular placement` strategy allows nodes to be placed in a circle based on an angle, a radius and a direction, which may be either `clockwise` or `counterclockwise`. Placing nodes in a grid with `grid placement` was contributed as part of the thesis and is explained in section 2.4.

Starting from a formal definition of a graph, users often have no visual representation in mind before they create a drawing. Automatic or *algorithmic* graph placements may help in finding such a representation and may even generate drawings pleasing enough to require no further adjustments. Even though the semi-automatic placement strategies provided by Ti*k*Z are reasonably powerful and sufficient in many situations, they are limited as soon as graphs get larger. Large graphs and the various existing drawing paradigms—ranging from trees and layered drawings to orthogonal drawings and force-based layouts—demand for more advanced methods. For this purpose, a new graph drawing subsystem called the *graph drawing engine* was recently added to Ti*k*Z/PGF.

The graph drawing engine allows arbitrarily complex algorithms to be implemented using a modern and lightweight scripting language—Lua. Since TEX has a reputation of being slow and difficult to program with, the development of complex drawing algorithms purely using TEX macros would not have been realistic. Lua on the other hand is not only a small modern language known to be fast, powerful and easy to learn; it also comes embedded in the prominent LuaTEX system built on top of TEX. With the `\directlua` macro provided by LuaTEX, Lua code can be executed in TEX documents easily, allowing arbitrary data to be exchanged between TEX and Lua scripts. This is what finally made the framework for graph drawing algorithms—a feature long planned by Tantau—possible.

2.3.1 Basic Concepts and Architecture of the Graph Drawing Engine

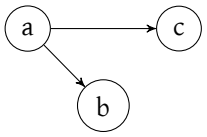
The central idea of the graph drawing subsystem provided by the `graphdrawing` library is to collect the nodes and edges declared in a graph drawing scope, e.g. a `\graph` macro, and hand them

over to a Lua framework where an object-oriented representation of the graph is constructed and processed with automatic graph drawing algorithms. These algorithms assign positions to the individual nodes and may inject additional Ti*k*Z options into the nodes and edges defined by users in order to shape edges and adjust the visual appearance of the final drawing in general. Once this process is completed, the modified nodes and edges are returned to the TEX layer where PGF takes care of rendering the graph. When graphs, nodes and edges are parsed, the graph drawing engine splits their options up into graph drawing options and regular Ti*k*Z options. The graph drawing options are parsed, evaluated and are either passed to Lua along with requests to create graph, nodes, and edge objects or are used to determine which graph drawing algorithm to run. Dedicated libraries such as `graphdrawing.force` are used to implement special graph drawing algorithms and provide additional Ti*k*Z options to configure them.

The Graph Drawing Process

To illustrate the basic control flow of the graph drawing engine, let us take a simple example:

```
\tikz \graph [nodes={draw,circle},
              layered drawing,
              orient=a-c]
{
  a -> b, a -> [layered drawing={minimum levels=2}] c,
};
```



The elements in this drawing are:

- a `\graph` object that defines the graph drawing scope,
- a regular Ti*k*Z option (`nodes={...}`) that remains untouched by the graph drawing engine,
- a `layered drawing` option that specifies the drawing algorithm to be used,
- an `orient` graph drawing option, which is passed down to Lua for the graph drawing engine to rotate the final drawing,
- three nodes (`a`, `b`, and `c`), and
- two directed edges (`a -> b` and `a -> c`), of which the second has an algorithm-specific option (`minimum levels`) that is used to separate `a` and `c` by a minimum number of levels or layers.

Whenever Ti*k*Z encounters an algorithm such as `layered drawing` that selects a graph drawing algorithm, it calls the `\pgfgdbeginscope` macro, which in turn enables the recording of node and edge declarations with the help of edge and node callback macros. This macro also requests

that the Lua framework allocates a new graph object by calling `Interface:newGraph` and passing graph parameters like `layered drawing` and `orient` to this method call. The `\graph` macro is not the only create a graph or trigger automatic graph drawing algorithms. More information about the alternatives is included in the TikZ manual.

When a node declaration is detected, TikZ calls `\pgf@gd@positionnode@callback`, which divides options into regular TikZ options and options specific to graph drawing and subsequently calls `Interface:addNode`, passing to it the two sets of options as well as the node's internal name and dimensions. For all edges declared in a graph drawing scope, `\pgfgdedge` is called after all nodes have been created. This macro separates graph drawing options from regular TikZ options and calls the method `Interface:addEdge` to create a Lua edge object based on the names of the edge's end nodes, the edge direction as well as its options.

When the graph drawing scope is closed, TikZ calls the `\pgfgdendscope` macro, which instructs the Lua framework to run the desired graph drawing algorithm (`Interface:drawGraph`) and return the modified graph elements back to TikZ for rendering (`Interface:finishGraph`).

Organization of the Graph Drawing Subsystem

The graph drawing engine, including the `graphdrawing` library, the Lua framework, and special-purpose libraries such as `graphdrawing.force`, resides in `/generic/pgf/graphdrawing` inside the PGF directory tree. Its directory structure is split up into core functionality and algorithms. The `core` consists of TikZ macros and syntax additions, located in `core/tikzlayer`, a set of PGF basic layer macros in `core/basiclayer` as well as the Lua interface and framework with its classes, residing in `core/lualayer`. The `algorithms` tree contains a directory for each family of algorithms, e.g. `algorithms/force` for force-based algorithms and `algorithms/trees` for tree algorithms. Each of these directories includes a file representing the TikZ library for the family, called `pgflibrarygraphdrawing.`⟨*family*⟩`.code.tex` by convention. This file implements the TikZ options necessary to select and configure the graph drawing algorithms associated with the family. The algorithms themselves are stored in scripts that are required to follow the `pgfgd-algorithm-`⟨*algorithm*⟩`.lua` naming scheme.

New algorithms or algorithm families do not need to be added to `/generic/pgf/graphdrawing`. Instead, they can be developed and stored anywhere in the `TEXMF` tree. The only requirement is for the corresponding filenames to match the filename naming schemes explained above.

2.3.2 Interface for Graph Drawing Algorithms

Extending the graph drawing engine by new algorithms is a process that involves two steps. First, the algorithm needs to be made available to Ti*k*Z users by adding the necessary Ti*k*Z options to the `pgflibrarygraphdrawing.`⟨*family*⟩`.code.tex` file. This includes a definition of the algorithm itself as well as graph, node and edge options to configure the algorithm and the drawings it produces. The second step is to add a script that actually implements the algorithm. For an algorithm family ⟨*family*⟩ and an algorithm ⟨*algorithm*⟩, the name of this script would be `algorithms/`⟨*family*⟩`/pgfgd-algorithm-`⟨*algorithm*⟩`.lua`.

Defining Options for Selecting and Configuring Algorithms

In Ti*k*Z, options are stored in a key-value tree that is in some ways similar to the directory tree on UNIX systems. This allows to separate options into different branches or `namespaces` and evaluate only parts of the entire key-value tree on demand. General Ti*k*Z options like `draw`, `red` or `decoration` are stored inside `/tikz`, while options introduced by the `graphs` library are located inside `/tikz/graphs`. All options related to graph drawing are stored in `/graph drawing`.

The main problem with Ti*k*Z options is that they are only available in Ti*k*Z/PGF and need to be transfered to Lua in a way that allows algorithms to access them easily. The graph drawing engine solves this by making `/graph drawing` options available in Ti*k*Z graphics using a technique called *key-forwarding* and by filtering all options related to graph drawing using special key handlers—macros that are executed whenever an option is set. Graph drawing specific options set for graphs, nodes and edges are forwarded to the Lua framework where they are stored in associative arrays in the corresponding graph, node and edge objects.

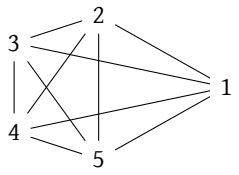
In order to give a basic understanding of how the declaration of algorithms and options works, the below example demonstrates this for a simple algorithm that places nodes on a circle and optionally allows nodes to be moved to a virtual circle with a different radius.

```
\pgfgdset{
  circular layout/radius/.graph parameter=evaluate math expression,
  circular layout/radius/.parameter initial=2cm,
  circular layout/node radius/.node parameter=evaluate math expression,
}
```

```
\pgfgddeclarealgorithmkey
  {flexible circle} % the TikZ algorithm name
  {circular layout} % the family whose options to evaluate
  {
    algorithm=flexible circle, % the actual name of the algorithm
  }
```

Once the options for an algorithm like `flexible circle` are set up, the algorithm can be selected and configured in Ti*k*Z graph drawings:

```
\tikz \graph [flexible circle] {
  1 [flexible circle={node radius=2cm}],
  subgraph K_n[n=5]
};
```



Implementing Algorithms in Lua

The second step of adding a new algorithm for automatic graph drawing in Ti*k*Z is to actually implement it using the Lua scripting language. Ti*k*Z makes this easy by automatically detecting new algorithms after they have been dropped into the `TEXMF` directory tree in the form of a `pgfgd-algorithm-`⟨*algorithm*⟩`.lua` script. Here, ⟨*algorithm*⟩ is the same string that would be passed to the `/graph drawing/algorithm` option, except that spaces are replaced by dashes. Thus, when an algorithm such as `algorithm=flexible circle` is used in Ti*k*Z, the graph drawing engine will attempt to load `pgfgd-algorithm-flexible-circle.lua` and run the algorithm implemented in this script.

In its Lua script, every algorithm is required to implement a special function that is used as an entry point and is called whenever the algorithm needs to be executed. The name of this function is once again the name passed to `/graph drawing/algorithm`, although this time spaces and dashes are replaced by underscores. The resulting basic structure of every graph drawing algorithm in Ti*k*Z is shown below.

```
pgf.module('pgf.graphdrawing')

function graph_drawing_algorithm_⟨algorithm⟩(graph)
  ...
end
```

The first line imports the `pgf.graphdrawing` module and makes all core classes and functions available in the script. The `graph_drawing_algorithm_`⟨*algorithm*⟩ function is where the algorithm logic is implemented. This function is called whenever a graph needs to be arranged with the algorithm. It receives an instance of the Lua `Graph` class that represents the TikZ graph created by the user. Its task is to assign every node in the graph a position consisting of x- and y-coordinates.

The `pgf.graphdrawing` module provides a number of useful classes for working with graphs, ranging from the base `Graph` class to powerful iterators, a callback-based depth-first-search implementation, several standard graph algorithms and data structures such as a priority queue. Of course, the central classes are `Graph`, `Node` and `Edge`, which can be used to traverse the nodes and edges of the input graph, read and generate TikZ options and create auxiliary copies of the graph. The following example demonstrates how the `flexible circle` algorithm can be implemented using a simple node traversal. It also shows how to use the `node.pos:set` method to change the coordinates of individual nodes.

```
pgf.module("pgf.graphdrawing")

function graph_drawing_algorithm_flexible_circle(graph)
  local radius = tonumber(
    graph:getOption('/graph drawing/circular layout/radius')
  )

  local alpha = 2 * math.pi / #graph.nodes
  local i = 0

  for node in table.value_iter(graph.nodes) do
    local node_radius = tonumber(
      node:getOption('/graph drawing/circular layout/node radius')
      or radius
    )

    node.pos:set{
      x = node_radius * math.cos(i * alpha),
      y = node_radius * math.sin(i * alpha)
    };

    i = i + 1
  end
end
```

All algorithms start from this basic structure with the goal to assign coordinates to all nodes in a way that satisfies a number of drawing conventions, aesthetic criteria and visual constraints. As

we will see in the following chapters, algorithms may implement arbitrarily complex placement strategies based on the Lua classes provided by the graph drawing engine. In order to focus on the contributions made as part of the thesis, not all classes and features of the graph drawing framework are discussed further in this chapter. Note, however, that the TikZ/PGF manual includes tutorials on implementing new algorithms and also provides extensive documentation on how to use the fundamental classes provided by Lua.

2.4 Contributions to TikZ in the Course of this Thesis

This section highlights some of the features I added to the `graphs` library and graph drawing subsystem while working on the thesis.

2.4.1 Graph Drawing Algorithms

Despite contributing many changes to the graph drawing subsystem, my main efforts went into researching and implementing the graph drawing algorithms presented in the following two chapters. Overall, the following algorithms were added: a simple algorithm for drawing arbitrary trees (`whetherell tree`), a spring algorithm (`Hu2006 spring`), two spring-electrical algorithms (`Hu2006 spring electrical` and `Walshaw2000 spring electrical`), an algorithm framework for layered drawings (`modular layered`) and several techniques for creating layered drawings.

2.4.2 Modifications and Additions to the Lua Framework

Improvements of the Core Framework

While working on the graph drawing engine and the graph drawing algorithms presented in this thesis, I made several modifications to existing Lua classes and added a few useful classes as well.

The `Graph`, `Node` and `Edge` classes were modified to store nodes and edges in the order they are declared in TikZ. An implementation of a pseudo-diameter algorithm was added to the graph class in the form of the `Graph:getPseudoDiameter` function. Edge options were implemented in the TikZ and Lua layers as well as optional bend points, allowing graph drawing algorithms to define the shape of edges. Last but not least, the `Edge` class was extended to allow edges

to be reversed. All methods affected by this were modified to honor reversed edges, including the node methods `Node:getInDegree`, `Node:getOutDegree`, `Node:getIncomingEdges` and `Node:getOutgoingEdges` and edge methods like `Edge:getHead`, `Edge:getTail`, `Edge:isHead` and `Edge:isTail`.

A class that I added to manipulate the positions of nodes is the `Vector` class, which provides a powerful interface for managing vectors of arbitrary length, including support for vector operations such as adding (`Vector:plus`, `Vector:plusScalar`), subtracting (`Vector:minus`, `Vector:minusScalar`), dividing (`Vector:dividedBy`, `Vector:dividedByScalar`), multiplying (`Vector:timesScalar`, `Vector:dotProduct`) and working with vector norms (`Vector:norm`, `Vector:normalized`). Like many other classes and functions in the Lua graph drawing framework, the `Vector` class is inspired by functional programming. In addition to typical vector operations, it provides methods `Vector:update` and `Vector:limit` that can be used to update or limit the elements of a vector using callback functions that receive an existing index and element and return a new value for this pair. The `Vector` class also supports setting a reference or *origin vector* with the `Vector:setOrigin` method.

Another notable class added to the graph drawing framework is `DepthFirstSearch`, which provides a general-purpose interface to writing iterative depth-first search algorithms. Depth-first search is very simple to implement using recursive function calls. However, if implemented iteratively—which has the benefit of avoiding stack overflows—depth-first search requires manual stack management. Combined with manual tracking of the traversal status, this often makes algorithms require more code, and thereby, harder to understand. The `DepthFirstSearch` class provides a simple interface for pre-order and post-order algorithms based on depth-first search. All that algorithms need to specify are an initialization function, a visit function and a complete function. These functions are called at the beginning of the depth-first search, whenever a node is visited for the first time and when all its children in a directed graph or tree have been visited, respectively. The code snippet below shows how such an algorithm would be implemented.

```
local function initialize(dfs)
  -- push initial items to the DFS queue using dfs.stack:push(item)
end

local function visit(dfs, node)
  -- a node is visited for the first time; push children to the DFS stack
end

local function complete(dfs, node)
  -- post-order call; all children of the node have been completed
end

DepthFirstSearch:new(initialize, visit, complete):run()
```

While working on the graph drawing algorithms, a number of other, mostly special-purpose classes were added to the Lua framework. These include a `PriorityQueue` implementation based on new new `FibonacciHeap` class as well as the classes `CoarseGraph`, `QuadTree`, `Ranking` and `NetworkSimplex`, whose purpose will become clear when reading the following chapters.

New Iterators and Helper Functions

As mentioned before, the Lua graph drawing framework is strongly influenced by functional programming. The result of this can be seen when looking at the various iterators and helper functions that were added to it as part of my work.

In Lua, an iterator is a function that has knowledge about an underlying data structure. If called multiple times, e.g. in a loop, it yields the elements of this data structure one by one. Lua only has one complex data structure called `table` that is used for flat arrays, associated arrays and objects alike. As a result, all iterators in Lua serve the purpose of looping over the indices, keys, values or pairs of such a table.

The following functions were added to the Lua graph drawing framework to generate iterator functions for tables: `table.value_iter`, `table.key_iter`, `table.reverse_value_iter`, `table.randomized_value_iter`, `table.randomized_pair_iter`. These iterators created can be modified on the fly using filter and mapping functions that are applied using the `iter.filter` and `iter.map` functions.

Modifying tables in graph drawing algorithms has become easy thanks to the addition of functions such as `table.filter_keys`, `table.filter_values` and `table.filter_pairs` for filtering table values; `table.combine_values` and `table.combine_pairs` for mapping the values or key-value pairs of a table to a single value; `table.map`, `table.map_keys`, `table.map_values`, `table.map_pairs` to copy a table and map its data to something else;`table.remove_values`, `table.remove_pairs`, `table.update_values`, `table.reverse_values` to change the content of a table itself; or `table.merge_custom` and `table.copy_custom` to merge and copy tables along with the object metadata associated with it.

Other helpers added to the framework include algorithms such as `algorithm.dijkstra`, based on the `PriorityQueue` class, and `algorithms.classify_edges`, which performs a depth-first search in a directed graph and classifies edges into forward, cross and back edges. There is also a traversal module featuring a `traversal.topological_sorting` iterator function that allows to traverse the nodes of a directed graph in topological order. Many algorithms need to remove loops and merge multiple edges before they can process the graph. The manipulation module provides functions such as `manipulation.remove_loops` and `manipulation.restore_loops` to do this and revert it at a later stage.

2.4.3 Newly Added Ti*k*Z Options

Options for Changing the Orientation of Algorithmically Generated Drawings

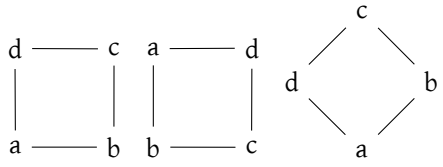
An alternative drawing orientation to that produced by graph drawing algorithms—such as the ones explained in the following chapters—is often desired. Sometimes, the majority of edges in a graph are supposed to point in a specific direction, e.g. from left to right. In other situations, a node is expected to appear to the left of or above another node. For this purpose, the `/tikz/graphs/orient=`⟨*orientation*⟩ and `/tikz/graphs/orient'=`⟨*orientation*⟩ options were created. They are implemented as a post-processing step performed at the end of `Interface:drawGraph` and are thus automatically applied whenever Lua graph drawing algorithms are used.

The ⟨*orientation*⟩ defines an axis consisting of two nodes and a rotation angle. It may can be specified using one of the following three notations:

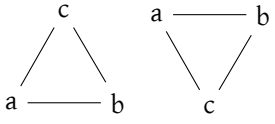
1. The ⟨*first axis node*⟩`-`⟨*second axis node*⟩ syntax will rotate the drawing so that the node with the internal name ⟨*first axis node*⟩ is positioned left of ⟨*second axis node*⟩ parallel to the x-axis.
2. ⟨*first axis node*⟩`|`⟨*second axis node*⟩ will adjust the orientation of the drawing so that ⟨*first axis node*⟩ appears on top of ⟨*second axis node*⟩ in parallel to the y-axis.
3. The third notation, ⟨*first axis node*⟩`:`⟨*angle*⟩`:`⟨*second axis node*⟩ allows a custom angle to be used for the axis created via the two nodes.

The `/tikz/graps/orient'` option provides the same functionality and syntax but causes the drawing to be flipped over the axis. The use of these two options is illustrated below.

```
\tikz \graph [spring layout,orient=a-b]
  { a -- b -- c -- d -- a };
\tikz \graph [spring layout,orient=a|b]
  { a -- b -- c -- d -- a };
\tikz \graph [spring layout,orient=a:45:b]
  { a -- b -- c -- d -- a };
```



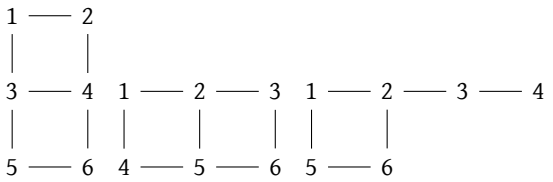

```
\tikz \graph [spring layout,orient=a-b]
  { a -- b -- c -- a };
\tikz \graph [spring layout,orient'=a-b]
  { a -- b -- c -- a };
```



2.4.4 Graph Macros and Placements

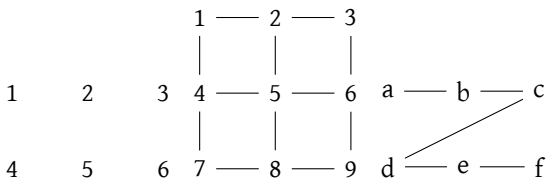
The `Grid_n` standard subgraph was added to the `graphs.standard` library. It takes two parameters, `/tikz/graphs/n=`⟨*N*⟩ and optionally `/tikz/graphs/wrap after=`⟨*C*⟩, and creates a mesh of ⟨*N*⟩ nodes, which is something I needed for testing the spring and spring-electrical algorithms presented in chapter 3. These nodes are connected so that the resulting mesh has ⟨*C*⟩ columns. If `wrap after` is not set, the macro attempts to create a $\lfloor\sqrt{\langle N\rangle}\rfloor \times \lfloor\sqrt{\langle N\rangle}\rfloor$ mesh. See below for an example demonstrating the use of the `Grid_n` subgraph.

```
\tikz \graph [grid placement] {
  subgraph Grid_n [n=6,wrap after=2]
};
\tikz \graph [grid placement] {
  subgraph Grid_n [n=6,wrap after=3]
};
\tikz \graph [grid placement] {
  subgraph Grid_n [n=6,wrap after=4]
};
```



In addition to this, a `grid placement` strategy was implemented in the `graphs` library using TEX macros. It places nodes in a $N \times M$ grid and is useful even though it is not as powerful as some of the graph drawing algorithms. The number of columns M is either set to `wrap after`, if this option is set, or is computed automatically as $\lfloor\sqrt{|V(G)|}\rfloor$, where $V(G)$ is the set of nodes in the graph. N is the number of rows needed to lay out the graph in a grid with M columns. The `grid placement` strategy can be used to draw mesh graphs and non-mesh graphs alike, as the following example demonstrates. Note that for this placement to work, the option `/tikz/graphs/n` always needs to be set.

```
\tikz \graph [grid placement] {
  subgraph I_n[n=6, wrap after=3]
};
\tikz \graph [grid placement] {
  subgraph Grid_n [n=9]
};
\tikz \graph [grid placement] {
  [n=6, wrap after=3]
  a -- b -- c -- d -- e -- f
};
```

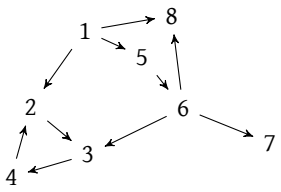


3

Force-Based Graph Drawing Algorithms for Arbitrary Graphs

This chapter covers force-based drawings of directed and undirected graphs and presents the algorithms implemented as part of this thesis. An example of a drawing produced by one of these algorithms is shown below, along with the corresponding Ti*k*Z syntax.

```
\tikz \graph [spring electrical layout] {
  1 -> 2 -> 3 -> 4,
  1 -> 5 -> 6,
  6 -> 7,
  6 -> 8,
  1 -> 8,
  6 -> 3,
  4 -> 2,
};
```



The discussion of algorithms for generating force-based layouts starts off with an introduction and background information section 3.1. This is followed by a presentation of the three algorithms implemented in the thesis, split across sections 3.2 and 3.3. Section 3.2 describes the general structure and building blocks of the spring and spring-electrical algorithms by Hu [31] and Walshaw [51 and 52]. Section 3.3 goes on to discuss the details and differences of these algorithms as well as their implementation in Ti*k*Z. The chapter closes with section 3.4 describing the various options added to Ti*k*Z for generating and fine-tuning spring and spring-electrical layouts.

3.1 Introduction to Force-Based Graph Drawing

Force-based or *force-directed* methods are a family of algorithms that simulate physical forces in order to produce visually appealing and balanced drawings of graphs. They are general-purpose methods in that they can be used to draw undirected and directed graphs alike. The two central aspects of these methods are the choice of the force model and the minimization of the overall system energy from which the final graph layout is derived. Researchers have experimented with simulating a variety of forces, ranging from spring forces to forces in an electric or magnetic field. A general introduction to force-based methods and an overview of various related techniques is given in [12].

The basic idea of a popular and often implemented force-based method is to arrange graphs by simulating the movements caused by attaching springs between pairs of nodes. The algorithm by Kamada and Kawai [35] connects all pairs of nodes with springs whose desired equilibrium length is set to be proportional to the graph distance of these nodes. Each spring is associated with a *spring force* that compresses or extends the spring depending on its actual length. Depending on whether two nodes are positioned too far apart or too close to each other, this force will either pull them closer to together or push them apart. A modification of this idea was developed by Eades [15], Fruchterman and Reingold [21]. In their algorithms, springs are only attached to pairs of nodes that are connected by an edge. In addition to this, all nodes are modeled as electrically charged particles or metal rings. While the spring force attempts to hold connected nodes at a certain distance, the electric charge induces an *electric force* between all pairs of nodes that has an attractive or repulsive effect, depending on its polarity. The drawings resulting from these two approaches are similar but differ in some aspects, as we will see later.

In graph drawing literature, spring methods based on the above ideas are referred to as *spring algorithms* and *spring-electrical algorithms*, the latter incorporating the electric forces by Eades, Fruchterman and Reingold. The generated drawings are sometimes called *spring layouts* and *spring-electrical layouts*. The term *spring embedder* is also popular and may refer to both concepts. Throughout this thesis the terms spring algorithm/layout and spring-electrical algorithm/layout are used consistently. The spring and electric forces are commonly—and somewhat misleadingly—called *attractive forces* and *repulsive forces*. The terms spring force and electric force are more accurate and thus replace common terminology in this chapter.

There are many reasons why spring and spring-electrical methods are popular in graph drawing literature and applications. One of their main advantages is that they are easy to understand due to their physical analogies and simple structure. Furthermore, they are known for producing good results with evenly distributed nodes, uniform edge lengths, and high degree of symmetry for small and medium-sized graphs. They also have the tendency to draw identical substructures

in a visually similar way, which is important when drawing grid-like graphs. Last but not least, several types of constraints can be implemented easily, including fixed node positions and node clusters [12].

The main disadvantages of spring and spring-electrical methods are poor running times and convergence towards local but not global energy minima. Minimizing the total system energy is an iterative process whose performance depends on the running time of the force computation at each step and the speed at which the solution converges towards the equilibrium state. Computing all electric forces in the spring-electrical system and moving nodes accordingly requires all pairs of nodes to be visited at least once. This is known as *N-body simulation* in physics. Solving it accurately requires $O(|V(G)|^2)$ operations. The overall convergence of the simulation depends on many factors and is commonly estimated to be in the order of $O(|V(G)|)$ iterations. Overall, this implies a running time of $O(|V(G)|^3)$, which is problematic as soon as graphs become larger. The other problem of spring and spring-electrical approaches is that systems of forces typically have many local energy minima, in particular so with large graphs. Starting from a random initial layout, a spring or spring-electrical algorithm is likely to settle for such a local but not global minimum and thereby produce a layout of lower quality.

Various techniques have been proposed to overcome these problems. Fruchterman and Reingold use a *grid-variant algorithm* that divides the plane into a grid of squares and only computes electric forces between nodes in nearby squares. This method is claimed to have a linear time complexity instead of a quadratic one if nodes are distributed uniformly on the grid [21]. A modified version of this approach was adopted in [51 and 52]. Fruchterman and Reingold also suspect that the *cooling scheme* used to reduce movements gradually over time—a mechanism to avoid infinite execution—is worth tuning in order to reduce the running time of spring and spring-electrical algorithms. Hu argues that the grid-variant algorithm ignores long-range electric forces and is unable to distribute far away nodes evenly. He instead proposes to approximate electric forces with the quadtree algorithm presented by Barnes–Hut [2 and 31], reducing the time complexity to $O(|V(G)| \log |V(G)|)$.

A popular strategy to avoid the problem of local minima is to apply a *multilevel* approach where the input graph is reduced to coarser and coarser graphs until its size falls below a certain threshold. The coarsest graph is then processed with a regular spring or spring-electrical algorithm that simulates the node movements step by step. Afterwards, the coarsening steps are reverted one by one while interpolating node positions from the previous layout and applying the spring or spring-electrical method once again at each of these steps. This approach is similar to moving whole parts of the graph at once and has been used successfully in graph partitioning before [30]. Fruchterman and Reingold suspect that the choice of force functions may have an

impact on the convergence towards a global minimum as well. They state that some of them may be better suited to overcome local minima than others [21].

3.2 Common Structure and Elements of Spring and Spring-Electrical Algorithms

In the course of the thesis, the algorithms presented in [21, 31, 35 and 51] were investigated. I chose to implement the spring and spring-electrical algorithms by Hu [31] and the spring-electrical algorithm by Walshaw [51] as they looked most promising and are documented well. My implementation of the Walshaw algorithm, however, differs from the original idea in that it employs the same multilevel approach and force approximation used in the Hu algorithms. This makes the algorithms almost identical in structure and emphasizes the effects that the few differences have on the layouts produced. These differences include the choice of force functions, the way to move nodes along a force, the cooling scheme, and the initial layout as well as scaling parameters. In this section, the structure and building blocks that all three algorithms have in common are presented.

3.2.1 Forces in Spring Algorithms

In the spring model by Kamada and Kawai [35], springs are attached to any pair of nodes. The scalar spring force is defined so that it satisfies Hooke's law, which requires the force to be proportional to the distance between the corresponding nodes.

Hooke's law expresses the restoring spring force as

$$F_s = -C_s(l - k) \tag{3.1}$$

where C_s is called the *spring constant* that encodes the material stiffness and $l-k$ is the difference between the actual length l and equilibrium length k of the spring. Applied to spring graph drawing algorithms, a natural choice for the spring force f_s is

$$f_s(u, v) = -C_s\left(\|x_u - x_v\| - k \cdot d(u, v)\right), \quad \{u, v\} \in S(G) \tag{3.2}$$

where x_u and x_v are positions of the nodes u and v on the plane and $d(u,v)$ denotes the graph distance between u and v. $S(G)$ refers to the set of springs attached to nodes in the graph G. In this model, the factor k describes the desired length of a single edge on the shortest path between u and v rather than the equilibrium length of the spring itself.

3.2.2 Forces in Spring-Electrical Algorithms

In the spring-electrical model by Eades [15], Fruchterman and Reingold [21], edges are modeled as springs that aim at keeping their end nodes at an equilibrium distance. In addition to this, nodes are regarded as electrically charged particles. The corresponding spring and electric forces are defined so that they satisfy Hooke's and Coulomb's laws, which require the spring force to be proportional to the distance between two end nodes and the electric attraction or repulsion of two nodes to be inversely proportional to the square of their displacement.

The Coulomb force of two charged particles is given by

$$F_e = C_e \frac{q_1 q_2}{r^2} \tag{3.3}$$

with C_e encoding properties of the space and q_1, q_2 representing the electric charge of two particles; r is the distance between these two particles. Applied to spring-electrical graph drawing algorithms, a natural choice for the spring force f_s and the electric force f_e is

$$f_s(u,v) = -C_s\left(||x_u - x_v|| - k\right), \qquad \{u,v\} \in S(G), \tag{3.4}$$

$$f_e(u,v) = C_e \frac{weight(u) \cdot weight(v)}{||x_u - x_v||^2}, \quad u \neq v \wedge u,v \in V(G) \tag{3.5}$$

where x_u and x_v are positions of the nodes u and v on the plane and the weights of u and v are used to represent their electric charge. $S(G)$ once again is the set of springs in the graph—in this case between pairs of adjacent nodes. This similar to what Eades and Di Battista et al. propose in [12 and 15].

Given these functions, the combined force on a node u can be expressed as

$$f(u) = \sum_{\substack{v \in V(G) \\ u \neq v}} f_e(u,v) \frac{x_v - x_u}{||x_v - x_u||} + \sum_{\{u,v\} \in S(G)} f_s(u,v) \frac{x_v - x_u}{||x_v - x_u||} \tag{3.6}$$

which turns the scalar forces f_s and f_e into vector forces and gives the overall force a direction. Note that if we set $f_e(u,v) = 0$ for all $u,v \in V(G)$, the same formula can be used for the combined force in spring algorithms.

3.2.3 The Problem of Overlapping Nodes

In order to avoid overlapping nodes, most implementations of spring and spring-electrical algorithms use a simple trick. If two nodes are located at the same position, one of them is temporarily displaced by a small random amount. This usually has a strong repulsive effect that will push the two nodes apart. All algorithms implemented in this thesis support this feature.

Another problem is that the above force function disregard the size of nodes. This is fine if nodes are uniformly sized and the natural spring dimension k is chosen large enough. In practical scenarios where nodes may contain numbers, words as well as entire paragraphs of text, the former is often not the case. In such graphs with arbitrarily sized nodes, disregarding the size may introduce undesired overlaps. To overcome this problem, several advanced methods haven been proposed [24, 29 and 34]. Straight-forward extensions of f_s, f_e and f that take the shape of nodes into account are also possible. The algorithms implemented in this thesis ignore the node size, leaving solutions up to future development. Possible approaches are described in chapter 6.

3.2.4 Multilevel Algorithm Structure

The common structure of the three algorithms implemented in the thesis is a result of using the same iterative force simulation approach and multilevel technique in order to avoid settling for local minima. This technique consists of the following steps:

1 Iterative graph coarsening
2 Computation of an initial force-based layout
3 Iterative graph expansion and force-based layout refinement

In the coarsening phase, the input graph is repeatedly reduced to coarse graphs $G_1, \ldots, G_l$, where G_1 is the input graph and G_l is first graph whose number of nodes reaches or drops below the threshold *min_size*. Several techniques to reduce the number of nodes are discussed in [31 and 51]. Walshaw proposes to collapse a maximal set of independent edges in G_i to generate the next coarse graph G_{i+1}. A graph partitioning algorithm by Hendrickson and Leland [30] is used to find a maximal edge matching and collapse these edges. The weights of the two end nodes of each collapsed edge are merged and their weights are summed up. In a similar way the weights of edges adjacent to both end nodes of a collapsed edge are updated. The maximal matching is constructed by visiting unmatched nodes in random order and picking the edge that connects them to an unmatched neighbor with minimum weight.

A similar coarsening technique is adopted by Hu [31]. The only difference is that the maximal matching is constructed by selecting heavy edges rather than edges to light neighbors. Hu compares this technique to a method that uses the nodes of a maximum independent vertex set for the next coarse graph. An implementation of both methods is intended in Ti*k*Z; however, at the time of writing only collapsing edges is implemented as it tends to produce slightly better results [31]. The light-vertex matching method proposed by Walshaw is not implemented as it is stated to produce similar results [31].

Once all coarse graphs have been generated, the coarsest graph G_l is laid out randomly. Even though the initial layout has a strong impact on the size and edge lengths in the final drawing, neither Hu nor Walshaw specify how exactly it is to be generated. After some experimentation with different random layout generators, it became obvious that the `Hu2006` and `Walshaw` algorithms react different to the initial with regards to scaling. The Ti*k*Z implementation of the three algorithms includes various modifications to cope with these differences and to produce drawings with similar size and edge lengths.

After the initial random layout has been computed, the regular spring or spring-electrical algorithm is applied to the coarsest graph. Hu and Walshaw present different variations of such an algorithm, which are explained in sections 3.3.2, 3.3.1 and 3.3.3. These variations are similar in structure but differ mainly with regards to the force and cooling functions being used.

Following this initial layout, the coarsening steps are reverted step by step. When the graph G_{l-1} is reconstructed from G_l, the positions of nodes in G_{l-1} are interpolated from the corresponding supernodes in G_l. Usually, all nodes $u \in V(G_{l-1})$ that correspond to a node $v \in V(G_l)$ are assigned the same coordinate as v. After reconstruction, a spring or spring-electrical layout is computed for the coarse graph G_{l-1}, applying the same force-based algorithm that is used to arrange the initial layout. This procedure—graph expansion and force-based layout refinement—is repeated for each coarse graph $G_{l-2}, G_{l-3}, \ldots, G_1$. The multilevel algorithm returns the final layout computed for the graph $G = G_1$.

If the coarsening is disabled, only step 2 is performed, that is, an initial random layout is computed followed by the regular force-based algorithm. This multilevel structure that all three algorithms implemented have in common is shown as pseudocode in listing 3.1. The differences are the computation of the random initial layout, the choice of the spring force f_s, the electric force f_e, the node movement MOVE-NODE-ALONG-FORCE, and the step size control UPDATED-STEP-SIZE.

3.2.5 Approximation of Electric Forces

In order to reduce the running time required to compute electric forces in the system, Hu [31] uses the tree-based quadtree algorithm proposed by Barnes and Hut [2]. Walshaw employs a different technique based on ideas by Fruchterman and Reingold [21]. The Ti*k*Z spring-electrical implementation of the Hu and Walshaw algorithms both use the Barnes–Hut technique. Note that, due to the absence of electric forces, this feature is not applicable and therefore not available in the implementation of Hu's spring algorithm.

Modeling and computing physical forces in a system of particles is known as N-body simulation. In order to obtain an exact solution for such a simulation, the forces between all pairs of

■ **Listing 3.1** Common structure of the spring and spring-electrical algorithms implemented in this thesis. The parameters *min_size* and *min_ratio* are used to decide when to stop coarsening the input graph. *step_size* is used to control the node movements. The maximum node movement is compared against the natural spring dimension k as a convergence criterion with a tolerance of *tol*.

```
MULTILEVEL-ALGORITHM(G_i, ...)
1   if coarsening disabled or |V(G_i)| ≤ min_size or (i > 1 and |V(G_i)|/|V(G_{i-1})| ≤ 1 − min_ratio) then
2       GENERATE-RANDOM-LAYOUT(G_i, ...)
3       SPRING-OR-SPRING-ELECTRICAL-ALGORITHM(G_i, ...)
4   else
5       G_{i+1} ← COARSEN(G_i, ...)
6       MULTILEVEL-ALGORITHM(G_{i+1}, ...)
7       INTERPOLATE-COORDINATES(G_{i+1}, G_i)
8       SPRING-OR-SPRING-ELECTRICAL-ALGORITHM(G_i, ...)

SPRING-OR-SPRING-ELECTRICAL-ALGORITHM(G, ...)
1   converged ← false
2   iterations ← 0
3
4   while iterations < max_iter and not converged do
5       iterations ← iterations + 1
6
7       for each node u ∈ V(G) do
8           f ← 0⃗
9           for each spring {u,v} ∈ S(G) do
10              f ← f + f_s(u, v) · (x_v − x_u) / ||x_v − x_u||
11          for each node v ∈ V(G) with u ≠ v do
12              f ← f + f_e(u, v) · (x_v − x_u) / ||x_v − x_u||
13
14          x_u ← MOVE-NODE-ALONG-FORCE(u, f, step_size, ...)
15
16      step_size ← UPDATE-STEP-SIZE(G, step_size, ...)
17
18      if maximum node displacement ≤ (k · tol) then
19          converged ← true
```

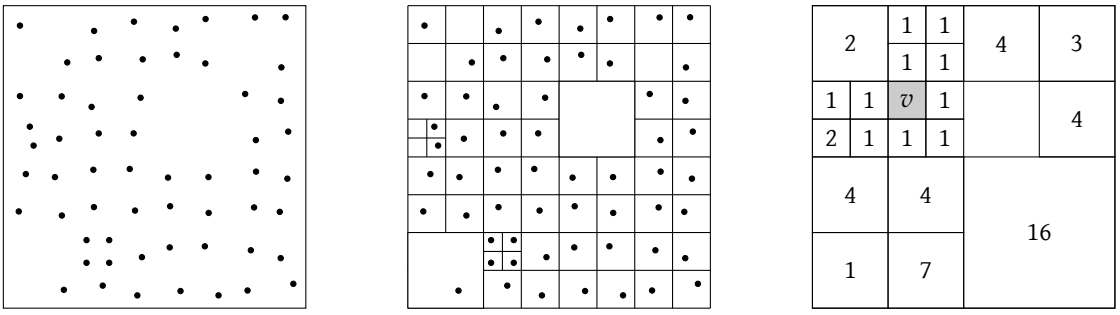
particles need to be computed, which has a time complexity of $O(N^2)$. N-body simulations are often used for analyzing systems with many particles. A popular application is the calculation of gravitational forces in planetary systems and the prediction of movements of objects in space.

To perform simulations of large systems a feasible running time, Barnes and Hut propose an algorithm that approximates electric forces in $O(N \log N)$ time. Its basic idea is to represent

the mass distribution of the system as a hierarchical tree structure of cells, each of which holds either a single particle or a list of subcells together with cumulated information about the total mass and gravitational center of particles in the subcells. The hierarchical cell structure is constructed as follows. First, an empty root cell is created that is large enough to contain all nodes. Particles representing the individual nodes are inserted one by one. As soon as a cell contains more than one particle, it is divided into four uniformly sized subcells. Its particles are moved into these subcells according to their position. If this process is continued until all particles have been inserted, the resulting cell hierarchy will have only empty inner cells with subcells and leaf cells to hold at most one particle. In a final step, the total masses and centers-of-mass are propagated and cumulated from the leaf cells up to the root cell such that every cell is tagged with the total mass and center-of-mass of its cell hierarchy.

Given this construction, the force on a particular node is approximated by collecting cells from the hierarchy in an *interaction list*. Instead of computing the electric force between the node and all other nodes, only the repulsion caused by cells in this interaction list—which has an average size of $log|V(G)|$—is taken into account. The interaction list is allocated empty and is filled using a recursive traversal of the cell hierarchy, starting at the root cell. Barnes and Hut propose the following criterion for choosing which cells to add to the list. Let l be the length of the cell currently being processed and D the distance between the cell's center-of-mass and the node particle p. The cell is added to the interaction list if it is a leaf or if $l \,/\, D < \theta$ where θ is a parameter to control the accuracy of the approximation. Cells that do not meet this criterion are resolved into their four subcells, each of which is recursively examined in the same way. The construction of the cell hierarchy as well as the force approximation are illustrated in figure 3.1.

■ **Figure 3.1** Construction of hierarchical cells in the Barnes–Hut algorithm. (i) shows an almost uniform distribution of equally weighted particles on the plane. The cell hierarchy created for this distribution is illustrated in (ii). (iii) demonstrates which of the cells are used to compute forces on particle v along with the number of particles they represent. The higher the number the more computations are saved.



(i) (ii) (iii)

Due to its approximative nature the Barnes–Hut algorithm may compute forces inaccurately and therefore result in drawings of lower quality. The effects are critical with small graphs in particular where deviations from perfect layouts may easily disturb authors and viewers alike. However, this optimization can lead to significant speedups and thus becomes valuable when laying out large graphs.

3.3 Spring and Spring-Electrical Algorithms Implemented for Ti*k*Z

In this section, the algorithms written for Ti*k*Z are discussed in detail, including a coverage of implementation details and modifications. All algorithms are based on the multilevel structure and common elements presented in the previous section. Thus, only the differences are described here.

3.3.1 Hu's Spring-Electrical Algorithm

Choice of Spring and Electric Forces

The algorithm presented by Hu [31] is based on the spring-electrical model. Inspired by earlier work of Fruchterman and Reingold [21], Hu defines the forces as

$$f_s(u,v) = \frac{\|x_v - x_u\|^2}{k}, \tag{3.7}$$

$$f_e(u,v) = \frac{-Ck^2}{\|x_v - x_u\|}. \tag{3.8}$$

These forces bare little resemblance to those given by Hooke's and Coulomb's law (formula 3.4) at first sight. This is because they already incorporate and compensate the interaction of attraction and repulsion. The forces $f_s(u,v)$ and $f_e(u,v)$ cancel each other out exactly if the nodes u and v have the desired distance k and $C = 1$.

Drawings generated by spring-electrical algorithms suffer from peripheral distortion, meaning that in the periphery tend to be closer to each other than nodes in the center. This becomes particularly visible with mesh-like structures in the form of a magnifier effect. Hu provides an in-depth analysis of this problem, identifying strong long range electric forces as the main reason for the distortion. To overcome this, he proposes a general electric force model defined as

$$f_e(u,v) = \frac{-Ck^{1+p}}{\|x_v - x_u\|^p}, \quad p > 0. \tag{3.9}$$

The larger p, the weaker the long range electric force and therefore the peripheral distortion. This model is an generalization of the previous definition of f_e, which is obtained by setting $p = 1$.

The consideration of node weights and Barnes–Hut cell masses when computing the force between either two nodes or a node and a cell is important for the system energy optimization to work properly. Throughout [31], node weights are assumed to be 1. This implies that the mass of a Barnes–Hut cell is the number of nodes it contains. Inspired by Walshaw, I decided to support arbitrary node weights by including the node weight and cell mass in f_e as a factor:

$$f_e(u,v) = \frac{-C \cdot \mathit{weight}(v) \cdot k^{1+p}}{||x_v - x_u||^p}, \quad p > 0. \tag{3.10}$$

When using the Barnes–Hut method, $\mathit{weight}(v)$ is replaced with the mass of individual cells used to approximate the electric forces on u. In this case, the mass of a cell is defined as the sum of weights of the nodes contained in the cell and all its subcells. Instead of x_v, the center-of-mass of the cell is used. This modification also allows users to specify weights for individual nodes, which is useful in situations where nodes need to be isolated visually. Note that, unlike in formula 3.4, only the weight of v needs to be included in $f_e(u,v)$ since the electric force between u and v is computed twice—the second time as $f_e(v,u)$, which includes the weight of u.

Node Movement

Once the total force f on a node u has been computed (line 8–12 in listing 3.1), the node is moved along this force. The force itself is a two-dimensional vector with a direction and a length, the latter of which represents the strength of the force. One would expect that this strength is reflected in the node's movement. Surprisingly, Hu chose to move all nodes by the same amount—*step_size*—and interpret the force only as a direction vector [31]. Thus, in `Hu2006 spring electrical`, the node movement is implemented as follows.

MOVE-NODE-ALONG-FORCE(u, f, *step_size*)
1 return $x_u + \mathit{step\_size} \cdot f \;/\; ||f||$

Again, this implies that despite forces of varying strength, all nodes are moved by the same amount. In [31], Hu provides no reasons or evidence for this decision.

Step Size Control and Convergence

Nodes are moved by the amount specified using the step size *step_size*. In order to reducing the movements over time and converge towards a minimum energy layout, this step size is updated

after every iteration (line 16 in listing 3.1). This is called *cooling*. The cooling function by which *step_size* is updated needs to support escaping local minima as well as a quick convergence of the algorithm. Many techniques are discussed in [21] including simulated annealing, sintering, simmering, and quenching.

Hu uses two different cooling functions [31]. In the refinement phase of the multilevel algorithm a simple scheme is employed where *step_size* is gradually reduced by a $cooling\_factor < 1$ until it is zero:

UPDATE-STEP-SIZE(*step_size*, *cooling_factor*)
1 return $step\_size \cdot cooling\_factor$

Hu proposes to set $cooling\_factor = 0.9$, which is stated to be adequate enough if the layout to be refined is not entirely random.

When applying the regular spring-electrical algorithm to a random initial layout, however, a more advanced *adaptive cooling* method is employed. Its central idea is to keep the step size unchanged if the total system energy is being reduced once but to increase the step size after five consecutive improvements. The step size is reduced only if the energy increases. This adaptive method requires a modification to the energy optimization loop (lines 4–16 in listing 3.1). Three variables are introduced, called *progress*, *energy_prev*, and *energy* that hold the number of consecutive improvements and the computed system energy after the previous and current iteration, respectively. These variables are passed to UPDATE-STEP-SIZE, which looks as follows.

UPDATE-STEP-SIZE(*step_size*, *progress*, *cooling_factor*, *energy_prev*, *energy*)
1 **if** $energy < energy\_prev$ **then**
2 $progress \leftarrow progress + 1$
3 **if** $progress \geq 5$ **then**
4 $progress \leftarrow 0$
5 $step\_size \leftarrow t \;/\; cooling\_factor$
6 **else**
7 $progress \leftarrow 0$
8 $step\_size \leftarrow step\_size \cdot cooling\_factor$
9 return *step_size*, *progress*

The total system energy is defined as the squared sum of the combined forces (3.6) on all nodes

$$energy = \sum_{u \in V(G)} f^2(u). \tag{3.11}$$

and is computed along with the forces and movements.

Hu further states that a suitable choice of the natural spring dimension k and the constant C is helpful in converging towards a minimum energy layout. Ideally, these parameters reflect the scaling and node distances in the initial layout. Throughout [31], C is fixed at $C = 0.2$ while k is held variable.

Initial Layout and Scaling

In Hu's spring-electrical algorithm, the initial layout is generated randomly. No details are given about the scaling of this initial layout. A common technique used in [21 and 51] is to generate a random initial layout and then set the natural spring dimension k to the average length of all edges. This, however, implies that users only have indirect control over the scaling and natural spring dimension by changing the range of the random values.

Clearly, a more intuitive and predictable approach is to choose a random distribution that results in an average node separation of k in the final layout and let the user choose k. This is roughly what the `Hu2006 spring electrical` implementation in TikZ does. While experimenting with a variety of initial placements I noticed that, for dense graphs like K_n, the algorithm tends to generate compressed-looking drawings if the initial layout is scaled poorly. To overcome this problem, my implementation of `Hu2006 spring electrical` distributes nodes in a range that is proportional to the desired natural spring dimension k and the density of the graph, denoted by $density(G)$. More precisely, all random initial positions lie in an area defined by

$$pos(v) = (random(-r, +r), random(-r, +r)), \quad v \in V(G) \tag{3.12}$$

where r is a scaling factor proportional to the graph density:

$$r = 3k \cdot density(G) \cdot \sqrt{|V(G)|} \,/\, 2. \tag{3.13}$$

After the initial layout has been generated, the natural spring dimension specified by the user and used for the random distribution is replaced with the average edge length in the initial layout. Even though this choice still does not guarantee an average edge length of k in the final drawing, it works reasonably well with dense and sparse graphs alike. If the input graph or the coarsest graph contains only two nodes, one is placed at (0,0) while the other is placed r apart in a random direction. This initial placement may appear somewhat artificial and other techniques may yield even better results. However, taking the density of the input graph into account seems to make a notable difference and is this worth to investigated further in future research.

When reverting the graph coarsening from a coarse graph G_i to a less coarse graph G_{i-1}, a naive interpolation of node coordinates and application of the regular spring-electrical algorithm to G_{i-1} may destroy useful properties inherited from G_i. In order to avoid this, Hu proposes to scale the coordinates of G_{i-1} after interpolation from G_i and before applying the regular spring-electrical algorithm [31]. The factor s_{i-1} that he uses to scale the coordinates of G_{i-1} is defined as

$$s_{i-1} = diam(G_{i-1}) \;/\; diam(G_i), \tag{3.14}$$

where $diam(G)$ denotes the pseudo-diameter of a graph G, that is, an approximation of the longest shortest path from any node to another. There are many ways to compute the pseudo-diameter, including such versatile methods as the one proposed by Paulino et al. [45]. My implementation of Hu's spring-electrical algorithm finds the pseudo-diameter by starting at a node u with minimum degree. It computes the graph distances to all other nodes and from the nodes that are farthest away it picks the node v with the smallest degree. The graph distance between u and v is stored as the temporary pseudo-diameter. This process is repeated with v as the new starting node until the pseudo-diameter cannot be increased further. While this algorithm fails to always find an exact solution, it is expected to converge quickly.

3.3.2 Hu's Spring Algorithm

Choice of Spring and Electric Forces

The second algorithm by Hu [31] is based on the spring model by Kamada and Kawai [35]. Springs are attached to any pair of nodes in the graph, with the ideal distance being proportional to the graph distance between the two nodes. This has two implications. Firstly, there is no electric force between nodes, that is, $f_e(u,v) = 0$ for all pairs $u,v \in V(G)$. The spring force is expressed as in formula 3.2:

$$f_s(u,v) = \|x_v - x_u\| - k \cdot d(u,v), \quad \{u,v\} \in S(G). \tag{3.15}$$

This model has no analogy for electrically charged nodes and thus, node weights are not supported by f_s. The second implication is that, since there are no electric forces, the Barnes–Hut algorithm cannot be applied, which means that the runtime cannot be reduced below $O(|V(G)|^2)$ per iteration. In my benchmarks I found the standard definition of f_s to work well enough to not require any modifications.

Node Movement, Step Size Control and Convergence

Nodes are moved exactly as in Hu's spring-electrical algorithm. Furthermore, the same cooling schemes and convergence criteria are applied.

Initial Layout and Scaling

The initial layout generated by `Hu2006 spring` differs from that produced by `Hu2006 spring electrical` only in terms of scaling parameters. In the experiments performed, drawings generated by `Hu2006 spring` were notably larger than spring-electrical layouts of the same graphs. Thus, order to produce drawings of similar scaling, a different range is used for the random distribution in `Hu2006 spring`, that is, r is defined as

$$r = 2k \cdot density(G) \cdot \sqrt{|V(G)|} \,/\, 2. \tag{3.16}$$

Like in `Hu2006 spring electrical`, the natural spring dimension used internally is set to the average length of edges in the initial layout once it has been generated.

3.3.3 Walshaw's Spring-Electrical Algorithm

Choice of Spring and Electric Forces

The spring-electrical algorithm presented by Walshaw [51 and 52] simulates the same electric force as is used by Hu's algorithm (see formula 3.7). However, Walshaw's original formula already incorporate the node weight as linear factor. Like with `Hu2006 spring electrical`, my `Walshaw2000 spring electrical` implementation of replaces this basic electric force with the general electric force model from formula 3.10 in order to give users control over the peripheral distortion:

$$f_e(u,v) = \frac{-C \cdot weight(v) \cdot k^{1+p}}{||x_v - x_u||^p}, \quad p > 0. \tag{3.17}$$

The spring force used by Walshaw differs from the one employed in [31] mainly in that the electric force between adjacent nodes is subtracted from the spring force, meaning that electric forces only act between nodes that are not connected. Walshaw further extends the spring force arguing that the contribution of the spring force between a node u and its neighbor v is $(||x_v - x_u|| - k) \,/\, |\Gamma(u)|$ in the direction of v, where $\Gamma(u)$ denotes the set of neighbors of u. Another

difference of the spring force used by `Walshaw2000 spring electrical` is that the deviation from the natural spring dimension is not squared:

$$f_s(u,v) = \frac{||x_v - x_u|| - k}{|\Gamma(u)|} - f_e(u,v), \quad \{u,v\} \in S(G). \tag{3.18}$$

There are two problems with the above forces. One is that the absence of electric forces between adjacent nodes makes it impossible to implement an electric charge option that users can adjust for each node. A node with high electric charge would only push non-neighbor nodes apart while its neighbors would remain at a close distance, which would come unexpected for users. In the implementation, this problem is worked around by not subtracting $f_e(u,v)$ if the node u has a charge $weight(u) \neq 1$. The other issue is that $f_e(u,v)$ is not computed for all pairs u and v of nodes when the electric forces are approximated using the Barnes–Hut algorithm. An incomplete workaround for this was implemented in TikZ that only subtracts the electric force $f_e(u,v)$ if the cell holding only v has an influence on the overall force on u. The electric force on u caused by cells containing more than one node is assumed to be 0 in f_s.

Node Movement, Step Size Control and Convergence

Unlike Hu, Walshaw proposes to use not only the direction but also the strength of the combined forces when calculating the movement of a node. In his algorithm, the step size is an upper limit for this movement:

MOVE-NODE-ALONG-FORCE($u, f, step\_size$)
1 return $x_u + f \;/\; ||f|| \cdot \text{MIN}(step\_size, ||f||)$

Walshaw applies a simple conservative rather than adaptive cooling function that gradually reduces the step size and thereby the node movement:

UPDATE-STEP-SIZE($step\_size$)
1 return $0.95 \cdot step\_size$

Initial Layout and Scaling

Like Hu, Walshaw generates the initial layout randomly. My implementation of Walshaw's algorithm uses the same random value distribution as in formulas 3.12 and 3.13. The two algorithms differ in that Walshaw scales the natural spring dimension rather than scaling the node positions when reversing the coarsening process. When interpolating the graph G_i from the coarser graph G_{i+1}, the natural spring dimension k_i used for G_i is computed as follows:

$$k_i = \sqrt{4 \,/\, 7} \cdot k_{i+1}. \tag{3.19}$$

For the coarsest graph G_l, Walshaw sets k_l to the average edge length in the random initial layout. To make the scaling of the generated drawings more predictable, I decided to set

$$k_l = k \,/\, \left(\sqrt{4 \,/\, 7}\right)^l, \tag{3.20}$$

which results in a natural spring dimension $k_1 = k$ for the original input graph.

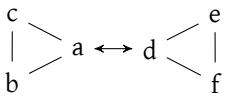
3.4 TikZ Syntax and Options for Creating Spring and Spring-Electrical Layouts

3.4.1 Syntax to Enable Spring and Spring-Electrical Layouts

Like all families of graph drawing algorithms, implementations of force-based algorithms are grouped in a dedicated TikZ library, called `graphdrawing.force`. Loaded together with the `graphs` and `graphdrawing` libraries, `graphdrawing.force` makes all force-based algorithms available to the user.

Spring layouts can be activated using the `/graph drawing/spring layout` TikZ option. This applies the default spring algorithm to the graph in question:

```
\tikz \graph [spring layout,orient=a-d] {
  { [clique] a, b, c },
  { [clique] d, e, f },
  a <-> d,
};
```

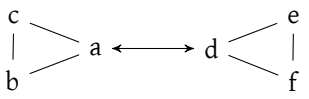


The default spring algorithm is `Hu2006 spring`. As soon as other spring algorithms are added to TikZ, the default algorithm used for `spring layout` can be changed like this:

```
\pgfkeys{
  /graph drawing/spring layout/default algorithm=⟨other algorithm⟩
}
```

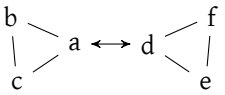
Spring-electrical layouts are created in the same manner using the `/graph drawing/spring electrical layout` option as is shown below.

```
\tikz \graph [spring electrical layout,
              orient=a-d] {
  { [clique] a, b, c },
  { [clique] d, e, f },
  a <-> d,
};
```



The default spring-electrical algorithm is `Hu2006 spring electrical`, which can be changed to `Walshaw200 spring electrical` or any other spring-electrical graph drawing algorithm by setting the `/graph drawing/spring electrical layout/default algorithm` option. Both, spring and spring-electrical algorithms may also be activated by entering their name rather than using one of the two `layout` options.

```
\tikz \graph [Walshaw2000 spring electrical,orient=a-d] {
  { [clique] a, b, c },
  { [clique] d, e, f },
  a <-> d,
};
```



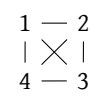
The spring and spring-electrical drawing algorithms are very similar in terms of their parameters and the constraints they can handle. They thus share a number of common Ti*k*Z options for fine-tuning. These options are split up into graph options that can be applied once to the entire graph and node options that can be specified for individual nodes.

3.4.2 Common Graph Parameters

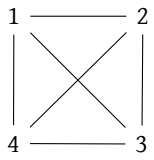
Length of Springs

One of the key parameters with the strongest and most predictable effect on the scaling of spring and spring-electrical layouts is the natural spring dimension *k*. It defines the equilibrium length of edges in the drawing and can be changed in Ti*k*Z using the `natural spring dimension` option. The following example shows how a simple graph can be scaled by changing the natural spring dimension:

```
\tikz \graph
  [spring layout={natural spring dimension=0.5cm},
   orient=1-2] { subgraph K_n[n=4] };
```



```
\tikz \graph
  [spring layout={natural spring dimension=1.1cm},
   orient=1-2] { subgraph K_n[n=4] };
```

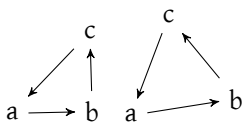


Changing the natural spring dimension may have unexpected side-effects such as a change in rotation or node positions, which is why the above example uses the `orient` option to fix the orientation of the two drawings. When used with spring algorithms, the `natural spring dimension` specifies the desired length of individual edges rather than the dimension of springs, which may span multiple edges if the graph distance between its nodes is greater than 1. The option is capable of parsing mathematical expressions and coordinates and is set to 1cm by default.

Step Size Update and Convergence Control

As can be seen in listing 3.1, minimizing the system energy by computing the forces and movements of the nodes is an iterative process. Depending on the characteristics of the input graph and the parameters chosen for the spring or spring-electrical algorithm, minimizing the system energy may require many iterations. In these situations it may come in handy to limit the number of iterations. By default, the upper limit for the number of iterations is set to be 500. This can be changed, however, using the `iterations` option, as is shown below.

```
\tikz \graph [spring layout={iterations=10}]
  { a -> b -> c -> a };
\tikz \graph [spring layout={iterations=100}]
  { a -> b -> c -> a };
```



This example demonstrates that the number of iterations has a strong impact on the quality of the generated drawings. Limiting this number to something small may improve the running time of the algorithm, which may be critical with large graphs. This of course comes at the cost of drawings of lower quality.

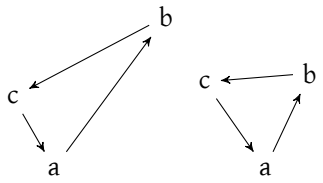
Apart from reducing running times, another use case of the `iterations` option is to draw the same graph multiple times with a different number of iterations and thereby demonstrate how the spring or spring-electrical algorithm improves drawings step by step. This may prove to be useful when debugging new algorithms or when presenting and explaining them to others.

The initial value of the step size that limits or controls the node movement along its force can be changed with the `initial step dimension` option. Like `natural spring dimension` this

parameter is able to parse mathematical expressions. If set to its default value 0, the natural spring dimension is used as the initial step size internally.

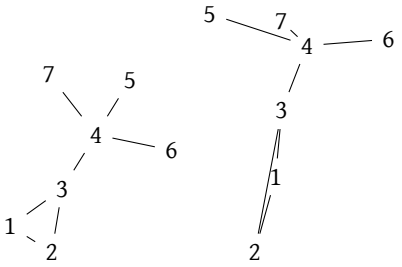
Another parameter that controls how layouts evolve over time is the cooling factor. It is used to gradually reduce the step size between one iteration to the next. A small positive cooling factor ≥ 0 means that the movement of nodes is quickly or abruptly reduced, while a large cooling factor ≤ 1 allows for a smoother step by step layout refinement at the cost of more iterations. The following example demonstrates how a smaller cooling factor may result in a less balanced drawing. By default, `Hu2006 spring`, `Hu2006 spring electrical`, and `Walshaw2000 spring electrical` use a cooling factor of 0.95.

```
\tikz \graph [spring layout={cooling factor=0.1}]
  { a -> b -> c -> a };
\tikz \graph [spring layout={cooling factor=0.5}]
  { a -> b -> c -> a };
```



All spring and spring-electrical algorithms implemented in the thesis terminate as soon as the maximum movement of any node drops below $k \cdot tol$. This tolerance factor can be changed with the `convergence tolerance` option:

```
\tikz \graph [
  spring layout={convergence tolerance=0.001}
] { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 }
};
\tikz \graph [
  spring layout={convergence tolerance=1.0}
] { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 }
};
```



The above example illustrates that a high tolerance may result in poorly arranged drawings. Therefore, `Hu2006 spring` and `Hu2006 spring electrical` both use a default tolerance of 0.01. `Walshaw2000 spring electrical` algorithm slightly differs from this by using 0.001 as the default value.

Multilevel Algorithm Configuration

In order to compare the results, users might want to switch between the regular spring or spring-electrical algorithm and the multilevel version. Toggling the graph coarsening process

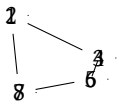
is possible using the boolean `coarsen` option, which can be either `false`, `true` or empty. If unset or empty, all three algorithms will default to assuming that coarsening is enabled and will apply the multilevel technique. The difference is often barely noticeable when drawing small graphs. However, with larger graphs, disabling coarsening usually reduces the running time of the algorithm at the cost of low-quality drawings. For such an example see chapter 5.

The minimum number of nodes down to which the graph is coarsened can be set with the `coarsening/minimum graph size` option. The first graph that has a smaller or equal number of nodes becomes the coarsest graph G_l , where l is the number of coarsening steps. The algorithm proceeds with the steps described earlier in this chapter.

In the following example the same graph is coarsened down to two nodes the first time and four nodes the second time. The layout of the original graph is interpolated from the random initial layout and is not improved further because the forces are not computed (0 iterations). Thus, in the two graphs, the nodes are placed at exactly two and four coordinates in the final drawing:

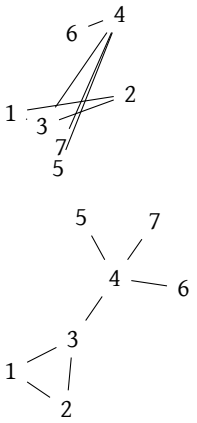
```
\tikz \graph [
  spring layout={
    iterations=0,
    coarsening={minimum graph size=2}
  }
] { subgraph C_n [n=8] };
```

```
\tikz \graph [
  spring layout={
    iterations=0,
    coarsening={minimum graph size=4}
  }
] { subgraph C_n [n=8] };
```



Another parameter that controls how far the graph is coarsened is the `coarsening/downsize ratio` option. It specifies the percentage by which the size of the graph needs to be reduced in order for coarsening to continue. The default downsize ratio is 0.25, meaning that coarsening will stop as soon as a coarse graph G_i does not have at least 25% fewer nodes than its parent G_{i-1}. Increasing this option potentially reduces the number of coarse graphs computed during the coarsening phase. This may speed up the algorithm but if the size of the coarsest graph G_l is much larger than the `minimum graph size`, the benefits of the multilevel approach may vanish:

```
\pgfgdset{
  spring electrical layout/default algorithm=%
    Walshaw2000 spring electrical
}

\tikz \graph [spring electrical layout={
    coarsening={downsize ratio=1.0},
}] { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 } };

\tikz \graph [spring electrical layout={
    coarsening={downsize ratio=0.2},
}] { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 } };
```



During the coarsening phase, the number of nodes in the graph is repeatedly reduced using a coarsening scheme such as `coarsening/collapse independent edges` or its alternative, the `coarsening/connect independent nodes` scheme. If `collapse independent edges` is enabled, which happens to be the default setting, coarse versions of the input graph are generated by finding a maximal independent edge set and by collapsing the edges from this set. Nodes adjacent to each of these edges are merged into supernodes by which they are replaced in the next coarse version of the graph. Edges and nodes that are not related to the maximum independent edge set are maintained in the new graph.

Collapsing the edges of a maximum independent edge set reduces the number of nodes of the graph by up 50%—but never more than that. This means that the upper limit for reasonable values of `downsize ratio` to be used in combination with `collapse independent edges` is 0.5. Compared to `connect independent nodes` this is the less aggressive coarsening scheme and typically yields better drawings. At the time of writing, only `collapse independent edges` is implemented.

Miscellaneous

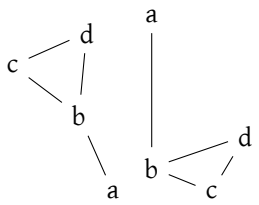
Drawings produced by spring and spring-electrical algorithms may occasionally appear unfinished or visually twisted. Asymmetry and undesired edge crossings are often caused by an unfavorable random distribution in the initial layout. In order to improve or unravel such drawings, the seed used for Lua's pseudo-random number generator may be changed in Ti*k*Z using the `random seed` option. The effects of changing the random seed are unpredictable, so experimenting with different values is sometimes necessary. The below example shows how to set the random seed to a different value, the default being 42:

```
\tikz \graph [spring layout={random seed=1}] { ... };
```

3.4.3 Common Node Parameters

Fixing nodes at certain positions is a constraint that can be useful in many situations. A TikZ node option that provides such a feature is `/graph drawing/desired at`. This is a global option that is not only available in force-based algorithms. It enables users to define a desired position for individual nodes and is supported by the three spring and spring-electrical algorithms implemented in the thesis. Here is a demonstration of this feature:

```
\tikz \graph [spring layout] {
  a -- b,
  { [clique] b, c, d }
};
\tikz \graph [spring layout] {
  a [desired at={(0,0)}] -- b [desired at={(0,-2cm)}],
  { [clique] b, c, d }
};
```

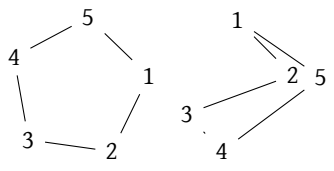


3.4.4 Dedicated Options for Spring-Electrical Layouts

Computing the electric forces in a graph G accurately requires $O(|V(G)|^2)$ operations in each iteration of spring-electrical algorithms. With `approximate electric forces` set to `true`, electric forces are approximated using the Barnes-Hut algorithm described in section 3.2.5. This reduces the number of operations needed to compute the electric forces to $O(|V(G)| \cdot \log|V(G)|)$ per iteration, which may improve the running time of spring-electrical algorithms significantly.

However, this optimization often comes at the cost of less appealing drawings which is noticeable in particular with small graphs. Even though the differences can be very subtle, the often reduced quality is the reason why this feature is turned off by default and only makes sense to speed up rendering with large graphs. The following code provides an example where the degraded quality of a drawing with electric force approximation becomes obvious:

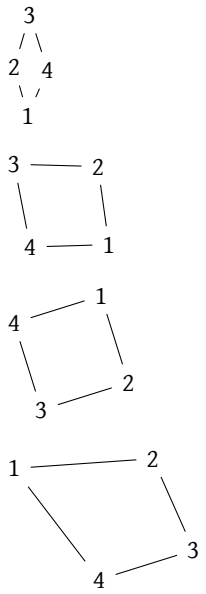
```
\tikz \graph [spring electrical layout]
  { subgraph C_n[n=5] };
\tikz \graph [spring electrical layout={
  approximate electric forces
}] { subgraph C_n[n=5] };
```



Sometimes when drawing symmetric and mesh-like graphs, the peripheral distortion caused by long-range electric forces may be undesired. The general electric force model proposed by Hu allows to reduce long-range forces and distortion effects by increasing the order of electric forces (see formula 3.10). In the `Hu2006 spring electrical` and `Walshaw2000 spring electrical` algorithms, this order can be changed with the `electric force order` option, which defaults to 1. Values between 0 and 1 increase long-range electric forces and the scaling of the generated layouts. Value greater than 1 decrease long-range electric forces and results in shrinking drawings.

Spring-electrical algorithms support a special node option called `electric charge` that gives users the chance to adjust the electric charge of individual nodes. This charge is applied as a linear factor to the electric force between a node and all others. Its default value is 1, meaning that all nodes have the same charge that does not affect the electric force (see e.g. formula 3.10). A value greater than 1 will cause a stronger repulsion between a node and others while negative values will attract other nodes rather than repulsing them. This effect is demonstrated in the following example.

```
% negative electric charge, attractive effect
\tikz \graph [spring electrical layout] {
  1 [spring electrical layout={electric charge=-5}],
  1 -- 2 -- 3 -- 4 -- 1,
};

% zero charge, reduced repulsive effect
\tikz \graph [spring electrical layout] {
  1 [spring electrical layout={electric charge=0}],
  1 -- 2 -- 3 -- 4 -- 1,
};

% charge of 1, uniform electric forces
\tikz \graph [spring electrical layout] {
  1 [spring electrical layout={electric charge=1}],
  1 -- 2 -- 3 -- 4 -- 1,
};

% positive charge > 1, repulsive effect
\tikz \graph [spring electrical layout] {
  1 [spring electrical layout={electric charge=5}],
  1 -- 2 -- 3 -- 4 -- 1,
};
```

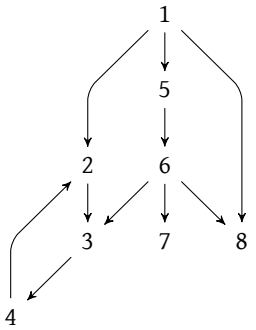


4

Algorithms for Layered Drawings of Directed Graphs

The second family of algorithms investigated in the thesis are algorithms for producing layered drawings of directed graphs. For comparison with the spring and spring-electrical layouts covered in the previous chapter, a layered drawing is shown below.

```
\tikz \graph [layered drawing] {
  1 -> 2 -> 3 -> 4,
  1 -> 5 -> 6,
  6 -> 7,
  6 -> 8,
  1 -> 8,
  6 -> 3,
  4 -> 2,
};
```



Section 4.1 provides a detailed introduction to the Sugiyama method, a commonly used conceptual framework for layered drawing algorithms. It also gives an overview of various heuristics that have been developed for the individual steps of this method. As part of the thesis, the Sugiyama framework was implemented as a modular algorithm that can be extended with new solutions easily. This modular implementation is described in section 4.2. The chapter closes with a description of the Ti*k*Z options added to configure the modular algorithm in section 4.3 and a presentation of the extension interface in section 4.4.

4.1 An Algorithm Framework Based on the Sugiyama Method

Like undirected graphs, directed graphs can be arranged with general-purpose methods such as force-based algorithms. However, the direction of edges usually encodes additional information that is worth paying attention to. A popular method tailored to drawing directed graphs is the so-called *Sugiyama method*. It was introduced in [50] and was followed up by an extended survey [18] that discusses many ground-breaking ideas still being used today. The Sugiyama method is based on the observation that directed graphs often have a general *direction* or *flow*. Its fundamental idea is that drawings of such graphs should highlight this flow. The building blocks of this method, as proposed in [18 and 50], have inspired an entire family of specialized graph drawing algorithms that differ from generic methods substantially in that they arrange nodes in horizontal or vertical layers so that the majority of edges in the graph points into the same direction. The layouts resulting from these algorithms are called *layered drawings*.

A natural goal of any approach to drawing graphs is to improve the layout quality by avoiding artifacts that hinder viewers comprehending the information represented by the graph. Therefore, algorithms inspired by the Sugiyama method aim at satisfying basic aesthetic criteria established in [18 and 50] such as minimization of edge crossings and creation of short, straight and uniform edges that place neighbors close to each other. These objectives are supported by empirical studies such as the ones performed by Purchase et al. [43 and 46], which have identified edge crossings and bends to be particularly distracting.

The layered graph drawing method proposed by Sugiyama et al. is inherently modular. Despite using inconsistent terminology, most algorithms inspired by this method implement the following steps in the same order.

1 Cycle removal
2 Layer assignment (sometimes called node ranking)
3 Crossing minimization (also referred to as node ordering)
4 Node positioning (or coordinate assignment)
5 Edge routing

All of these steps have a well-defined input and output format, allowing them to be implemented and replaced independently. Despite 30 years of research efforts having been invested into the design and analysis of algorithms for producing layered drawings, no perfect all-in-one solution has been developed yet. Since each of the above steps has a strong influence on the resulting layouts, it is well worth experimenting with different ideas for each of them, in particular so as even the most promising heuristics fail to always generate optimal layouts. For instance, a cycle removal algorithm sounds perfectly reasonable if it reverses a provable minimum amount

of edges. If poorly designed, however, such an algorithm might ignore the overall structure and flow of the graph and therefore generate odd results. Making experimentation possible allows for an analysis of how well the various heuristics work together.

The rest of this section is a survey of graph drawing literature related to layered drawings of directed graphs. It gives an overview of existing methods developed by researchers for each step of the Sugiyama framework and compares their structure and performance. Notes on their implementation and usage in Ti*k*Z are provided in sections 4.2 and 4.3.

4.1.1 Algorithms for Cycle Removal

This section describes some of the heuristic solutions to the problem of making a directed graph acyclic while changing the direction of as few edges as possible. A more complete survey can be found in [3]. This problem is closely related to the *maximum acyclic subgraph problem* where the aim is to find a maximum set $E_a \subset E$ for a graph $G = (V,E)$ such that the subgraph $G_a = (V,E_a)$ contains no cycles. An alternative approach to describing the same thing is via the *minimum feedback arc set problem*. Here, the goal is to find a minimum set $E_f \subset E$ of edges such that the remaining graph $G_{\setminus f} = (V,E \setminus E_f)$ is acyclic. Both of these problems have been among the first problems proven to be NP-hard [25 and 33].

It is easy to prove that reversing the edges in E_f instead of removing them also yields a graph without cycles. This is important because the presence of these edge has a strong impact on the final drawing. By simply reversing said edges, the original adjacency information is preserved and the edges along with their weights remain part of the graph. The purpose of all heuristics presented here is to find an approximative solution to the *minimum feedback arc set problem* efficiently that reverses as few edges as possible.

A Simple Greedy Algorithm

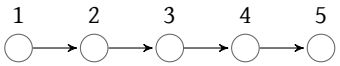
Berger and Shor present a linear-time greedy algorithm for cycle removal based on a simple observation: if for every node either its incoming or its outgoing edges are added to E_a and all its edges are removed from the graph, the resulting subgraph G_a turns out to be acyclic [6]. In an analogous manner, adding either the incoming or outgoing edges to E_f results in an acyclic subgraph $G_{\setminus f}$.

Their GREEDY-CYCLE-REMOVAL algorithm processes the nodes of the input graph one by one in the order of their creation. For every node it checks whether there are more remaining outgoing

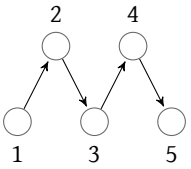
than incoming edges. It reverses the type of which there are fewer edges and thereby guarantees to reverse no more than half of the edges overall. The algorithm runs in linear time as it examines each node and edge only once.

The upper bound it guarantees for the size of E_f is neither hard to see nor is it very useful. Any cycle removal algorithm reversing more than half of the edges can be transformed to have the same upper bound—or a better one even—simply by reversing the E_a instead of E_f. The algorithm can be shown to perform poorly on directed acyclic graphs. Since DAGs are acyclic by definition, no edges would need to be reversed at all. Yet, if the nodes are specified or stored in an unfortunate order, the simple greedy algorithm ends up reversing a significant amount of the edges, resulting in highly unpleasant zig-zag drawings. This is demonstrated in figure 4.1.

■ **Figure 4.1** Illustration of the zig-zag effect caused by applying th GREEDY-CYCLE-REMOVAL algorithm to a graph specified in an unfortunate order.



Order of nodes: 1, 2, 3, 4, 5.
No edges are reversed



Order of nodes: 2, 4, 1, 3, 5.
Edges (1,2) and (3,4) are reversed.

A Greedy Heuristic Prioritizing Sources and Sinks

The simple greedy algorithm processes nodes in the order they are stored with in the set $V(G)$. If $V(G)$ is represented as a flat array or a vector, they are examined in the order of their definition. While this order may somehow reflect the structure of the input graph, paying special attention to less obvious properties instead can improve the results. This may not only avoid the zig-zag effect but also also helps in further reducing the guaranteed maximum size of E_f.

One useful property is that directed graphs often have a number of sources and sinks, that is, nodes with no incoming or outgoing edges. Obviously, edges incident to these nodes cannot be part of cycles and thus never need to be reversed. In [16], Eades et al. propose a modification of the simple greedy algorithm that prioritizes sources, sinks, and nodes with a maximum difference of out- and indegree over other nodes. This algorithm is shown in listing 4.1.

The enhanced greedy algorithm has been proven to always generate an feedback arc set with a size of at most

■ **Listing 4.1** Enhanced greedy algorithm that prioritizes sources, sinks, and nodes with a maximum difference of out- and indegree.

```
PRIORITIZING-GREEDY-CYCLE-REMOVAL(G)
1   E_a ← ∅
2   while V(G) is not empty do
3       while G contains a sink v do
4           add INCOMING-EDGES(G, v) to E_a
5           remove INCOMING-EDGES(G, v) from E(G)
6
7       delete isolated nodes from V(G)
8
9       while G contains a source v do
10          add OUTGOING-EDGES(G, v) to E_a
11          remove OUTGOING-EDGES(G, v) from E(G)
12
13      if V(G) is not empty then
14          v ← node with maximum value
15              OUTGOING-EDGES(G, v).size -
16              INCOMING-EDGES(G, v).size
17
18          add OUTGOING-EDGES(G, v) to E_a
19
20          remove INCOMING-EDGES(G, v) from E(G)
21          remove OUTGOING-EDGES(G, v) from E(G)
22
23          remove v from V(G)
24
25  for each edge e ∈ E(G)\ E_a do
26      reverse edge e
```

$$|E_f| \leq \frac{|E(G)|}{2} - \frac{|V(G)|}{6},$$

assuming that the graph contains no cycles involving only a single pair of nodes [3]. As multiple edges are removed prior to the cycle removal step in algorithms based on the Sugiyama method, we can safely expect this assumption to hold. Obviously, this upper bound for the number of reversed edges is a slight improvement over the previous method. The runtime of PRIORITIZING-GREEDY-CYCLE-REMOVAL is once again linear in the size of the input graph [16].

What is more interesting though is that by filtering out sources and sinks in each iteration prior to reversing any edge, the algorithm generates an optimal feedback arc set for directed acyclic graphs. The only situation where edges are marked to be reversed is in lines 13–23, which is if no

sources, sinks, and isolated nodes are left in the graph. If the input graph is acyclic, all nodes are removed iteratively in lines 3–5. This, in turn, implies that the condition in line 13 is never met and thus no edges are reversed. It is also not hard to see that for any graph containing only a single cycle, only one edge is reversed to break the cycle. As a result, the PRIORITIZING-GREEDY-CYCLE-REMOVAL algorithm is the ideal choice for directed acyclic graphs and graphs with only a single cycle.

A Randomized Algorithm

An even better overall performance guarantee has been shown for a randomized algorithm based on GREEDY-CYCLE-REMOVAL. This algorithm computes a random ordering of the nodes before processing them in exactly the same way as the simple greedy algorithm [6]. Its provable upper bound for the size of E_f is

$$|E_f| \leq \left(\frac{1}{2} - \Omega\left(\frac{1}{\sqrt{\Delta(G)}}\right)\right)|E(G)|,$$

where $\Delta(G)$ denotes the maximum degree of any node in $V(G)$ and Ω describes a lower bound up to a constant factor. For the sake of completeness, the randomized linear-time algorithm is included in listing 4.2. Despite the improved maximum size of E_f, this method suffers from the same disadvantages as the simple greedy algorithm.

■ **Listing 4.2** A randomized greedy algorithm with an improved upper bound for the size of E_f.

```
RANDOMIZED-CYCLE-REMOVAL(G)
1   V(G) ← SHUFFLE(V(G))
2   GREEDY-CYCLE-REMOVAL(G)
```

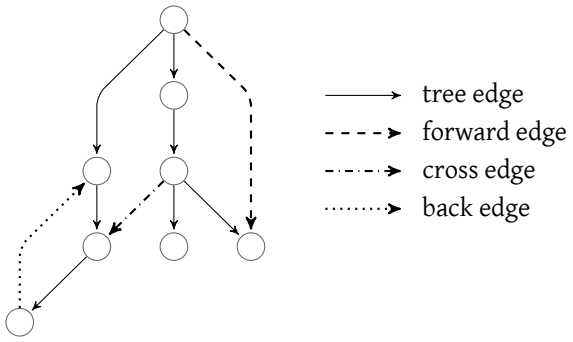
An Algorithm Based on Edge Classifications

In Ti*k*Z, graphs are usually created by humans manually. This allows to make assumptions about the input graph that would otherwise not be possible. For instance, it seems reasonable to assume that the order in which nodes and edges are entered by the user somehow reflects the natural flow the user has had in mind for the graph.

In order to preserve the natural flow of the input graph, Gansner et al. propose to remove cycles by performing a series of depth-first searches starting at individual nodes in the order they appear in the graph [27] (see listing 4.3). This algorithm implicitly constructs a spanning tree of the nodes reached during the searches. It thereby partitions the edges of the graph into tree

edges and non-tree edges. The non-tree edges are further subdivided into forward edges, cross edges, and back edges, as is illustrated in figure 4.2. Forward edges point from a tree nodes to one of their descendants. Cross edges connect unrelated branches in the search tree. Back edges connect descendants to one of their ancestors.

■ **Figure 4.2** The different types of edges in a depth-first search.



It is not hard to see that reversing back edges will not only introduce no new cycles but will also make any directed graph acyclic. Gansner et al. argue that this approach is more stable than others in that fewer inappropriate edges are reversed compared to other methods [27], despite the lack of a provable upper bound for the number of reversed edges [3]. With a linear runtime the algorithm is comparable to the other solutions discussed before. According to [27], the resulting drawings are more informative than those obtained by alternative methods such as merging all nodes of a cycle into a supernode or duplicating one of the nodes in each cycle as was suggested by others [7, 18 and 48].

4.1.2 Algorithms for Layer Assignment

Algorithms for producing layered drawings place nodes on discrete layers from top to bottom. These layers can be formalized as a partition $L = \{L_1, \ldots, L_h\}$ of the nodes with $\bigcup_{i=1}^{h} L_i = V(G)$. Let $layer : V(G) \to \{1, \ldots, h\}$ with $layer(v) = i$ iff $v \in L_i$ be the characteristic or *layer assignment* function of this partition.

Layer assignment is the problem of finding a partition L so that for all edges $e = (u,v) \in E(G)$ the equation $layer(u) < layer(v)$ holds. Such a partition is called a *layering*. This definition can be extended by introducing edge weights or priorities and minimum length constraints which has practical applications and allows users to fine-tune the results. Edge weights are defined by a function $w : E(G) \to \mathbb{N}$. The minimum length of an edge $e = (u,v)$ is the minimum allowed layer difference $layer(v) - layer(u)$ of its nodes, denoted by $minlength(e)$. The actual length $layer(v) -$

■ **Listing 4.3** Cycle removal algorithm based on reversing back edges of a depth-first search.

```
DFS-CYCLE-REMOVAL(G)
1   for each node v ∈ V(G) in the order of creation do
2       v.visited = false
3       v.stacked = false
4
5   for each node v ∈ V(G) in the order of creation do
6       DFS-VISIT(G, v)

DFS-VISIT(G, v)
1   if not v.visited then
2       v.visited = true
3       v.stacked = true
4
5       for each edge e ∈ OUTGOING-EDGES(G, v) in the order of creation do
6           u = GET-NEIGHBOR(G, e, v)
7
8           if u.stacked then
9               e.reversed = true // e is a back edge
10          else
11              DFS-VISIT(G, u)
12
13      v.stacked = false
```

layer(u) of an edge $e = (u,v)$ is often abbreviated as *length*(e). This modification—which we will refer to as *length-constrained layer assignment*—describes the problem of finding a partition L so that for all edges $e = (u,v)$ the equation $layer(v) - layer(u) \geq minlength(e)$ is true. In contrast to the regular layer assignment problem, length-constrained layer assignments allow the nodes u and v of an edge $e = (u,v)$ with $minlength(e) = 0$ to be placed on the same layer.

A layering L resulting from length-constrained layer assignment highlights the general flow of an acyclic input graphs by pointing all its edges into the same direction. Another aesthetic criterion to be satisfied by the layer assignment step is locality of information, that is, placing neighbors at short distance. Short edges are not only desirable in general, they also have an impact on the runtime of later steps like crossing minimization and node positioning. The *weighted length-constrained layer assignment* problem allows individual edges to be prioritized when it comes to deciding which ones to make short and which to make long. It can be described with the following linear optimization program:

$$\begin{aligned} &\text{minimize} && \sum_{(u,v)\in E(G)} weight(u,v)\cdot(layer(v)-layer(u)) && (4.1)\\ &\text{subject to} && layer(v)-layer(u)\geq minlength(u,v) && \text{for all } (u,v)\in E(G)\\ &\text{and} && layer(v)\in\mathbb{N} && \text{for all } v\in V(G). \end{aligned}$$

Throughout graph drawing literature there is a mix of algorithms that work on this linear program and algorithms that follow different ideas. Most of these require a directed graph without multiple edges, loops, and cycles as their input. Removing multiple edges and loops is usually implemented as a pre-processing step (see section 4.2). Strategies to eliminate cycles are covered in the previous section. Thus, we can safely assume receiving such simple input graphs. In the following, three approaches to solving the layer assignment problem are discussed. A more complete survey can be found in [3 and 18].

Proper Layerings

Algorithms for the subsequent node positioning step (section 4.1.4) requires a *proper layering* to be generated beforehand. A layering is called *proper* if every edge $e \in E(G)$ has a length of 1. This means that any pair of neighbors needs to be placed on adjacent layers. If this requirement is not satisfied by the layer assignment step already, a proper layering can easily be obtained by inserting so-called *dummy nodes* along edges $e = (u,v)$ with a $length(e) > 1$. In the proper layering, edges $e = (u,v)$ with a length k are replaced by a path $(u = v_1,v_2,\ldots,v_k = v)$ of length k. The dummy nodes $v_2,\ldots,v_{k-1}$ are placed in the layers between u and v, as is illustrated in figure 4.3.

It is unclear how to distribute the weights of the original edges (u,v) to the edges (v_i,v_{i+1}). Dividing the original weight by the number of dummy edges and assigning the result to the individual edges (v_i,v_{i+1}) is only one possible strategy. The Ti*k*Z implementation simply assigns the original weight to all edges of the path. This is supposed to give long edges with many internal dummy edges a higher priority when deciding which edges to straighten in the node positioning step. The results do not indicate other weight choices to be necessary.

Layerings for a Given Width

The height of a layering is defined as the number h of layers. Its width is the maximum number of nodes on any of these layers. A natural approach is to minimize the height or width while the other dimension is fixed or bound by an upper limit. The *Coffman-Graham algorithm* [8] which originates from job scheduling takes as input a number w and a graph without transitive edges.

■ **Figure 4.3** A regular layering (i) and its proper version (ii) with dummy nodes v_2,v_3 (painted as gray dots) inserted along the edge (u,v).

(i) (ii)

It generates a layered drawing of width at most w while trying to minimize the height. This method is able to compute an optimal minimum height layering for $w \leq 2$. Overall, the resulting height h can be shown to be bounded by

$$h \leq \left(2 - \frac{2}{w}\right) h_{opt} \tag{4.2}$$

relative to the optimal height h_{opt}.

The pseudocode for an adaption of this algorithm for graph drawing can be found in [3].

Layerings with a Minimum Height

Another technique originally developed for use in job scheduling is the *longest-path algorithm*. It aims at minimizing the height of the layering while not giving any upper bound guarantee for the resulting width per se. As is shown in listing 4.4, the algorithm works by traversing the nodes of the input graph in a topological order, assigning to each node its minimum allowed layer depending on its predecessors in the graph topology. The resulting layering is often called a *longest-path layering* as the height is equal to the length of the longest path between any two nodes in the graph, at least if $minlength(e)$ is set to 1 for all edges $e \in E(G)$. In addition to always generating layerings with a minimum height and being very simple, another advantage of this method is its linear runtime [40]. It has however been reported to generate many dummy nodes and occasionally result in an unfavorable density of edges [42]. A topological order can be computed in linear time by iteratively visiting of the graph and removing them immediately afterwards [32]. Lines 3-8 of the algorithm shown in listing 4.4 examine each node and edge exactly once, resulting in an overall linear runtime of the algorithm.

■ **Listing 4.4** The longest-path algorithm applied to the layer assignment problem.

```
LONGEST-PATH-LAYER-ASSIGNMENT(G)
1   V ← TOPOLOGICAL-ORDERING(G)
2
3   for each node v ∈ V do
4       v.layer = 1
5
6       for each edge e ∈ INCOMING-EDGES(G, v) do
7           u = GET-NEIGHBOR(G, e, v)
8           v.layer = MAX(v.layer, u.layer + e.minlength)
```

A Solution Based on the Network Simplex Method

As mentioned before, the generated layering is transformed into a proper layering prior the node positioning step. This typically results in a larger graph due to the insertion of dummy nodes and the corresponding edges. As many researches have noted [3, 18 and 27] it is critical to keep the number of dummy nodes low. Not only does the potential quadratic increase in size make subsequent steps more expensive, dummy nodes are also the only cause for edge bends in the final drawing (see section 4.1.5). Although there are various ways to straighten edges, more dummy nodes usually means more bends. Finally, the need to add dummy nodes reflects the presence of long edges which in turn make drawings harder to understand and should generally be avoided.

Solving the linear program for the weighted length-constrained layer assignment problem (see formula 4.1) has been proven to minimize the total length of the edges and thus the number of dummy nodes that need to be added [22]. Furthermore, Eades and Sugiyama state that minimizing the number of dummy nodes yields a drawing with minimum overall height [18]. The minimum possible height is limited by the longest path from any node in the graph to all others which in turn depends on how well the cycle removal step performs. Gansner et al. propose that the linear program be solved by transforming the layer assignment problem into an equivalent minimum-cost flow problem or by applying the simplex algorithm [27]. For their graph drawing tool Graphviz they chose to implement a solution based on the network simplex method which is a specialized version of the simplex algorithm known from linear programming. More precisely, it is a variation of the bounded-variable primal simplex method specifically designed for solving the minimum-cost flow problem [10, 28 and 38].

A few additional definitions introduced in [10 and 27] are helpful in describing and understanding the concepts behind this method. Please note that throughout the work of Gansner et al.,

the terms *rank*, *ranking*, and *rank assignment* are used instead of *layer*, *layering*, and *layer assignment*. A *feasible layering* is a layering that satisfies the minimum length constraint, that is $length(e) \geq minlength(e)$ holds for all $e \in E(G)$. The *slack* of an edge is defined as the difference of its length and its minimum length. This in turn means that a layering is feasible if and only if all edges of the input graph have a non-negative slack. Edges are further called *tight* if they have zero slack.

The network simplex method is built around the idea of constructing and modifying an undirected spanning tree of the input graph. Such a spanning tree always corresponds to a layering of the graph. This layering can be constructed by picking an initial node of the spanning tree and assigning it a layer. The other nodes are then layered iteratively by assigning them the layer of an already ranked node adjacent in the original directed graph, incremented or decremented by the minimum length of the connecting edge, depending on whether the current node is the head or tail node of this edge. Note that this layering is not unique as a different initial node and an alternative order of nodes in the iterative layer assignment may yield other valid layerings. A spanning tree that allows a feasible layering is called a *feasible spanning tree*. Due to the way feasible spanning trees are constructed all their edges are tight.

Given a directed input graph and a corresponding feasible spanning tree, a *cut value* can be defined for each tree edge. Assuming that a tree edge is deleted, the spanning tree will break into two partial spanning trees, one of which is called the *tail component* while the other is called the *head component*. The classification of either component as head or tail is determined by the direction of the original directed edge corresponding to the deleted tree edge. The cut value of this tree edge is defined as the weighted sum of directed edges from the tail to the head component minus the weighted sum of edges pointing into the other direction.

These definitions allow for a relatively straight-forward description of the network simplex method for layer assignment. Given an arbitrary feasible layering generated at the beginning of the algorithm, an initial feasible spanning tree of tight edges is constructed. In order to minimize the weighted sum of edge lengths (see formula 4.1), the network simplex method iteratively replaces tree edges with negative cut values by appropriate non-tree edges with minimum slack. Degeneracy problems put aside, a negative cut value typically indicates that the total weighted sum of edge lengths can be reduced by lengthening the tree edge. By making this tree edge as long as possible, one of the head to tail component edges eventually becomes tight. Exchanging these two edges in the tree and updating the cut values and layers of all nodes yields a new feasible spanning tree. This procedure is repeated until no further improvement is possible, that is, all edges of the spanning tree have non-negative cut values. The complete algorithm is shown in listing 4.5. In addition to minimizing the total weighted sum of edge lengths, it also normalizes layers so that they are in the range $1,2,\ldots,h$. Lastly, in a final balancing step

it attempts to distribute nodes evenly on these layers while of course preserving the minimum length constraint of the edges.

■ **Listing 4.5** The network simplex layer assignment algorithm, as proposed in [27].

```
NETWORK-SIMPLEX-LAYER-ASSIGNMENT(G)
1   T ← INITIAL-FEASIBLE-TREE(G)
2
3   while T contains an edge with negative cut value do
4       e ← GET-EDGE-WITH-NEGATIVE-CUT-VALUE(G, T)
5       f ← FIND-TIGHT-NON-TREE-EDGE(G, T, e)
6       REPLACE-TREE-EDGE(G, T, e, f)
7
8   NORMALIZE-LAYERS(G)
9   BALANCE-LAYERS(G)
```

The Graphviz implementation presented in [27] includes a number of optimizations that make updating the cut values more efficient. Gansner et al. also propose a more efficient technique for checking whether a node lies in the head or tail component of a tree edge. These optimizations are crucial as the same algorithm is later applied to the node positioning problem that is explained in section 4.1.4.

4.1.3 Algorithms for Crossing Minimization

The number of edge crossings in a layered drawing is determined by the ordering of nodes at each of its layers. Therefore, crossing minimization is the problem of reordering the nodes at each layer so that the overall number of edge crossings is minimized. The crossing minimization step takes a proper layering where every edge connects nodes in neighbored layers, allowing algorithms to minimize crossings layer by layer rather than all at once. While this does not reduce the complexity of the problem, it does make it considerably easier to understand and implement. Techniques based on such an iterative approach are also known as *layer-by-layer sweep* methods. They are used in many popular heuristics due to their simplicity and the good results they produce.

Sweeping refers to moving up and down from one layer to the next, reducing crossings along the way. In layer-by-layer sweep methods, an initial node ordering for one of the layers is computed first. Depending on the sweep direction this can either be the first layer or the last; in rare occasions the layer in the middle is used instead. Followed by this, the actual layer-by-layer sweep is performed. Given an initial ordering for the first layer L_1, a downward sweep first holds the nodes in L_1 fixed while reordering the nodes in the second layer L_2 to reduce the number

of crossings between L_1 and L_2. It then goes on to reorder the third layer while holding the second layer fixed. This is continued until all layers except for the first one have been examined. Upward sweeping and sweeping from the middle work analogous.

Obviously, the central aspect of the layer-by-layer sweep is how the nodes of a specific layer are reordered using a neighbored layer as a fixed reference. This problem is known as *one-sided crossing minimization*, which unfortunately is NP-hard [19, 20 and 26]. In the following various heuristics to solve this problem are presented. A formalization of the crossing minimization problem and an extensive survey of the heuristics available are once more given in [3 and 18].

The Barycenter and Median Heuristics

The *barycenter* and *median* heuristics are based on the assumption that, in order to reduce edge crossings, nodes should be positioned close to their neighbors. Both methods are simple, relatively fast and produce pleasing results.

Let L_i be the layer that is reordered and L_{i-1} the layer held fixed. The barycenter method first assigns to each node $v \in L_i$ the barycenter (average) of the positions of its neighbors in L_{i-1}, defined as $N_{i-1}(v) = \{u \mid (u,v) \in E(V)\}$:

$$barycenter(v) = \frac{1}{|N_{i-1}(v)|} \sum_{u \in N_{i-1}(v)} pos_{i-1}(u), \tag{4.3}$$

where $pos_{i-1}(u)$ denotes the position of u in the layer L_{i-1}. Nodes $v,w \in L_i$ that are assigned the same value $barycenter(v) = barycenter(w)$ need to be moved apart by a small amount. In a second step, the nodes of L_i are sorted according to their barycenter values.

An interesting property of the barycenter heuristic is that it generates an ordering with no crossings if one is possible at all [9]. Computing the barycenter values requires linear time. If the barycenter values and layer positions are restricted to be integers, sorting can be implemented in $O(n)$ as well using integer sorting algorithms such as bucket sort or radix sort. This results in a linear runtime for the overall layer-by-layer sweep.

The median method assigns to each node $v \in L_i$ the median of the x-coordinates of its neighbors $N_{i-1}(v)$. Let $u_1,u_2,\ldots,u_j$ be the neighbors of v with $pos_{i-1}(u_1) < pos_{i-1}(u_2) < \ldots < pos_{i-1}(u_j)$. In this case the median is defined as

$$median(v) = \begin{cases} pos_{i-1}(u_{\lceil \frac{j}{2} \rceil}) & \text{if } j \text{ is odd and } j > 1, \\ pos_{i-1}(u_{\frac{j}{2}-1}) & \text{if } j \text{ is even, or} \\ 1 & \text{if } j = 0. \end{cases} \tag{4.4}$$

Like with the barycenter method, the nodes of L_i are sorted by their median values afterwards. It can be shown that, for the one-sided crossing minimization problem, the median method always generates orderings with at most three times as many edge crossings than in the minimum solution. An interesting statement in this context comes from [18], where it is claimed that there is no fast method known to produce drawings with an overall number of crossings within a constant factor of the minimum. Like the barycenter heuristic, the median approach can be implemented to run in linear time.

The Greedy Switch Heuristic

Unlike the barycenter and the median heuristics the greedy switch method explicitly counts the number of crossings between two layers. This makes it more expensive as all pairs of edges between the two layers need to be checked with regard to whether they cross or not.

A pair (l,u) and (r,v) of edges with $u,v \in L_i$ and $l,r \in L_{i-1}$ crosses if the tail and head nodes are not in the same order, that is if

$$\text{sign}(pos_i(v) - pos_i(u)) \quad \neq \quad \text{sign}(pos_{i-1}(r) - pos_{i-1}(l)). \tag{4.5}$$

The *crossing number* c_{uv} of two consecutive nodes $u,v \in L_i$ with $pos_i(u) < pos_i(v)$ is defined as the number of crossings between edges incident to u and edges incident to v. Switching the positions of u and v changes the total number of crossings in the drawing by $c_{vu} - c_{uv}$. This observation is the basis for the main idea behind greedy switching.

At each layer to be reordered to minimize crossings, the greedy switch algorithm iterates over all consecutive pairs u,v of nodes. Whenever it comes across a pair for which $c_{vu} < c_{uv}$, it switches the positions of the two nodes. This is repeated until no further improvement is possible.

The crossing numbers can be precomputed in linear time according to [49]. Thus, the overall complexity of this method is $O(|L_i|^2)$ for each layer L_i [3]. As others have noted [27], greedy switching is particularly useful as a post-processing step in combination with other heuristics due to its nature of making changes only when improvements are possible.

A Combination and Refinement of the Individual Strategies

Gansner et al. combine an initial ordering based on a depth-first search with the median and greedy switch heuristics applied in the form of an alternating layer-by-layer sweep based on a weighted median [27]. The basic structure of their algorithm is shown in listing 4.6.

■ **Listing 4.6** Crossing minimization using a combination of the median and greedy switch layer-by-layer sweeping heuristics.

```
COMBINED-CROSSING-MINIMIZATION(G)
1   ordering ← COMPUTE-INITIAL-DFS-ORDERING(G)
2   best ← ordering
3
4   for i ← 0, ..., iterations do
5       if i is even then
6           ordering ← WEIGHTED-MEDIAN-SWEEP(G, ordering, 'up')
7       else
8           ordering ← WEIGHTED-MEDIAN-SWEEP(G, ordering, 'down')
9
10      ordering ← GREEDY-SWITCH-SWEEP(G, ordering, 'down')
11
12      if COUNT-CROSSINGS(G, ordering) < COUNT-CROSSINGS(G, best) then
13          best ← ordering
14
15  return best
```

The initial ordering is constructed while traversing the directed acyclic graph using a depth-first search, starting at its source nodes. Nodes are placed from left to right on layers that correspond to levels in the depth-first search tree. This means that the sources are moved to the first layer, their immediate children in the search tree are moved to the second layer and so forth.

The weighted median sweep is a refined version of the regular median heuristic. Compared to the original method it only behaves differently in situations where the number of neighbors on the fixed layer is even. If there are only two neighbors on the fixed layer, the arithmetic mean of their position is used as the median value. If the number of neighbors is even and there are more than two neighbors, a weighted median is applied that moves nodes to the side where their neighbors are more closely packed [27].

4.1.4 Algorithms for Node Positioning and Horizontal Balancing

The second last step of the Sugiyama method decides about the final x- and y-coordinates of the nodes. The main objectives of this step are to position nodes so that the number of edge bends is kept small and edges are drawn as vertically as possible. Another goal is to avoid node and edge overlaps which is crucial in particular if the nodes are allowed to have non-uniform sizes.

The y-coordinates of the nodes have no influence on the number of bends. Obviously, nodes need to be separated enough geometrically so that they do not overlap. It feels natural to aim

at separating all layers in the drawing by the same amount, which we will refer to as the *level distance* in the following, denoted by $leveldist(G)$. Large nodes, however, may force node positioning algorithms to override this uniform level distance in order to avoid overlaps. Friedrich and Schreiber discuss ways to solve this problem in more detail [23]. The ideas presented in this section assume uniform node heights. This allows to associate each layer with a specific y-coordinate and compute y-coordinates of all nodes in a straight-forward way using the following formula:

$$y(v) = layer(v) \cdot leveldist(G), \quad v \in V(G). \tag{4.6}$$

The main problem to solve in this step is to find x-coordinates that are in accordance with the objectives described above. Most solutions found in graph drawing literature are based on the methods developed for previous steps by reusing techniques from layer assignment and crossing minimization in particular. In the following, two of these solutions are discussed. Additional ideas are presented in [3, 18 and 27].

Algorithm Based on the Median Heuristic

Bastert and Matuszewski [3] and Gansner et al. [27] describe an algorithm based on the median heuristic used in crossing minimization. The basic idea is to first compute initial x-coordinates for all nodes, for instance by positioning them from left to right in the order generated during crossing minimization, followed by several iterations of improving this layout. This iterative improvement involves three phases: positioning, straightening, and packing. Positioning applies such techniques as the median heuristic and variations of it. As this may result in many edge bends, the next phase attempts to straighten the edges again, usually by trying to assign all nodes along a path of dummy nodes and edges the same x-coordinate. These two steps may increase the width of the drawing. Hence, a packing algorithm is applied afterwards to compress the drawing in horizontal direction without introducing new edge bends.

This iterative approach is stated to generate good layouts in little time on one hand but is also described as difficult to implement on the other hand [27].

Exact Algorithm Based on the Network Simplex Method

The problem of finding x-coordinates for nodes so that edges are drawn as straight and vertical as possible can also be defined as an integer optimization problem, as is proposed in [27]:

$$\begin{aligned} &\text{minimize} && \sum_{e=(u,v)\in E(G)} priority(e) \cdot weight(e) \cdot |x(v) - x(u)| \\ &\text{subject to} && x(b) - x(a) \geq minxdist(a,b), \end{aligned} \tag{4.7}$$

where a is the left neighbor of b in the same layer and $minxdist(a,b)$ is the minimum horizontal distance allowed between the centers of a and b. Given a minimum allowed node separation $siblingdist(G)$ between any two nodes in the same layer, the minimum distance between two specific nodes a and b is defined as

$$minxdist(a,b) = \frac{a.width + b.width}{2} + siblingdist(G). \tag{4.8}$$

The function $priority(e)$ is used to straighten long edges with higher priority than short edges. As edges between original nodes in adjacent layers can always be drawn straight, the priority needs to be higher for edges incident to dummy nodes. It is thus defined as

$$priority(u,v) = \begin{cases} 1 & \text{if } u \text{ and } v \text{ are original nodes,} \\ 2 & \text{if one of } u,v \text{ is a dummy node, or} \\ 8 & \text{if } u,v \text{ are both dummy nodes.} \end{cases} \tag{4.9}$$

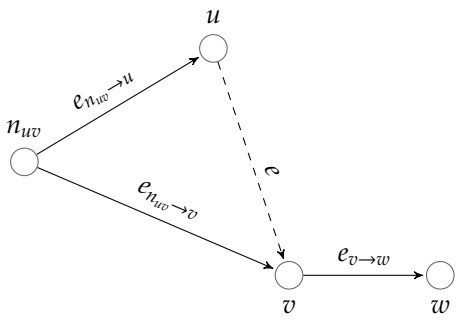
This optimization problem can be transformed into linear program and be solved with the simplex method. Unfortunately, this increases the size of the simplex matrix quadratically, making it unreasonable for larger graphs [27].

In [27], Gansner et al. therefore propose the application of the network simplex method to an auxiliary graph G' that is combinatorially analog to the mathematical transformation into a linear program. The graph G' has the same nodes as the input graph G plus a new node n_{uv} for every edge $e = (u,v) \in E(G)$. Two types of edges are present in G'. The first are used to separate nodes in the same layer. If u is the left neighbor of v in a layer, a new edge $e_{u\to v}$ is created with $minlength(e_{u\to v}) = minxdist(a,b)$ and $weight(e_{u\to v}) = 0$. The second type of edges is used to straighten the original edges of G as much as possible. For each original edge $e = (u,v) \in E(G)$, there are two new edges $e_{n_{uv}\to u} = (n_{uv},u)$ and $e_{n_{uv}\to v} = (n_{uv},v)$. Each of these edges are set to have a minimum length of 0 and a weight of $weight(e)\cdot priority(e)$. This construction is illustrated in figure 4.4. Gansner et al. show that applying the layer assignment problem to the auxiliary graph yields a layering that directly corresponds to an optimal solution for the horizontal node positioning problem.

4.1.5 Algorithms for Edge Routing

The original layered drawing method described by Eades and Sugiyama in [18] does not include the routing or shaping of edges as a main step. This makes sense if all nodes have the same size

■ **Figure 4.4** Construction of the auxiliary graph for computing x-coordinates with the network simplex method.

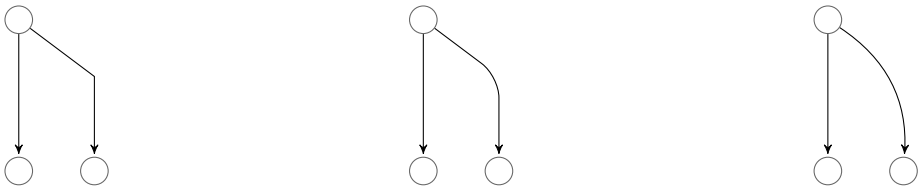


and shape. In practical scenarios, however, this assumption often does not hold. In these cases, advanced techniques may have to be applied in order to avoid overlaps of nodes and edges.

Due to time constraints, this problem was not investigated in this thesis. I felt that a simple comparison of the different drawing styles for bent edges would suffice, given that supporting non-uniformly sized nodes would require further adjustments to the other steps as well.

Three basic edge drawing styles are shown in figure 4.5. The traditional way to draw bent edges is to draw them as series of connected line segments called *polygonal chains* or *polylines*. Polyline drawings contain only straight lines and are thus easy to create at low computational cost. They may, however, emphasize sharp corners too much and result in busy drawings of larger graphs. An alternative to polylines are polylines with rounded corners. Their visual appearance is similar to that of polyline drawings and yet they may result in drawings that look less cluttered and more smooth in general. The radius of rounded corners can often be adjusted, which makes corner rendering more flexible and scalable.

■ **Figure 4.5** Different ways to draw bent edges.



Polyline drawing

Polyline drawing with rounded corners

Edges drawn as Bézier curves

Another way to draw bent edges is to replace the bend points with Bézier curves or splines. Bézier curves are specified by defining two end points and a number of control points (typically

two) that define the shape of the curve. Drawings generated with curves or splines have the advantage of looking very smooth but may look unbalanced if they are created independently for edges with different lengths.

4.2 Modular Implementation of the Framework in Ti*k*Z

The methods discussed in section 4.1 follow different approaches and all have their own characteristics and impacts on the final drawing of a directed graph. It is not always easy to see how solutions for the different steps interact and influence each other. Also, a particular solution may work well with one graph but may fail with another. Thus, in the context of this thesis, I chose to implement the layered drawing framework in the form of a modular layered drawing algorithm. Unlike other tools that ship hard-coded graph drawing algorithms, including Graphviz [27] and Mathematica [53], my solution provides a simple extension interface allowing interested researchers to add new implementations for each of the main steps of the Sugiyama method. Users are given the choice between different heuristics for each step by means of a straight-forward Ti*k*Z syntax that is explained in section 4.3.

4.2.1 Additional Steps and Outline of the Modular Algorithm

For the different steps to work with multiple edges and loops and in order to support features like user-defined node clusters, a number of intermediate steps are necessary in addition to cycle removal, layer assignment, crossing minimization, node positioning and edge routing.

Unlike multiple edges, loops—that is, edges pointing from a node to itself—have no influence on any of the algorithm steps except for the final pass where edges are routed and rendered. They can thus be removed from the graph prior to all other steps and only need to be restored before routing edges.

Multiple edges between different nodes are slightly more complicated to deal with. Depending on the heuristics selected, their weight may be used to decide which edges are made short and which are allowed to be longer. The weight is also used as a factor in deciding how much priority is given to making them appear as close to straight lines as possible. In the following, the terms weight and priority are used interchangeably. In order to avoid special handling of multiple edges in the implemented heuristics, the modular Ti*k*Z algorithm merges multiple edges into a single superedge whose weight is set to the sum of weights of the individual edges. Like loops, multiple edges are restored before routing edges.

Node clusters provide a way to place groups of nodes on the same layer or to align them at the same x-coordinate. Nodes in horizontal clusters need to be merged into cluster supernodes before the layer assignment step. The incoming and outgoing edges of these nodes need to be updated so that they leave or enter the corresponding supernodes instead. After layer assignment, the individual nodes need to be restored, adopting the relevant properties—the assigned layers in this case—from their supernodes. Creating supernodes may lead to the creation of new cycles, multi-edges, and loops, so care needs to be taken of removing these after merging clusters and restoring the original edges after expanding them again later. Vertical layers are also possible but were not investigated in this thesis.

■ **Listing 4.7** Complete outline of the modular layered drawing algorithm implemented in Ti*k*Z, including support for horizontal clusters, loops, and multiple edges. Vertical clusters are not supported in this algorithm yet.

```
GENERATE-LAYERED-DRAWING(G)
1   G_C ← G
2
3   merge horizontal clusters in G_C
4   remove loops in G_C
5   merge multiple edges in G_C
6
7   remove cycles in G_C
8   assign layers to nodes V(G_C) of G_C
9   apply layer information to the nodes V(G) of G
10  minimize edge crossings of G
11  position nodes of G
12  route edges of G
```

The final outline of the modular algorithm is shown in listing 4.7. Note that instead of merging edges and nodes in the original graph and expanding them again later, the graph is copied once and layers assigned to nodes in the copied version are applied back to the corresponding nodes in the original graph before minimizing edge crossings.

4.2.2 Algorithms Implemented for the Individual Steps

As mentioned before, the modular algorithm is more like an algorithm framework than an actual algorithm on its own. The methods used at every of the main steps listed in section 4.1 can be changed with a special Ti*k*Z option, as is explained section 4.3.5. Table 4.1 gives an overview of all heuristics implemented for each of the steps as part of the thesis.

Of the cycle removal algorithms discussed in section 4.1.1, the simple greedy algorithm (named `BergerS1990a`), the PRIORITIZING-GREEDY-CYCLE-REMOVAL algorithm (`EadesLS1990`), the RANDOMIZED-CYCLE-REMOVAL algorithm (`BergerS1990b`, based on the Lua pseudo-random number generator) and the DFS-CYCLE-REMOVAL algorithm (`GansnerKNV1993`) were implemented.

Table 4.1 Algorithms for the individual steps of the layered drawing framework.

cycle removal	layer assignment	crossing minimization	node positioning	edge routing
GansnerKNV1993	GansnerKNV1993	GansnerKNV1993	GansnerKNV1993	simple
BergerS1990a	longest path			
BergerS1990b				
EadesLS1990				

For layer assignment, the network simplex method (`GansnerKNV1993`) and the longest-path layering method (`longest path`) were implemented in Ti*k*Z.

The `GansnerKNV1993` algorithm is based on the reference implementation in Graphviz. It includes most of the optimizations proposed in [27] and produces similar layouts. The main notable difference is that nodes may be stored and visited in a different order, which can result in slightly different layerings or layer orderings. The network simplex method frequently traverses the graph using depth-first search algorithms, all of which are implemented recursively in Graphviz. I chose to implement most of these searches iteratively in order to avoid potential stack overflows with large graphs. Unfortunately, iterative DFS requires manual stack management and makes in- and post-order operations harder to implement. This was solved by adding a general-purpose `DepthFirstSearch` class that takes an initialization function used for defining the nodes to start with, a visit function to be called whenever a node is visited and a complete function, which is called post-order. Given that Lua's default recursion depth is 20000 on most machines, these efforts are less beneficial than they sound at first. In other situations, however, having an iterative DFS implementation that is flexible and easy to use may prove to be useful.

Considering that the `GansnerKNV1993` algorithm for layer assignment computes an optimal result, the `longest path` was merely added to have something to compare against. The version implemented in Ti*k*Z is based on the topological ordering algorithm by Kahn [32] and fills layers from top to bottom, meaning that the resulting drawings typically are more dense an the top as opposed to other implementations of the same algorithm where most nodes are located at the bottom of the drawing.

For crossing minimization and node positioning, only the methods presented by Gansner et al. in [27] were implemented. Their crossing minimization algorithm incorporates several of the other ideas, such as the layer-by-layer sweep, the median heuristic, and greedy switching. As a consequence, implementing these individual methods separately is not expected to provide better results. The `GansnerKNV1993` algorithm for node positioning reuses the network simplex code already implemented for the layer assignment algorithm with the same name. This made it simple to implement. All that had to be done was to construct the auxiliary graph described earlier. Since this algorithm yields optimal results fast, no other technique was investigated in the course of the thesis.

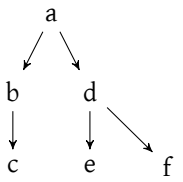
Edge routing was implemented in the most trivial way possible. In the `simple` algorithm shipped with TikZ, all dummy nodes along an edge are replaced by bend points, resulting in regular polyline drawings. Using the `/tikz/rounded corners` option for edges, such drawings can easily be transformed into polyline drawings with rounded corners.

4.3 TikZ Syntax and Options for Creating Layered Drawings

4.3.1 Syntax to Enable Layered Drawings

Like all types of drawing algorithms, algorithms for layered drawings are located in a dedicated TikZ library called `graphdrawing.layered`. This library needs to be loaded together with the `graphdrawing` library as is explained in section 2.3. Having loaded these libraries, the layered drawing of a TikZ graph can be obtained via the `/graph drawing/layered drawing` option that activates the default algorithm for layered drawings. As the time of writing, the modular algorithm presented in this thesis is the only algorithm available and is thus selected by default.

```
\tikz \graph [layered drawing] {
  a -> {
    b -> c,
    d -> { e, f },
  }
};
```



Alternatively, the name of the algorithm itself can be used instead of `layered drawing`.

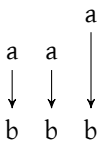
```
\tikz \graph [modular layered] { ... };
```

4.3.2 Separation of Layers and Nodes

Nodes and layers need to be separated by a certain distance along the x- and y-axis. By default, the horizontal and vertical distance is set to 1cm, which is a sane choice for graphs with relatively small nodes rendered on A4 or A5 paper. This covers many graphs found in literature and lecture notes about combinatorial mathematics and computer science.

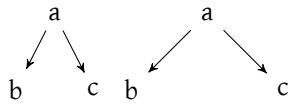
The vertical separation of the different layers of a drawing can be changed by setting the `/graph drawing/level distance` option to a mathematical expression that may include units.

```
\tikz \graph [layered drawing] { a -> b };
\tikz \graph [layered drawing,level distance=1cm] { a -> b };
\tikz \graph [layered drawing,level distance=0.5cm*3] { a -> b };
```



The horizontal separation between consecutive nodes on the same layer can be adjusted in a similar way by using the `/graph drawing/sibling distance` option.

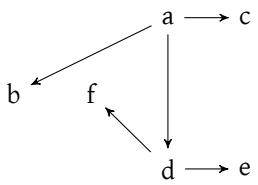
```
\tikz \graph [layered drawing]
  { a -> { b, c } };
\tikz \graph [layered drawing,sibling distance=2*1cm]
  { a -> { b, c } };
```



4.3.3 Support for Node Clusters

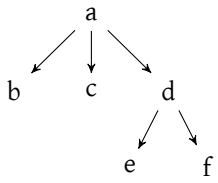
The modular layered drawing algorithm supports horizontal clusters of nodes that are positioned on the same layer. Clusters are defined by passing the `/tikz/graphs/new cluster` option to the `\graph` macro. This option takes a cluster name as its only argument and may be used multiple times to create more than one cluster. Nodes are added to a cluster by passing the cluster name to them as an option. In the following example, three clusters `first`, `second` and `third` are defined, forcing all nodes in each cluster to be placed on the same layer.

```
\tikz \graph [
  layered drawing,
  new cluster=first,
  new cluster=second,
  new cluster=third,
] {
  a [first] -> {
    b [second],
    c [first],
    d [third] -> { e [third], f [second] },
  }
};
```



Without node clusters the same graph is drawn as a tree with all edges pointing downwards.

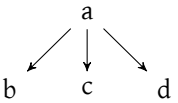
```
\tikz \graph [layered drawing] {
  a -> {
    b,
    c,
    d -> { e, f },
  }
};
```



4.3.4 Edge Parameters

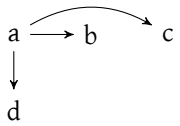
In adherence to the layer concept, edges are drawn pointing downwards with a minimum span of at least one layer. However, there are many scenarios where support for horizontal edges is important, e.g. when working with graphs of finite state machines. Also, forcing certain edges to have an increased minimum span can sometimes unravel drawings with undesired edge crossings. For this purpose, the modular layered drawing algorithm provides the `/graph drawing/layered drawing/minimum layers` edge option. This option takes an integer greater or equal to zero and can be used to adjust the minimum layer span of individual edges, as is demonstrated in the following examples. First, let us take a look at a very simple graph drawn with default parameters.

```
\tikz \graph [layered drawing]
  { a -> { b, c, d } };
```



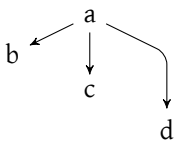
In the next example, the same graph is used, only this time two of its edges have zero minimum layer span. The algorithm is not capable of detecting overlapping edges between nodes in the same layer and thus, the ordering is not optimal. This requires one of the edges to be bent to the left. In the example this is done using the `/tikz/bend left` option that generates a simple Bézier curve.

```
\tikz \graph [layered drawing] {
  a ->[layered drawing={minimum levels=0}] b,
  a ->[layered drawing={minimum levels=0},bend left] c,
  a -> d,
};
```



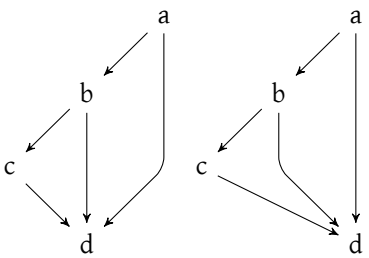
In the final example, the same graph is drawn with its three edges having a minimum layer span of 1, 2, and 3, respectively.

```
\tikz \graph [layered drawing,level distance=0.5cm] {
  a -> b,
  a ->[layered drawing={minimum levels=2}] c,
  a ->[layered drawing={minimum levels=3}] d,
};
```



Several of the main algorithm steps, that is, layer assignment, crossing minimization, and node positioning support weights or priorities to be attached to individual edges. The modular layered drawing algorithm has an edge option called `/graph drawing/layered drawing/weight` for this. It takes any integer or floating point number greater or equal to zero and is set to 1 by default. Edges with a higher weight are prioritized when it comes to making edges short and to straightening them. The following example demonstrates this effect by showing the drawing of a graph with and without prioritizing its longest edge (*a*,*d*).

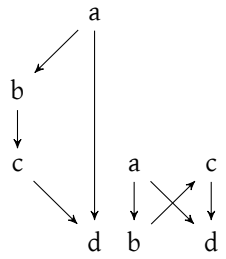
```
\tikz \graph [layered drawing] {
  a -> b -> c -> d,
  a -> d,
  b -> d,
};
\tikz \graph [layered drawing] {
  a -> b -> c -> d,
  a ->[layered drawing={weight=10}] d,
  b -> d,
};
```



4.3.5 Options for Changing the Individual Steps

As is explained in section 4.2.2, Ti*k*Z ships a number of algorithms for the different steps of the modular layered drawing algorithm By default, the `GansnerKNV1993` methods are used at every step, except for edge routing, which defaults to the `simple` algorithm. This behavior can be changed easily with the help of dedicated options that Ti*k*Z provides as part of the library `graphdrawing.layered`. To remove cycles with the `BergerS1990b` algorithm, for example, the `/graph drawing/layered drawing/cycle removal` option needs to be changed.

```
\tikz \graph [layered drawing] {
  a -> b -> c -> d,
  a -> d
};
\tikz \graph [layered drawing={cycle removal=BergerS1990b}]
{
  a -> b -> c -> d,
  a -> d
};
```



In the same way, alternatives for all the other steps may be selected using the options `layer assignment`, `crossing minimization`, `node positioning` and `edge routing`.

4.4 Extension Interface of the Algorithm Framework

The modular layered drawing algorithm provides an easy way for graph drawing researchers and other people interested in experimenting with layered drawings to extend the list of algorithms shown in table 4.1.

4.4.1 Prerequisites and Naming Schemes

Adding a new implementation for any of the steps is as simple as dropping a Lua script into the `TEXMF` directory tree. The framework defines a strict naming scheme for scripts containing algorithms for the individual steps of the Sugiyama method, namely

`pgfgd-algorithm-modular-layered-cycle-removal-`⟨*algorithm*⟩`.lua`
`pgfgd-algorithm-modular-layered-layer-assignment-`⟨*algorithm*⟩`.lua`
`pgfgd-algorithm-modular-layered-crossing-minimization-`⟨*algorithm*⟩`.lua`
`pgfgd-algorithm-modular-layered-node-positioning-`⟨*algorithm*⟩`.lua`
`pgfgd-algorithm-modular-layered-edge-routing-`⟨*algorithm*⟩`.lua`

where ⟨*algorithm*⟩ is allowed to contain alphanumeric characters (`A-Z`, `a-z`, `0-9`), dashes (`-`) and underscores (`_`) only. Lua scripts matching this scheme are automatically picked up by Ti*k*Z without the need to touch any of the internal files. Each of the algorithms is associated with a Lua class that holds the actual implementation and is expected to reside in the corresponding Lua script. In Ti*k*Z, each algorithm can be accessed via a simple name that is mapped to the corresponding class and file name internally.

Resolving a sub-algorithm works by taking a human-readable name such as `longest path` or `GansnerKNV1993` from a Ti*k*Z option like `layer assignment` and mapping it to a file and class name. Filenames are generated by simply converting all spaces to dashes. In the above examples, `longest path` would therefore be converted to `longest-path`, while a name such as `GansnerKNV1993` would remain untouched. The result is combined with the file naming scheme described above. The filename expected of a `longest path` algorithm for layer assignment therefor is

```
pgfgd-algorithm-modular-layered-layer-assignment-longest-path.lua
```

The names of Lua classes shipped with Ti*k*Z are in camel case style, that is, instead of separating words with spaces, the first letter of each word is capitalized. In order to map algorithm names to camel case class names, the following rules are applied in order. First, the name of the step, e.g. `layer assignment`, and the name of the algorithm are concatenated. Dashes and underscores are converted to spaces. Afterwards, the initial letter of every word is capitalized and all spaces are stripped from the name. Thus, in our examples the resulting class names would be `LayerAssignmentLongestPath` and `LayerAssignmentGansnerKNV1993`.

Classes for sub-algorithms are required to implement two methods, `new()` and `run()`, used to instantiate and run the algorithm. Both methods have different signatures depending on the step the class implements. This is explained in more detail below.

4.4.2 Interface for Cycle Removal Algorithms

The modular layered drawing algorithm instantiates cycle removal implementations by passing a `Graph` object to the `new()` method. The `run()` method on the other hand takes no arguments and has no return value. For the subsequent steps to work properly, however, cycle removal algorithms are required to remove all cycles in the input graph.

As explained in section 4.1.1, cycles can be removed by reversing a subset of the edges called the feedback arc set. A natural approach to designing the cycle removal interface would be to have the `run()` method return such a set. However, I decided that it would be more convenient to implement an edge reversal feature in the `Edge` class itself and activate it by setting

the `reversed` property of an edge. This way, no data structure housekeeping is necessary and methods like `Edge:getIncomingEdges()` or `Edge:getOutDegree()` work transparently. Also, the main algorithm can later restore the original input graph by simply clearing the `reversed` property.

Overall, the basic recipe for cycle removal algorithms is this:

```
pgf.module("pgf.graphdrawing")

CycleRemoval⟨algorithm⟩ = {}
CycleRemoval⟨algorithm⟩.__index = CycleRemoval⟨algorithm⟩

function CycleRemoval⟨algorithm⟩:new(graph)
  local algorithm = { graph = graph }
  setmetatable(algorithm, CycleRemoval⟨algorithm⟩)
  return algorithm
end

function CycleRemoval⟨algorithm⟩:run()
  ...  use 'edge.reversed = true' to reverse edges whenever necessary ...
end
```

4.4.3 Interface for Layer Assignment Algorithms

Layer assignment implementations take a `Graph` object as the input to their `new()` method. Their `run()` method has no parameters and is supposed to return a `Ranking` object that represents a valid layering for the input graph.

The `Ranking` class provides a convenient way to manage and update the layers and layer orderings in a graph. It stores, for each node, its layer—which is set using `Ranking:setRank(node, rank)`—and its position in this layer. For each layer it stores the set of nodes assigned to it. The class also provides methods to enumerate and normalize the layers—i.e. shift them so that the minimal layer is 1—and reorder them by switching the position of two nodes or by computing new positions for all nodes in a layer at once. Another noteworthy feature of the `Ranking` class is that it supports arbitrary layers, that is, no distinction is made between discrete layers such as 1,2,3... or non-integer layers like 28.5 or 166.233734. This allows the class to be reused in the node positioning step where layers correspond to x-coordinates, which may be floating point numbers. The Ti*k*Z manual has a detailed description of the methods and features of the `Ranking` class.

Implementing a new layer assignment algorithm essentially boils down to implementing the same algorithm structure as is used for cycle removal algorithms, except that the algorithm class is named `LayerAssignment⟨algorithm⟩`.

4.4.4 Interface for Crossing Minimization Algorithms

The main task of crossing minimization is to the reorder nodes at each layer so that the overall number of edge crossings is reduced to a minimum. The previous layer assignment step returns a layering in the form of a `Ranking` object which is used as the initial layering of crossing minimization. As a consequence, crossing minimization algorithms take a `Graph` and a `Ranking` object as parameters to their `new()` method. Their `run()` method is required to return another `Ranking` object in which nodes have potentially been rearranged.

There are two approaches to generating new layer orderings. The first one is to create an entirely new `Ranking` instance and add nodes to layers in the desired order. The other method is to take the input `Ranking` and reorder nodes by calling one of the following methods (assuming that the object is called `layering`):

```
layering:switchPositions(left_node, right_node)
layering:reorderLayer(layer, get_index_func, is_fixed_func)
```

The `reorderLayer()` function is of particular interest as it allows to reorder an entire layer at once. In this case, `get_index_func` is a function that takes a position and a `Node` object and returns a new position for the node—e.g. the median of its neighbors. The `is_fixed_func` parameter is a function that takes the same input parameters and tells whether or not a node has a fixed position—which is the case with nodes that have no neighbors in the fixed neighbor layer. With this in place, updating the positions based on precomputed median values (with a median less than zero referring to a fixed node) is as simple as this:

```
local function get_index(n, node) return median[node]     end
local function is_fixed(n, node)  return median[node] < 0 end

layering:reorderLayer(layer, get_index, is_fixed)
```

Aside from a different class name and input/output conventions, the basic structure of crossing minimization algorithms follows the scheme used in previous steps.

4.4.5 Interface for Node Positioning Algorithms

Algorithms implementing node positioning once again receive a `Graph` object and a `Ranking` as input parameters. Their task is to assign each node in the input graph an x- and an y-coordinate, like so:

```
node.pos:set{ x = ..., y = ... }
```

The `run()` method is expected to modify the graph in-place and thus, takes no arguments and returns nothing. The layers and node positions at each layer of the input layering already reflect the relative placement of the nodes. However, the exact coordinates may differ across algorithms and depend on the dimensions of individual nodes as well as the desired horizontal and vertical separation.

A code template for node positioning algorithms can once again be extracted from previous steps by changing the class name prefix to `NodePositioning` and changing the function signatures.

4.4.6 Interface for Edge Routing Algorithms

Edge routing algorithms are used to adjust the shape of edges in the final drawing. Their job is to remove all dummy nodes and edges from the graph and draw the original edges in a visually pleasing way, ideally without edge/node overlaps. As input to `new()`, these algorithms receive a `Graph` object and a list of dummy nodes. In addition, each non-dummy edge in the graph has a list of corresponding dummy nodes stored in its `bend_nodes` property. No arguments are passed to its `run()` method and no return value is expected from a call to it either. However, if any dummy nodes remain in the graph this will lead to errors when passing the graph back to the TEX layer.

There are two ways to shape edges. Polyline edges (see figure 4.5) can be generated by filling the `bend_points` table of each edge with coordinates. To replace the dummy nodes along an edge, all that needs to be done is the following.

```
for edge in table.value_iter(graph.edges) do
  -- check whether we have an original edge
  if #edge.bend_nodes > 0 then
    for dummy_node in table.value_iter(edge.bend_nodes) do
      -- convert the dummy node into a bend point
      table.insert(edge.bend_points, dummy_node.pos:copy())
      graph:deleteNode(dummy_node)
    end
  end
end
```

A more flexible approach to bending edges is the `algorithmically_generated_options` property of edges. It can be used to inject T*i*kZ options into an edge, possibly overriding the options already specified by the user. The possibilities with this method are almost unlimited and range from simple left/right bends to complex path instructions by means of the `/tikz/to path` option. As usual, the T*i*kZ manual contains more information on this topic. The example below demonstrates how an edge can be bent to the left, entering/leaving at an angle of 135/45 degrees, and how it can be decorated as a zig-zag curve:

```
edge.algorithmically_generated_options['bend left'] = 45
edge.algorithmically_generated_options['postaction'] =
  '{decorate,decoration=zigzag}'
```

The basic structure of edge routing algorithms is exactly the same as that of the prior steps.

5

Evaluation and Benchmarks

In this chapter, the implemented graph drawing algorithms described in the previous chapters are evaluated. This evaluation is performed using standard graphs, graphs found in related literature and lecture nodes, and special such as graphs with clusters or graphs of finite state machines. Unlike other evaluation approaches that perform a detailed benchmark of the number of edge crossings, edge length distribution and other measurable properties of graph drawings, the evaluation in this chapter focuses on less technical aspects and a comparison of the different solutions implemented. It also briefly discusses performance matters for both types of algorithms I worked on in the course of the thesis.

Section 5.1 presents and compares drawings generated by the spring and spring-electrical algorithms presented in chapter 3 and discusses some of the known issues of these algorithms. In the second part of the chapter, section 5.2, the same evaluation is performed for the modular layered algorithm from chapter 4 and the different heuristics that I implemented. Layered drawings are often easy to tweak if they are not matching the expectations of users. Therefor, section 5.2 also presents some tricks to fix or adjust these drawings. The time required to draw all individual example graphs used in this chapter is measured and listed at the end of each part of the evaluation in order to give a hint at the overall performance of the developed algorithms.

5.1 Evaluation of the Spring and Spring-Electrical Algorithms

Chapter 3 broadly discusses the design of the new spring and spring-electrical algorithms added to T*i*kZ. It also presents options to configure these algorithms. However, the drawings produced by these algorithms for realistic input graphs are never shown or compared. This is the main purpose of this section.

5.1.1 Spring and Spring-Electrical Layouts of Standard Graphs

Figures 5.1, 5.3, 5.5 present a set of spring and spring-electrical layouts generated for standard graphs—consisting of complete graphs (K_n), circles (C_n) and grids—with the multilevel technique enabled. The same graphs are also drawn without the multilevel technique in figures 5.2, 5.4 and 5.6.

The main observation concerning the `Hu2006 spring` algorithm is that the layouts generated for standard graphs with and without graph coarsening look very similar and feature a high degree of symmetry. A noticeable difference can be observed with regards to scaling, as the drawings produced without the multilevel technique are typically larger and feature longer edges. This is something I noticed with most of the graphs I used for testing, even though the grid graphs with 6 and 16 nodes are an exception to this rule and additional exceptions can be found through experimentation. Another property of the `Hu2006 spring` algorithm is that it yields drawings with shorter edges the larger the input graphs get. See the grid with 16 nodes compared to the grid with 9 nodes, for instance, or the drawing of K_{10} compared to that of K_4. In general, both variations of the algorithm yield drawings with almost uniformly distributed edge lengths.

In chapter 3 it is stated that spring algorithms do not suffer from peripheral distortion. Given the slightly concave layouts generated for the grid examples this appears to be only partly true. The distortion may be less prominent compared to the spring-electrical layouts shown in figures such as 5.3, yet it is clearly present.

Like `Hu2006 spring`, the `Hu2006 spring electrical` algorithm succeeds in producing layouts that are highly symmetric. In terms of scaling, the drawings generated with and without graph coarsening are similar to the ones generated by `Hu2006 spring`. The relative positioning of nodes also resembles that of the corresponding spring layouts. What appears to be one of the few differences compared to the results of `Hu2006 spring` is that the spring-electrical drawings are not perfectly symmetric, which can be seen in the slightly distorted drawing of K_{10}. Another observation is that the size of graphs drawn with `Hu2006 spring electrical` does not have an impact on the edge lengths as strong as with `Hu2006 spring`. This cannot be said for all graphs—see K_{10} compared to K_4—but with coarsening enabled the edge lengths in drawings generated for grid graphs does only change subtly from the 9-nodes grid to the 16-nodes grid, if at all. The `Hu2006 spring electrical` results without coarsening are once again similar to those with coarsening except that that they are slightly larger.

What may come as a surprise is that the multilevel technique has almost no impact on the quality of drawings generated by both algorithms. This is different with the `Walshaw2000 spring electrical` algorithm, whose results are shown in figures 5.5 and 5.6. The quality of drawings

produced by this algorithm strongly depend on whether the multilevel technique is used or not. Without it, the results look as if they are unfinished. The situation can be improved for some graphs by allowing more iterations—e.g. 1000 instead of the default 500—but even that is often insufficient to produce good layouts. Overall, this supports the claims by Hu [31] and Walshaw [51] that the multilevel approach leads to be better layouts and suggests to turn on it on by default. This is also what Ti*k*Z does.

The `Walshaw2000 spring electrical` algorithm has a significantly different scaling behavior compared to the `Hu2006` algorithms. While the issues with dense graphs were solved successfully for these algorithms by scaling the initial layout—see section 3.3.1, drawings of dense graphs produced by `Walshaw2000 spring electrical` still suffer from strong compression. This can most likely be attributed to the absence of electric forces between adjacent nodes (see formula 3.18). Without the repulsive effect of electric forces, the only objective of the algorithm is to produce edges with a length that equals the natural spring dimension. It does so almost accurately by yielding an average edge length between 0.71cm and 0.88cm for the graphs in figure 5.5, even though this alone does not lead to visually appealing drawings. The only solutions to this might be to subtract only a fraction of the electric force of adjacent nodes, depending on the density of the graph, or to simply increase the natural spring dimension when drawing dense graphs.

A major disadvantage of using the `Walshaw2000 spring electrical` algorithm for standard graphs is that the results are distorted stronger compared to the `Hu2006` algorithms. In particular, none of the C_n graphs looks as expected and the grid graphs appear as if they fall over to one side.

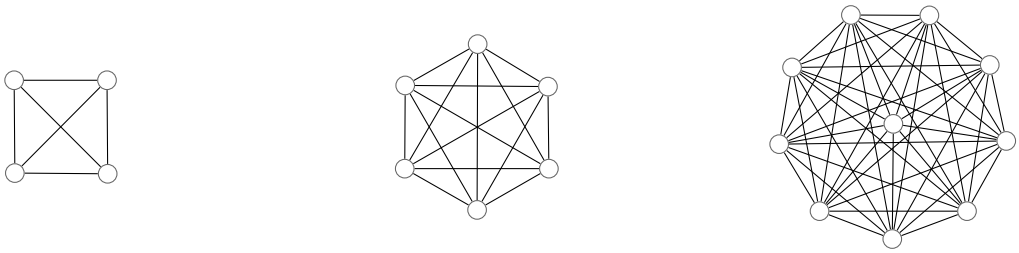
The drawings shown in figure 5.7 demonstrate that the developed spring-electrical algorithms perform well not only for small but also for large graphs. Despite working with two dimensional node coordinates, the resulting layouts reflect the three dimensional structure of the example graphs and in particular of the torus.

5.1.2 Spring and Spring-Electrical Layouts of Graphs from Literature

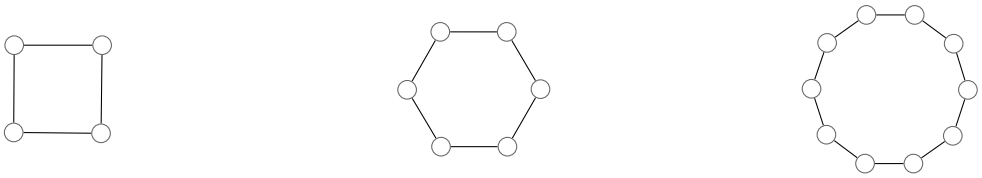
Figures 5.8, 5.9 and 5.10 show the results of applying the three spring and spring-electrical algorithms to graphs found in literature and lecture notes. Aside from different scaling behavior, less distortion and some problems with overlapping nodes in the drawings produced by `Hu2006 spring`, the visual structure of all layouts in these figures is similar. The generated drawings once again are highly symmetric.

What is worth noting is that Mathematica and Graphviz both ship implementations of the same `Hu2006 spring` and `Hu2006 spring electrical` algorithm. While their implementations may

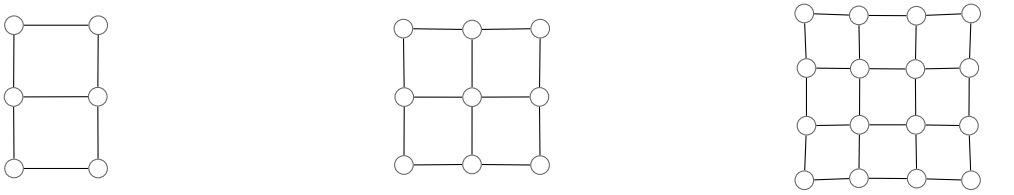
■ **Figure 5.1** Spring layouts of standard graphs drawn with `Hu2006 spring` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



K_4 K_6 K_{10}

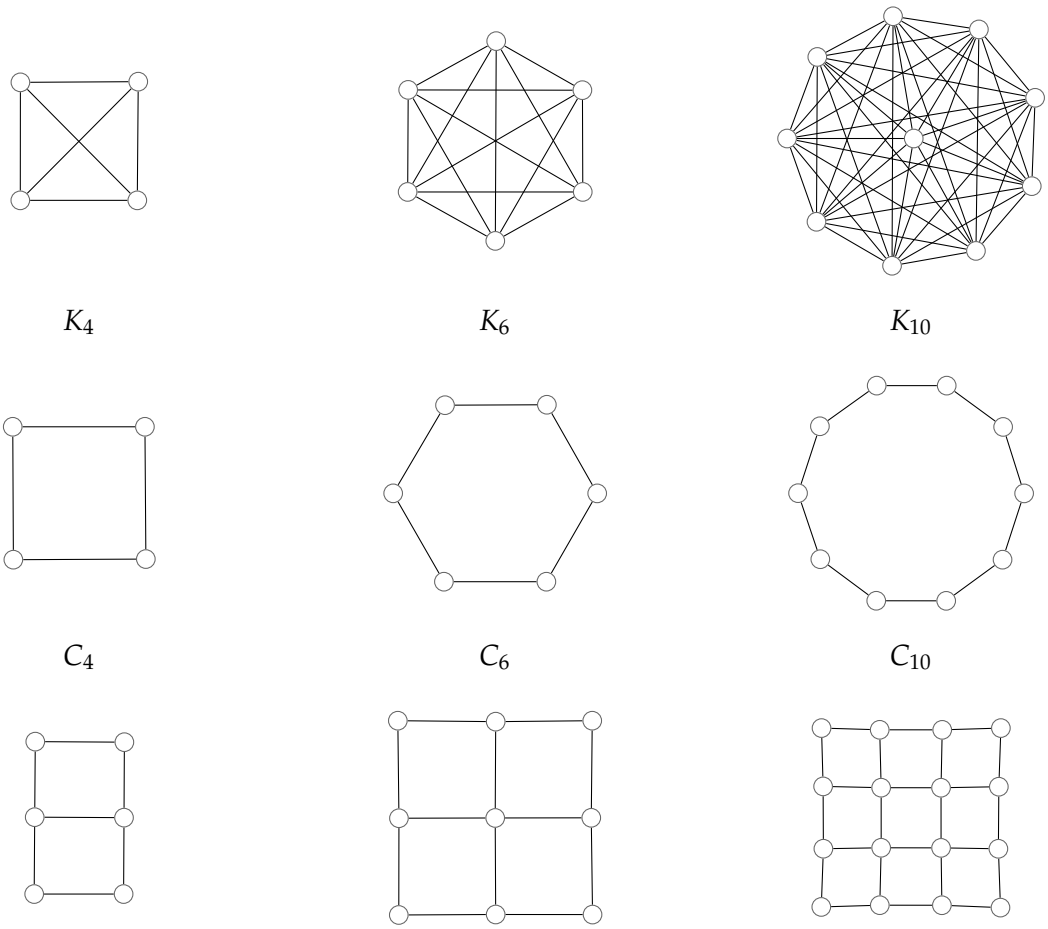


C_4 C_6 C_{10}

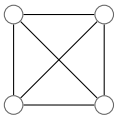


Grid with 6 nodes Grid with 9 nodes Grid with 16 nodes

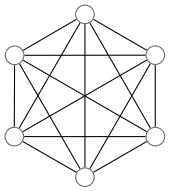
■ **Figure 5.2** Spring layouts of standard graphs drawn with `Hu2006 spring` and graph coarsening disabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



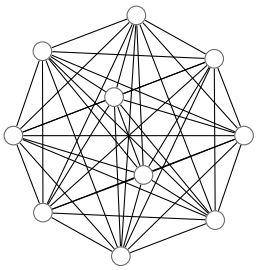
■ **Figure 5.3** Spring-electrical layouts of standard graphs drawn by `Hu2006 spring electrical` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



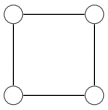
K_4



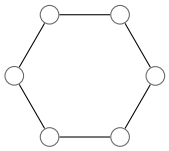
K_6



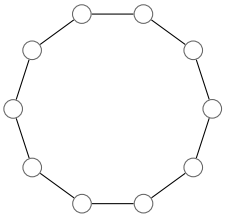
K_{10}



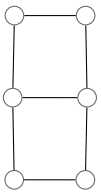
C_4



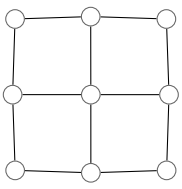
C_6



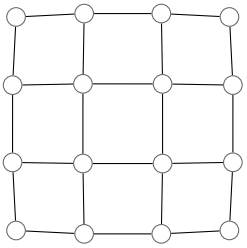
C_{10}



Grid with 6 nodes

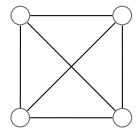


Grid with 9 nodes

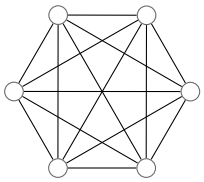


Grid with 16 nodes

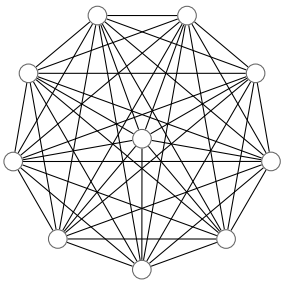
■ **Figure 5.4** Spring-electrical layouts of standard graphs drawn by `Hu2006 spring electrical` and graph coarsening disabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



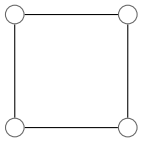
K_4



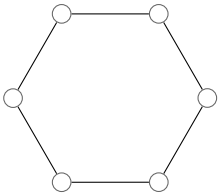
K_6



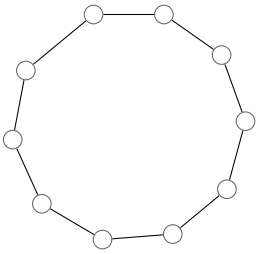
K_{10}



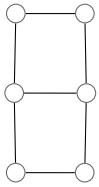
C_4



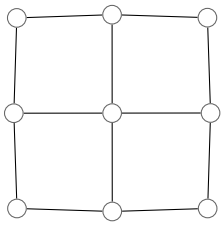
C_6



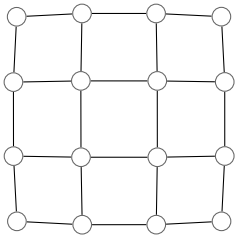
C_{10}



Grid with 6 nodes

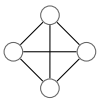


Grid with 9 nodes

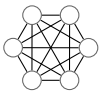


Grid with 16 nodes

■ **Figure 5.5** Spring-electrical layouts of standard graphs drawn with `Walshaw2000 spring electrical` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



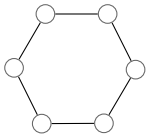
K_4



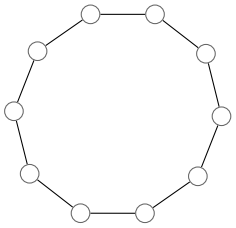
K_6

K_{10}

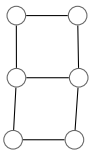
C_4



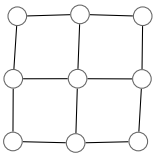
C_6



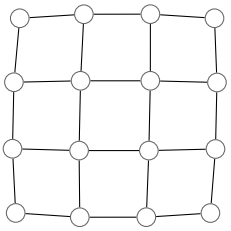
C_{10}



Grid with 6 nodes

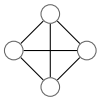


Grid with 9 nodes

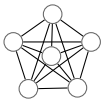


Grid with 16 nodes

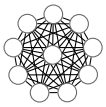
■ **Figure 5.6** Spring-electrical layouts of standard graphs drawn with `Walshaw2000 spring electrical` and graph coarsening disabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



K_4

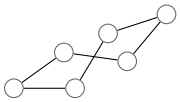


K_6

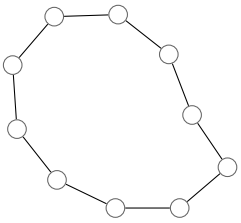


K_{10}

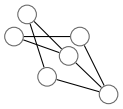
C_4



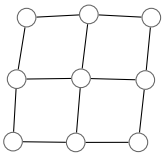
C_6



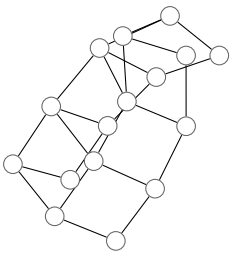
C_{10}



Grid with 6 nodes

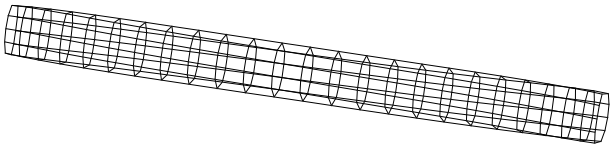


Grid with 9 nodes

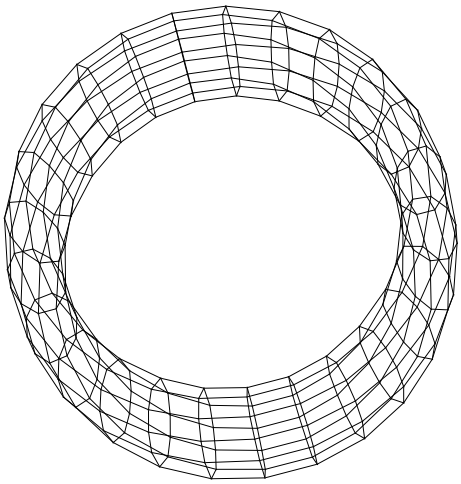


Grid with 16 nodes

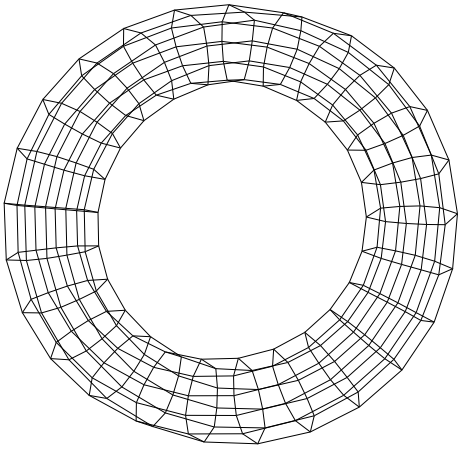
■ **Figure 5.7** Spring-electrical layouts generated for large graphs with graph coarsening enabled and a natural spring dimension of 2cm. The results are scaled down to fit onto fit onto the page.



Cylinder with 250 nodes drawn with `Hu2006 spring electrical`



Torus with 250 nodes drawn with `Hu2006 spring electrical`



Torus with 250 nodes drawn with `Walshaw2000 spring electrical`

differ from mine in details, the drawings produced by my implementation for the Mathematica and Traffic-Lights graphs—which were taken from [53] and the Graphviz website—are almost identical.

5.1.3 Graphs with Clusters

While working on the spring and spring-electrical algorithms, I was interested in seeing how well they would work on graphs with a nested cluster-like structure. One such example would be a graph with a number of nodes connected in a circle, with a complete subgraph like K_3 attached to each of these nodes. For my experiments I created different cluster graphs to which I applied the three algorithms. The results are shown in figures 5.11, 5.12 and 5.13.

What can be noticed is that, while rendering the pair of triples graph best of all algorithms, `Hu2006 spring` produces clusters so narrow that nodes tend to overlap. The drawings generated by `Hu2006 spring electrical` present the same behavior, albeit in a less distinct way. Both algorithms yield layouts with a high degree of symmetry. The drawings produced by the `Walshaw2000 spring electrical` algorithm are not always as well-balanced, see for instance the quadruple graph. However, they are distributed more evenly and the clusters have less tendency to collapse into narrow subgraphs with overlapping nodes.

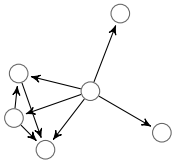
5.1.4 Performance

A major downside of spring and spring-electrical algorithms is their poor computational performance. In order to obtain a better understanding of how fast or slow the implemented algorithms are when being applied to typical graphs, their running times on all example graphs used in this evaluation were measured. The results of this are listed in table 5.1.

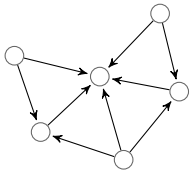
It is important to note here that all test graphs are small compared to some of the graphs used in other benchmarks. The effect of approximating electric forces on the performance of the spring-electrical algorithms was not measured as this makes sense only for graphs with a size of 50 nodes and more. The measured running times therefore only give a rough indication of the performance of the implemented algorithms in practical scenarios. Nevertheless, they do present a few interesting differences between these algorithms.

The first thing to notice is that, despite a few exceptions, the running times of all three algorithms are significantly shorter when graph coarsening is enabled. The speedup ranges from

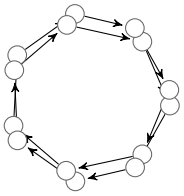
■ **Figure 5.8** Spring layouts of graphs collected from literature drawn with `Hu2006 spring` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



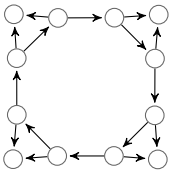
Mathematica-1



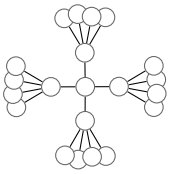
Mathematica-3



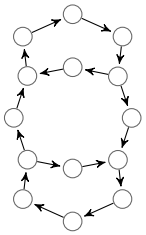
Mathematica-4



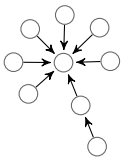
Mathematica-5



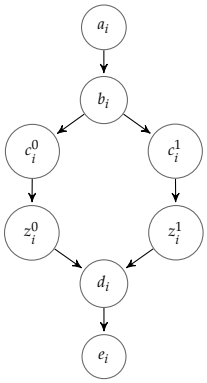
Complete-4-Ary-Tree



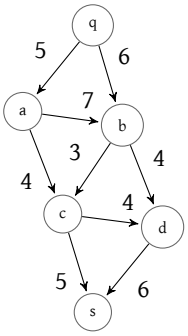
Traffic-Lights



Tantau-PV-2011-10.3.2

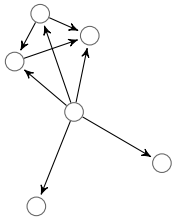


Jakoby-Algo-2011-8.4b

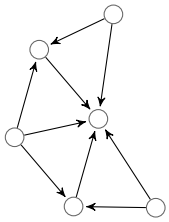


Reischuk-Algodes-2011-fig6

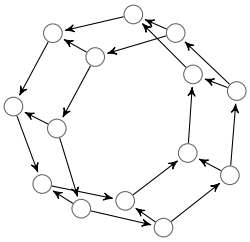
■ **Figure 5.9** Spring-electrical layouts of graphs collected from literature drawn with `Hu2006` `spring electrical` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



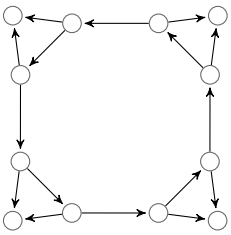
Mathematica-1



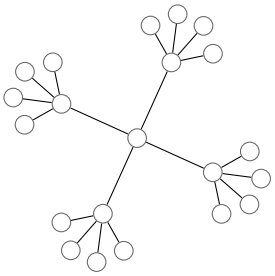
Mathematica-3



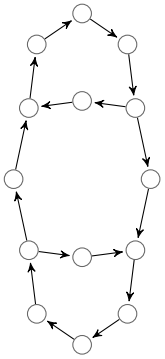
Mathematica-4



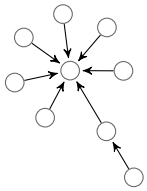
Mathematica-5



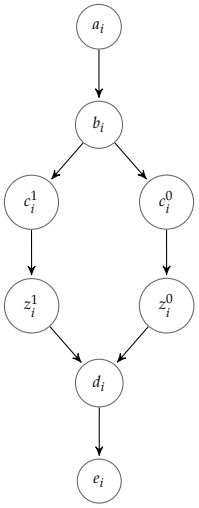
Complete-4-Ary-Tree



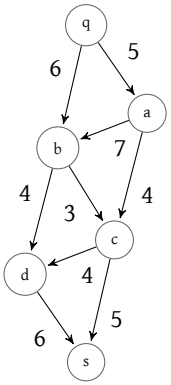
Traffic-Lights



Tantau-PV-2011-10.3.2

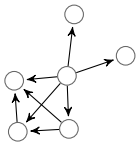


Jakoby-Algo-2011-8.4b

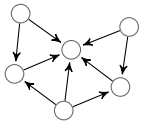


Reischuk-Algodes-2011-fig6

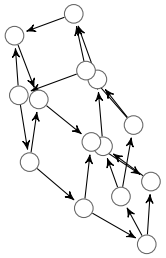
■ **Figure 5.10** Spring-electrical layouts of graphs collected from literature drawn with `Walshaw2000` `spring electrical` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



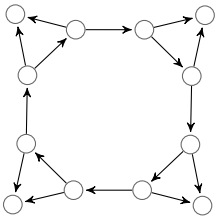
Mathematica-1



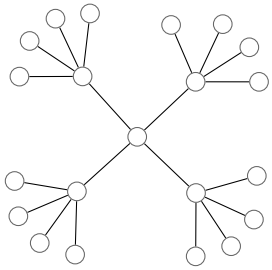
Mathematica-3



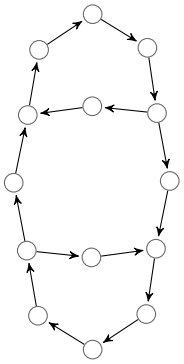
Mathematica-4



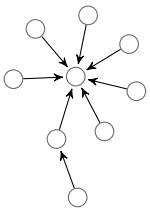
Mathematica-5



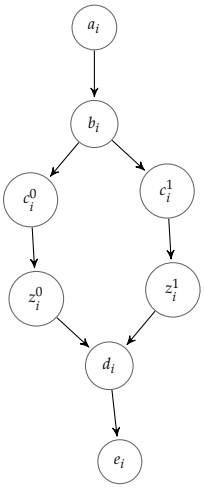
Complete-4-Ary-Tree



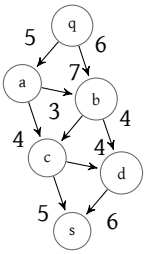
Traffic-Lights



Tantau-PV-2011-10.3.2

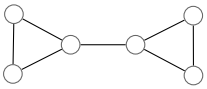


Jakoby-Algo-2011-8.4b

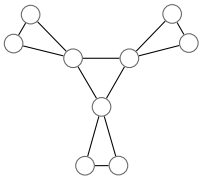


Reischuk-Algodes-2011-fig6

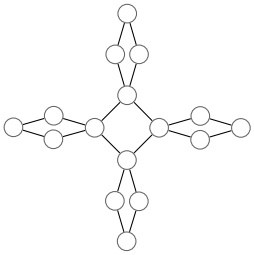
■ **Figure 5.11** Spring layouts of clustered graphs drawn with `Hu2006 spring` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



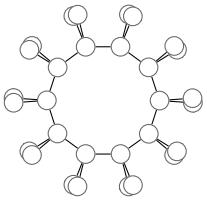
Pair of triples



Triple of triples



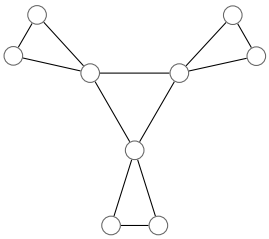
Quadruple of quadruples



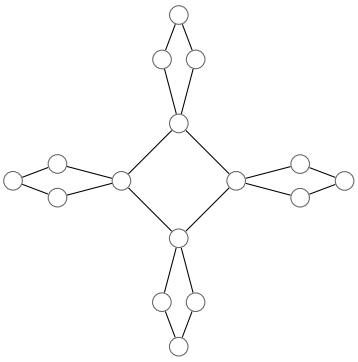
10-circle of triples

■ **Figure 5.12** Spring-electrical layouts of clustered graphs drawn with `Hu2006 spring electrical` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.

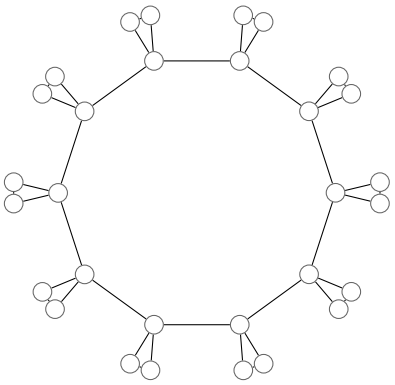
Pair of triples



Triple of triples

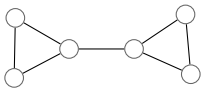


Quadruple of quadruples

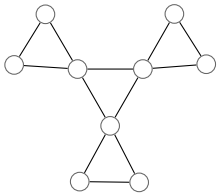


10-circle of triples

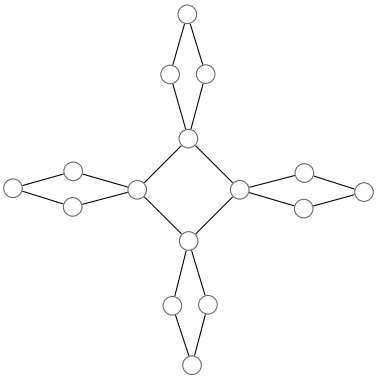
■ **Figure 5.13** Spring-electrical layouts of clustered graphs drawn with `Walshaw2000 spring electrical` and graph coarsening enabled. The natural spring dimension is set to 0.8cm, other parameters are kept at their standard values.



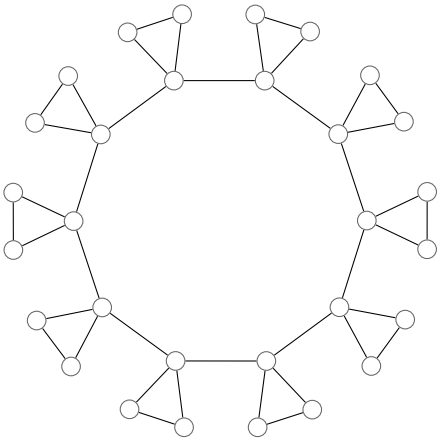
Pair of triples



Triple of triples



Quadruple of quadruples



10-circle of triples

30% to about 80%, which supports Hu's statement that algorithms based on the multilevel approach converge towards an optimal layout more quickly than regular spring or spring-electrical algorithms [31]. The `Hu2006 spring` algorithm based on his work appears to have better running times than the two spring-electrical algorithms implemented. Of these two, the `Walshaw2000 spring electrical` algorithm based on the work by Walshaw [51] outperforms `Hu2006 spring electrical` in most situations, most notably the generation of the complete 4-ary tree and the Tantau-PV-2011-10.3.2 graph.

An unfortunate observation that can be made for all three algorithms is that the time it takes them to generate the final layout grows quickly depending on the size of the input graph. While a small complete graph like K_4 is generated in an acceptable time of 0.04–0.37 seconds with coarsening enabled, larger graphs like the grid with 16 nodes or the complete 4-ary tree take 4.66–27.91 seconds already. It gets even worse with the layouts generated of the large cylinder and torus graphs for which the spring-electrical algorithms need between 652.01 and 1744.71 seconds to finish.

There are several possible reasons for the poor performance of these algorithms, which been reported to be able to generate layouts for sparse graphs with 10,000 nodes in less than 30 seconds [51]. One reason might be that, despite being considered a fast scripting language, Lua's performance is unlikely to match that of compiled languages. The Lua compiler developed by the LuaJIT project[10] might improve the situation by making Lua code run significantly faster. Unfortunately, the use of this compiler is not supported by LuaTEX yet. Another reason for slow running times might be that the code was written with readability in mind rather than performance. Its heavy use of iterators, elements of functional programming and specialized data structures are most likely the cause of significant slowdowns compared to performance-optimized code. The book *Lua Programming Gems* [13] and the website of a game called *Complete Annihilation*[11] discuss common performance issues with certain features of Lua, identifying iterators, inline functions and non-optimized table modifications as some of the main reasons for the reduced performance of Lua scripts, all of which are used frequently throughout my algorithms and the Ti*k*Z graph drawing engine in general.

Clearly, these issues will have to be addressed at some point in order to allow larger collections of graphs and graphs with many nodes to be embedded into real documents. With the current performance of the implemented algorithms, preventing drawings from being regenerated whenever a document is built becomes essential. The use of externalization features and other tricks is therefore recommended.

[10] LuaJIT project website: http://luajit.org
[11] Lua performance benchmarks: http://trac.caspring.org/wiki/LuaPerformance

■ **Table 5.1** Running times of the `Hu2006 spring` (HuS), `Hu2006 spring electrical` (HuSE) and `Walshaw2000 spring electrical` (WaSE) algorithms on all example graphs, with and without coarsening. The drawings were generated on a machine powered by an Intel Core2 Duo T9300 processor running at 2.50GHz using ConTEXt version 2011.06.13 and LuaTEX version beta-0.70.1-2011051923.

Graph	HuS coarsening	HuS	HuSE coarsening	HuSE	WaSE coarsening	WaSE
K_4	0.22s	0.66s	0.37s	1.01s	0.04s	0.05s
K_6	0.63s	1.33s	1.05s	2.53s	0.15s	0.67s
K_{10}	1.70s	2.91s	3.51s	5.37s	1.85s	1.54s
C_4	0.21s	0.47s	0.32s	1.00s	0.28s	0.47s
C_6	0.67s	1.42s	0.93s	1.56s	1.03s	0.94s
C_{10}	1.89s	3.74s	2.35s	9.03s	2.65s	2.36s
Grid, 6 nodes	0.72s	1.37s	0.39s	1.86s	0.74s	1.24s
Grid, 9 nodes	1.51s	2.92s	2.26s	3.31s	2.68s	1.98s
Grid, 16 nodes	4.66s	9.66s	5.92s	9.38s	7.58s	6.01s
Cylinder, 250 nodes			1212.65s[12]			
Torus, 250 nodes			652.01s		1744.71s[12]	
Mathematica-1	0.71s		1.53s		1.48s	
Mathematica-3	0.60s		1.53s		1.09s	
Mathematica-4	4.01s		7.06s		7.25s	
Mathematica-5	3.17s		5.83s		3.39s	
Complete-4-Ary-Tree	16.90s		27.91s		10.00s	
Traffic-Lights	3.40s		6.56s		6.07s	
Tantau-PV-2011-10.3.2	2.70s		6.39s		1.88s	
Jakoby-Algo-2011-8.4b	1.00s		2.16s		1.56s	
Reischuk-Algo-des-2011-fig6	0.57s		1.49s		0.29s	

5.2 Evaluation of the Layered Drawing Algorithm

Like with the spring and spring-electrical algorithms, the `modular layered` graph drawing algorithm is evaluated based on example drawings. For this, I collected a set of graphs from literature and lecture notes. The results are shown in figures 5.14 and 5.15, respectively.

In general, these drawings are on par with those generated by Graphviz and Mathematica. This is not surprising as both tools implement the same algorithm based on [27]. Graphviz and Mathematica often traverse nodes in a different order, which may lead to alternative layouts.

The main criteria for evaluating layered drawings are the number of back edges, edge crossings, edge bends and edge lengths. The examples in figure 5.14 demonstrate that, in its standard configuration, the `modular layered` algorithm highlights the general flow of the graphs well and is successful in avoiding back edges. Only the drawings of the Mathematica-4 and Mathematica-5 graphs taken from [53] contain back edges, which are, however, easy to spot and do not make the graphs harder to understand. Applied small graphs, the algorithm appears to have no problems generating layered drawings that have only a few edge crossings and bends, even though it does not produce optimal results in all situations—see for instance the drawing of BarthMJ-fig2 where the edge crossing could easily be avoided. Overall, most edges are as short as possible. The distribution of edge lengths is almost uniform, with only 3–4 edges per graph deviating from the rest.

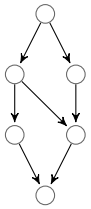
The effect of switching between different algorithms at one of the steps of `modular layered` is demonstrated in figure 5.16, which shows the same graph drawn with four different cycle removal algorithms. The properties and problems of these algorithms are already explained in section 4.1. In figure 5.16, the `EadesLS1993` cycle removal algorithm produces the best layout with only three layers, one edge crossing, three bent edges and the most uniform distribution of edge lengths compared with the other drawings. What is worth noting is that none of them is able to produce optimal results for all possible input graphs. It is thus recommended to try different combinations of sub-algorithms whenever the default configuration does not yield satisfying results.

Unsurprisingly, the drawings in figure 5.16 are less evenly distributed and feature more crossing than the smaller graphs in figures 5.14 and 5.15. This is a natural consequence of using cycle removal and crossing minimization heuristics that only work between neighbored ranks rather than globally. Figure 5.17 is an example of a large graph that is arranged relatively well considering its size. However, a significant increase in the number of crossings, bends and long edges is to be expected in general whenever layouts for large graphs need to be generated.

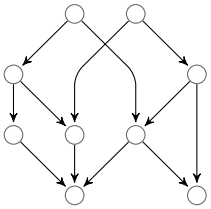
---

[12] This result is not reliable as other computations were performed in parallel.

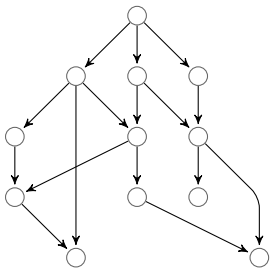
■ **Figure 5.14** Layered drawings produced by the `modular layered` algorithm for graphs found in graph drawing literature. All graphs are processed using the standard heuristic at each step of the algorithm except for Mathematica-4, which is generated with the `BergerS1990b` cycle removal heuristic. The distance between layers and siblings is set to 0.8cm in all drawings.



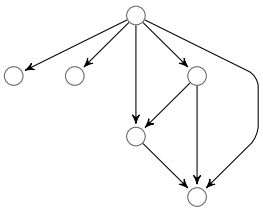
EadesS1990-fig3



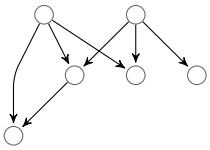
EadesS1990-fig9



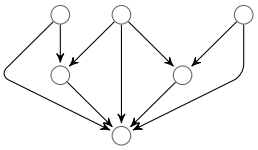
BarthMJ-fig2



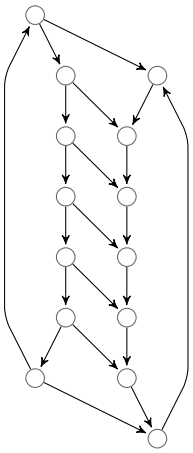
Mathematica-1



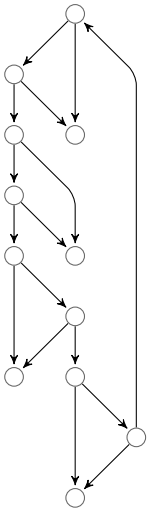
Mathematica-2



Mathematica-3

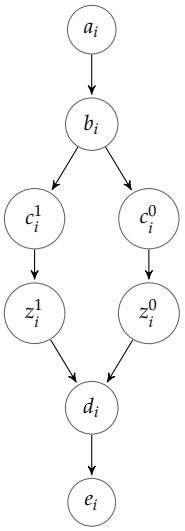


Mathematica-4

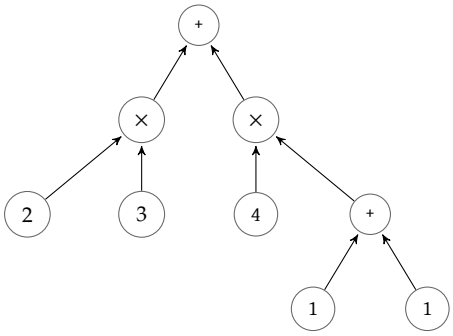


Mathematica-5

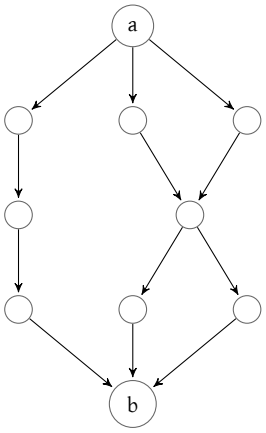
■ **Figure 5.15** Layered drawings produced by the `modular layered` algorithm for graphs collected from lecture notes. All graphs are processed using the standard heuristic at each step of the algorithm. The distance between layers is set to 1.25cm, the distance between siblings is 1.5cm in all drawings.



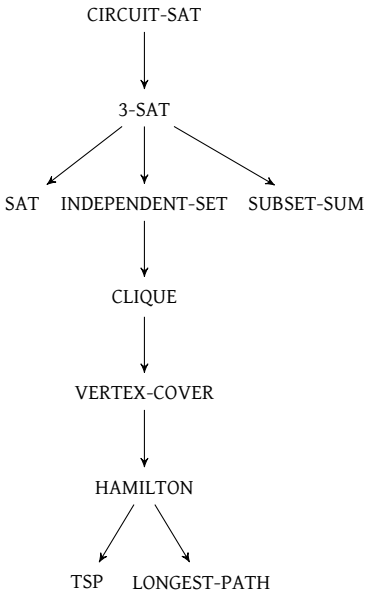
Jakoby-Algo-2011-8.4b



Tantau-PV-2011-11.2.1

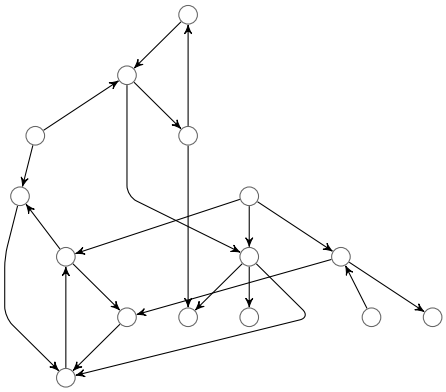


Tantau-PV-2011-3.1.1

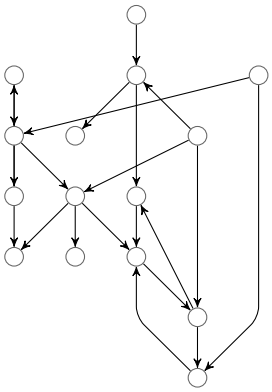


Tantau-Theoretical-CS-2009-28-13

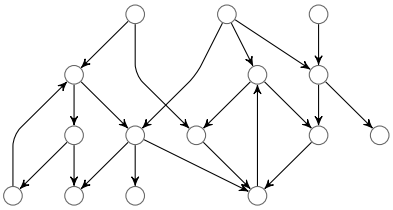
■ **Figure 5.16** A layered drawing of an example graph produced by the `modular layred` algorithm using four different cycle removal heuristics. All other steps are preformed using the standard techniques. The random seed was set to 3 for the `BergerS1990b` example.



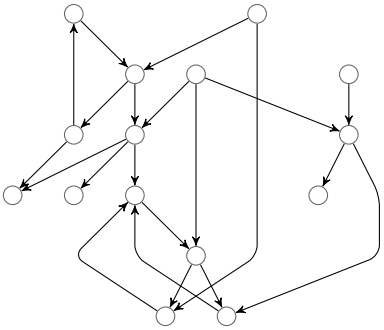
BastertM2001-1
`BergerS1990a` cycle removal



BastertM2001-1
`BergerS1990b` cycle removal

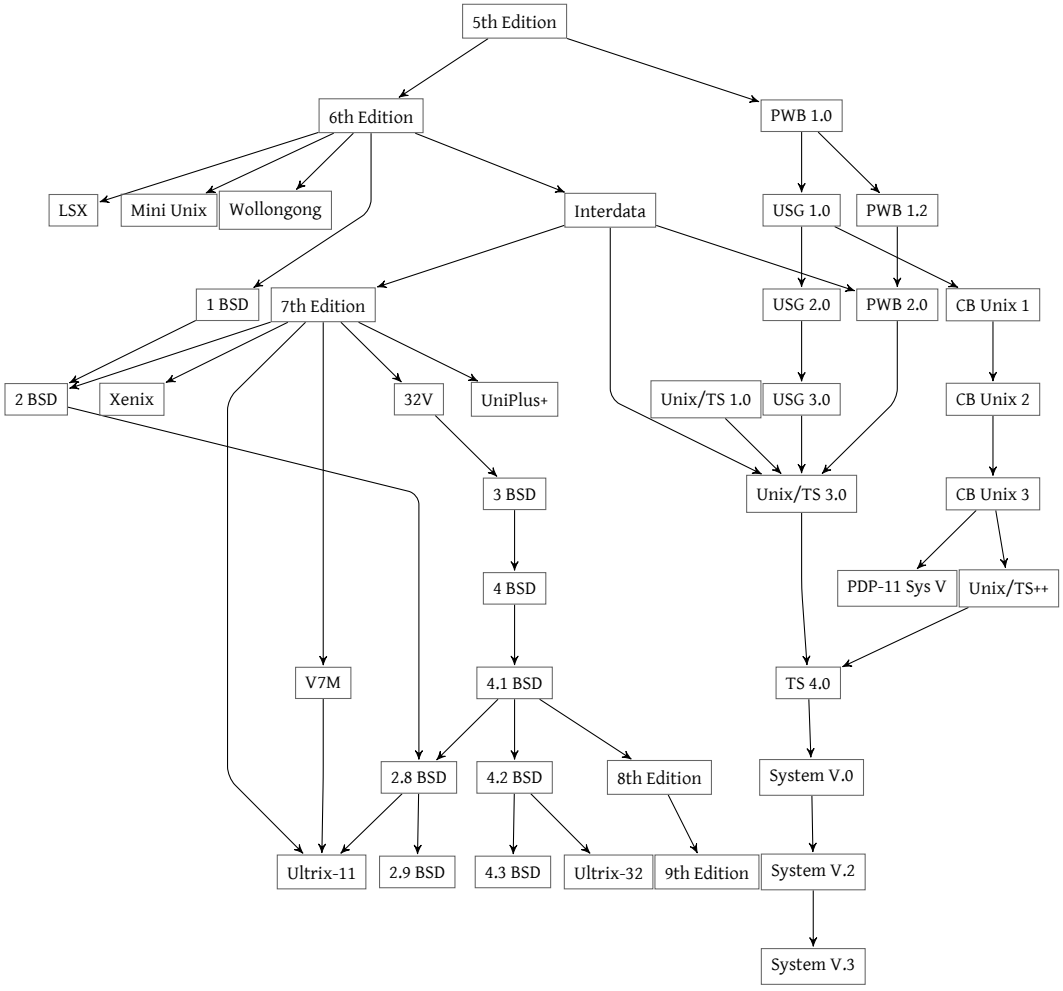


BastertM2001-1
`EadesLS1993` cycle removal



BastertM2001-1
`GansnerKNV1993` cycle removal

■ **Figure 5.17** A layered drawing of the history of UNIX created with the `modular layered` algorithm using the standard method at each step. A level and sibling distance of 1.25cm is used to separate the nodes.



Performance

Compared to the spring and spring-electrical algorithms, the `modular layered` algorithm is significantly faster. As opposed to `Hu2006 spring electrical`, which requires between 5.92 and 9.38 seconds for a grid with 16 nodes, the layered drawing algorithm is capable of drawing the UNIX-History graph with 41 nodes in 3.98 seconds. The layered drawings of all other example graphs are generated in a second or less, as is shown in table 5.2. The Lua optimizations as described as necessary in section 5.1 are also applicable to the `modular layered` algorithm, which would further improve its performance.

Table 5.2 Running times of the `modular layred` algorithm on all example graphs, using the default method at each step of the algorithm. The performance of other methods such as `BergerS1990a` cycle removal or `longest path` layer assignment are not listed as the differences are marginal. The drawings were generated on a machine powered by an Intel Core2 Duo T9300 processor running at 2.50GHz using ConTEXt version 2011.06.13 and LuaTEX version beta-0.70.1-2011051923.

Graph	`modular layered`
EadesS1990-fig3	0.14s
EadesS1990-fig9	0.35s
BarthMJ-fig2	0.53s
Mathematica-1	0.23s
Mathematica-2	0.15s
Mathematica-3	0.23s
Mathematica-4	0.81s
Mathematica-5	0.57s
Jakoby-Algo-2011-8.4b	0.13s
Tantau-PV-2011-11.2.1	0.17s
Tantau-PV-2011-3.1.1	0.23s
Tantau-Theoretical-CS-2009-28-13	0.16s
BastertM2001-1	1.14s
UNIX-History	3.98s

6

Conclusion and Outlook

This is the first thesis to cover the new graph drawing features of Ti*k*Z, including its compact graph syntax and the graph drawing engine. It is also the first to illustrate how its Lua-based algorithm framework can be used to develop new graph drawing algorithms to support users in creating graph drawings with Ti*k*Z. In that sense, the successful implementation of the spring, spring-electrical and layered drawing algorithms is not the only achievement made, but it is also the work on completing the graph drawing engine itself that is to be considered one of the major accomplishments of this thesis. Hopefully, the results will provide a sound basis for future research and development.

The spring and spring-electrical layouts as well as the layered drawings produced by the algorithms developed in this thesis are comparable in terms of quality to those produced by other tools such as Graphviz and Mathematica. The extensibility of the algorithm framework for layered drawings is a clear advantage over other solutions as it opens an attractive opportunity to foster experimentation with new ideas and algorithms. Given the common structure of the spring and spring-electrical algorithms, a similar framework could have been developed for force-based algorithms as well. This conclusion, however, only came up as an afterthought. A notable drawback of the implemented algorithms is their slow performance. Possible solutions and workarounds for this problem were discussed in chapter 5 and will have to be addressed in future work.

Open Problems

Despite satisfying a number of use cases, the algorithms developed in this thesis do not cover all related issues yet. Supporting arbitrary node sizes and avoiding overlaps in graph drawings

is a difficult problem whose resolution would clearly make all implemented graph drawing algorithms more valuable. Various articles have been written about this topic [23, 24, 29 and 41] and might provide useful information for researchers and developers interested in implementing possible solutions.

In Ti*k*Z, nodes and edges may have multiple labels attached to them. These labels may again be arbitrarily complex nodes. Processing such label nodes with Lua algorithms is something that is not supported yet. Future research efforts could go into extending the Lua framework by an interface to access and modify labels in Lua algorithms. Computing an optimal label placement is then another challenge that could be worked on, based on ideas presented in [37 and 39].

Another open problem is routing edges so that they are visually appealing and do not cross other edges or nodes. The framework for layered drawing algorithms already provides an interface for implementing dedicated algorithms for edge routing as one of the steps of the Sugiyama method. Whether this interface is powerful enough remains yet to be seen. Possible edge routing algorithms that could be integrated into Ti*k*Z are presented in [3 and 27]. The spring and spring-electrical algorithms, on the other hand, include no such interface and assume all edges are to be drawn straight. An interesting approach that could perhaps be applied here is a technique called *confluent drawings* presented by Dickerson et al. [11].

Possible Areas of Improvement and Future Development

Aside from the big open problems described above, a number of straight-forward improvements and extensions of both, the algorithms and the graph drawing engine are conceivable. They range from changes in algorithms and underlying data structures to additional features that are directly beneficial to users. The following list is not extensive but nevertheless gives an idea of possible areas of future work.

Many of the implemented techniques either work with auxiliary graphs or temporary modify the input graph itself. Creating copies of a graph, associating nodes and edges of the copy with those of the original graph, as well as keeping track of temporarily removed, merged, and added edges has proven to be complicated and error-prone with the graph class provided by Ti*k*Z. This could be improved significantly by offering a high-level interface for managing different versions of the same graph without the need to keep track of changes manually. The conception of such an interface is a major challenge on its own and its introduction would affect all areas of the graph drawing engine, which is why it is left open as a future development task.

Something that definitely needs to be solved are the scaling issues of Walshaw's spring-electrical algorithm mentioned in chapter 5. These issues are most likely due to the absence of electric

forces between adjacent nodes. A solution for this might be to always include electric forces but scale them depending on the density of the input graph.

At the time of writing, only a few basic constraints are implemented such as fixed coordinates for nodes in spring and spring-electrical layouts or horizontal clusters in layered drawings. In order to make the developed algorithms more flexible, additional constraints are worth adding. A number of constraints supported by both drawing methods are presented in [12] along with canonical solutions to implementing them. The clustering feature of layered drawings could be made more powerful in particular, for instance by introducing special clusters for the first and last layer as well as vertical clusters to align nodes at the same x-coordinate.

Two elegant features supported by Mathematica [53] are special drawing techniques for loops and multiple edges as well as routines for drawing the connected components of a graph independently and arranging them in a grid afterwards. Clearly, having the same features in Ti*k*Z would be useful.

Another plausible addition would be a Ti*k*Z macro to query the list of algorithms available for the different steps of the modular layered drawing framework. Without such a macro, the only way for users to find out about the alternatives they have is to browse the TEXMF directory tree.

The work presented in this thesis introduces graph drawing features valuable for users and graph drawing researchers. It can be seen as an effort to not only provide effective graph drawing algorithms that are applicable to a wide range of practical scenarios, but also as an attempt to lay the groundwork for future research and algorithm development using Ti*k*Z. In addition to the open problems and possible improvements discussed here, many alternative drawing methods are yet to cover in order to exploit the full potential of graph drawing. This thesis demonstrates how this can be done for two such methods—force based layouts and layered drawings—and hopefully manages to spark interest in using Ti*k*Z as a tool for drawing graphs and performing future graph drawing research based on the facilities it provides.

Bibliography

1. R. Bergmann. *Interaktive und automatisierte Hypergraphenvisualisierung mittels NURBS-Kurven*, Diplomarbeit, University of Lübeck, 2009.
2. J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.
3. O. Bastert and C. Matuszewski. Layered drawings of digraphs. In M. Kaufmann and D. Wagner, editors, *Drawing graphs: methods and models*, Lecture Notes im Computer Science, 2025:87–120, Springer, 2001.
4. W. Barth, P. Mutzel, and M. Jünger. Simple and efficient bilayer cross counting. *Journal of Graph Algorithms and Applications*, 8(2):179–194, 2004.
5. Graph Drawing. In F.-J. Brandenburg, editor, *Proc. Symposium on Graph Drawing, GD '95*, Lecture Notes in Computer Science, 1024, Springer, Passau, 1996.
6. B. Berger and P. Shor. Tight bounds for the acyclic subgraph problem. *Journal of Algorithms*, 25:1–18, 1997.
7. M. Carpano. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Software Engineering*, SE-12(4):538–546, 1980.
8. E. Coffman and R. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.
9. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to algorithms*, McGraw-Hill Higher Education, 2001.
10. W. H. Cunningham. A network simplex method. *Mathematical Programming*, 11(1):105–116, 1976.

11. M. Dickerson, D. Eppstein, M. T. Goodrich, and J. Y. Meng. Confluent drawings: visualizing non-planar diagrams in a planar way. *Journal of Graph Algorithms and Applications*, 9(1):31–52, 2005.
12. G. Di Battista, P. Eades, R. Tmassia, and I. G. Tollis. *Graph drawing: algorithms for the visualization of graphs*, Prentice Hall, Upper Saddle River, NJ 07458, 1999.
13. In L. H. de Figueiredo, W. Celes, and R. Ierusalimschy, editors, *Lua Programming Gems*, Lua.org, Rio de Janeiro, 2008.
14. J. Dubinski. A parallel tree code. *New Astronomy*, 1(2):133–147, 1996.
15. P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
16. P. Eades, X. Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47:319–323, 1993.
17. G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.
18. P. Eades and K. Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–436, 1990.
19. P. Eades and S. Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131(2):361–374, 1994.
20. P. Eades and N. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.
21. T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.
22. A. Frick. Upper bounds on the number of hidden nodes in Sugiyama's algorithm. *Graph Drawing*, Lecture Notes in Computer Science, 1190:169–183, Springer, 1997.
23. C. Friedrich and F. Schreiber. Flexible layering in hierarchical drawings with nodes of arbitrary size, *Proc. 27th Australasian conference on Computer science, Vol. 26*, 26:369–376, Australian Computer Society, Inc., 2004.
24. E. R. Gansner and Y. Hu. Efficient node overlap removal using a proximity stress model. In Tollis, I.G. and M. Patrignani, editors, *Graph Drawing*, Lecture Notes in Computer Science, 5417:206–217, 2009.
25. M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*, W. H. Freeman and Company, New York, NY, 1979.
26. M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM. J. on Algebraic and Discrete Methods*, 4:312–316, 1983.
27. E. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.
28. A. Goldberg. Network flow algorithms. *Algorithms and Combinatorics*, 9, 1990.

29. D. Harel and Y. Koren. Drawing graphs with non-uniform vertices, *Proc. Working Conference on Advanced Visual Interfaces - AVI '02*, ACM Press, New York, 2002.
30. B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In S. Karin, editor, *Proc. Supercomputing '95*, ACM Press, New York, 1995.
31. Y. Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 2006.
32. A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.
33. R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, Plenum Press, 1972.
34. Y. Koren and A. Civril. The binary stress model for graph drawing. In I. Tollis and M. Patrignani, editors, *Graph Drawing*, Lecture Notes in Computer Science, 5417(1):193–205, 2009.
35. T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.
36. E. Kruja, J. Marks, A. Blair, and R. Waters. A short note on the history of graph drawing. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, Lecture Notes in Computer Science, 2265:602–606, Springer, 2002.
37. K. G. Kakoulis, B. Madden, and I. G. Tollis. Edge labeling in the graph layout toolkit. In S. H. Whitesides, editor, *Graph Drawing*, Lecture Notes in Computer Science, 1547(70):356–363, Springer, Heidelberg, 1998.
38. D. J. Kelly and G. M. O. Neill. *The minimum cost flow problem and the network simplex solution method*, Ph.D. thesis, University College Dublin, 1991.
39. K. G. Kakoulis and I. G. Tollis. On the edge label placement problem. In S. North, editor, *Graph Drawing*, Lecture Notes in Computer Science, 1190(70):241–256, Springer, Heidelberg, 1997.
40. K. Mehlhorn. *Data structures and algorithms, volume 2: graph algorithms and NP-completeness*, Springer, New York, 1984.
41. K. Marriott, P. Stuckey, V. Tam, and W. He. Removing Node Overlapping in Graph Layout Using Constrained Optimization. *Constraints*, 8:143–171, 2003.
42. N. S. Nikolov and A. Tarassov. In search for eficient heuristics for minimum-width graph layering with consideration of dummy nodes. *ACM Journal of Experimental Algorithmics*, 10:1–27, 2005.
43. H. C. Purchase, R. F. Cohen, and M. James. Validating graph drawing aesthetics. *Graph drawing*, Lecture Notes in Computer Science, 1027:435–446, 1996.

44. H. C. Purchase, M. Mcgili, L. Colpoys, and D. Carrington. Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. In P. Eades and T. Pattison, editors, *Australian Symposium on Information Visualisation*, Conferences in Research and Practice in Information Technology, 9:129–137, 2001.
45. G. Paulino, I. Menezes, M. Gattass, and S. Mukherjee. A new algorithm for finding a pseudoperipheral vertex or the endpoints of a pseudodiameter in a graph. *Communications in numerical methods in engineering*, 10(11):913–926, 1994.
46. H. C. Purchase, C. Pilcher, and B. Plimmer. Graph drawing aesthetics — created by users not algorithms. *IEEE transactions on visualization and computer graphics*, 2010c.
47. H. Purchase. Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interacting with Computers*, 13(2):147–162, 2000.
48. G. Robbins. The ISI grapher, a portable tool for displaying graphs pictorially. *Technical Report IST/RS-87-196*, Information Sciences Institute, Marina Del Rey, CA,.
49. G. Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing 1994*, Lecture Notes in Computer Science, 894:194–205, Springer, 1995.
50. K. Sugiyama, S. Tagawa, and M. Toda. Effective representations of hierarchical structures. *Research report number 8*, 1979.
51. C. Walshaw. A multilevel algorithm for force-directed graph drawing. *Mathematics Research Report 00/IM/60*, University of Greenwhich, London, 2000.
52. C. Walshaw. A multilevel algorithm for force-directed graph drawing. In J. Marks, editor, *Graph Drawing*, Lecture Notes in Computer Science, 1984:31–55, 2001.
53. Wolfram Research. Wolfram Mathematica Tutorial Collection: Graph Drawing, 2011.