JMFGraph—A Modular Framework for Drawing Graphs in Java

Alexander Stedile

JMFGraph—A Modular Framework for Drawing Graphs in Java

Master's Thesis

at

Graz University of Technology

submitted by

Alexander Stedile

Institute for Information Processing and Computer Supported New Media (IICM),
Graz University of Technology
A-8010 Graz, Austria

18^{th} November 2001

Advisor: Univ.Ass. Dr. Keith Andrews

JMFGraph—Ein modulares Gerüst zum Zeichnen von Graphen in Java

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Alexander Stedile

Institut für Informationsverarbeitung und Computergestützte neue Medien (IICM),
Technische Universität Graz
A-8010 Graz

18. November 2001

Diese Arbeit ist in englischer Sprache verfaßt.

Betreuer: Univ.Ass. Dr. Keith Andrews

Abstract

This thesis describes JMFGraph. JMFGraph is an open, modular graph drawing framework for displaying and browsing graphs. JMFGraph is implemented in Java following the object-oriented programming paradigm. The program includes two variations of Sugiyama's hierarchical graph layout algorithm, a standard and a focus-based version.

The framework is equipped to support both a static layout mode (global view mode) and a dynamic mode (local view mode) which creates a graph layout with respect to one focussed node in the displayed graph. By repeatedly moving the focus to different nodes the user may explore the displayed graph. This navigation method is often called "browsing". JMFGraph additionally provides the possibility to visualise intermediate results from sub-algorithms by a user controlled step-by-step execution mode.

Interfaces for modular extensions are provided. This extendibility concerns layout algorithms and sub-algorithms, graphical representations for nodes and edges, and modules for inputting graphs from a variety of sources.

Kurzfassung

Diese Diplomarbeit beschreibt JMFGraph. JMFGraph ist ein offenes, modulares Graphenzeichengerüst zum graphischen Darstellen und interaktiven Besichtigen von Graphen. JMFGraph ist in Java nach dem objectorientierten Programmierparadigma geschrieben. Das Programm beinhaltet zwei Variationen des hierarchischen Algorithmus zur Berechnung von Graphen-Layouts von Sugiyama: eine Standardversion und eine Fokus basierte Version.

Das Programmgerüst ist mit Unterstützung sowohl für einen statischen Layout-Modus (global view mode), als auch für einen dynamischen Modus (local view mode) ausgestattet. Letzteres gestaltet die Darstellung eines Graphen in Hinblick auf einen ausgezeichneten, „fokussierten“ Knoten. Indem dieser Fokus wiederholt auf andere Knoten verschoben wird, kann der dargestellte Graph von einem Benutzer untersucht werden. Diese Art der Navigation wird oft „browsing“ (engl. schmökern, grasen) genannt. Zusätzlich bietet JMFGraph die Möglichkeit, Zwischenergebnisse von Teilalgorithmen im Rahmen einer schrittweisen Ausführung des Darstellungsalgorithmus anzuzeigen.

Es stehen Schnittstellen zur modularen Erweiterung zur Verfügung. Diese Erweiterbarkeit betrifft: Layout-Algorithmen und Teilalgorithmen, graphische Representationen für Knoten und Kanten und Module zur Einlesen von Graphen aus verschiedensten Quellen. Diese so genannten „Modi“ sind dafür verantwortlich, eine Verbindung zu jeweils einer spezifischen Graphenquelle aufzubauen.

Diese Diplomarbeit ist in englischer Sprache verfaßt.

I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.

Ich versichere hiermit, diese Arbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

Acknowledgements

Now, after my master's thesis has been written successfully, I want to gratefully thank all those nice people who supported me. Support that has been invaluable for me not only for the preparation of this thesis but also during the years since I began my studies.

First, for the thesis supporters, I want to thank my advisor Keith Andrews. He proved a whole lot of patience, especially in the first months after starting the work for my thesis. In many productive discussions he helped me to set up this document. Thanks go also to Klaus Schmaranz who gave me highly qualified help and directions for the design of my graph drawing framework. Also highly helpful persons were my English course teachers Clare Melcher-Goddard and Frank Newman who succeeded in enhancing my English skills. Further thanks for helping me go to Franz Aurenhammer.

However, the writing of this thesis would have been impossible without the important help of many people during my studies. As most important overall supporters I want to gratefully thank my parents. For the years of my studies they gave me generous financial donations and mentionable amounts of patience, too. Thanks also to my grandparents for mental and financial aid. Grandmother Grete B.: Thank you for being so proud of me.

Not to forget, thanks to Verena's family for all kinds of help and support. Greetings and thanks go to my friends Karli, Guy, Barbara, Martin, Marc, my brothers Gerald and Roland, my mates from the university sports courses aikido, volley-ball, trampoline, and all my other friends.

Last, but not least, special thanks go to my fiancée Verena for acceptance of and encouraging during—sometimes not so pleasant—periods of work. It did not make things easier that the time for our theses' writing unfortunately happened to coincide in the recent months, but now we are happy and proud of having mastered all the trouble and difficulties.

Alexander Stedile
Graz, Austria, November 2001

Credits

- The introduction to basic paradigms of graph drawing (Chapter 2) follows the corresponding chapters of [dBETT99] and [dBETT94].
- Figure 5.10, is taken from `http://www2.iicm.edu/keith/papers/vis97/ipyr.html`. Used with kind permission of Keith Andrews.
- Figure 5.8 from [AC96] was provided by Mark Apperley and is used with his kind permission.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Human beings can keep seven (plus or minus two) distinct elements simultaneously in their short term memory (STM) [Mil56]. Unfortunately this capacity often limits one's efficiency; not only in technical areas, but also in daily life. One very powerful trick to overcome this handicap is grouping. Grouping simply means to put some—for example seven—things into a bag. Then the bag can be labelled with any kind of symbol representing a common property of everything contained. In fact, this procedure reduces the apparent objects' quantity from several elements to only one bag. An expressive description of this strategy is "reduction by abstraction" or "chunking". The bag's label abstractly represents all the contained elements.

Continuing the above idea, the next step is to put bags as well as elements into bigger bags. If eventually everything is wrapped into one biggest bag, a hierarchy is obtained. Now let us take a look at a theoretical search operation:

1. Somebody wants to find an element within the big outer bag. By examining the outer bag's label it can be found out that the desired element should be inside.

2. Now this bag is opened and finds some elements and some more bags. If the searched element is not found amongst the unwrapped ones, another bag has to be opened. The labels, describing the contents, tell which one.

3. The previously described step is continued until the element is found. Or, if the element is not present, the searcher may stop if no more bag's label matches the searched element.

Of course, this all works only if expressive labels have been carefully chosen. If ambiguous labels are selected, difficulties arise when picking the next bag to be opened. Two very common applications of hierarchical structures are telephone numbers and computer file systems. Worldwide, the telephone networks are divided into countries, regions, cities, and districts. The second example, computer file systems, are pervasive in current computer systems. Data files are organised in folders and subfolders.

The hierarchical structure has many advantages. It is very understandable to human users because people are used to classifying things. A word for this behaviour—although in a negative, overdone way—is "pigeonholing"; which means to put everything into some supposedly fitting bag. The number of elements in each bag can be kept very small. This is important in connection with the number 7 as stated above. If too many elements (and bags) happen to be at the same level, an additional level can be introduced easily by subgrouping the contents.

However, hierarchical structure has its limitations. Elements, for example, can not generally be inside two different bags simultaneously. The only way to overcome this restriction would be to

duplicate the element in question, although duplicate elements are generally an unfavourable solution. Another shortcoming is that two different bags can not be mutually contained. Even duplication is no solution here, because it would result in a cycle and therefore infinite replications. The theoretical reason for this is a hierarchy's property of *cycle freeness* explained in Chapter 2.

If those shortcomings are critical, the *graph* might be the structure of choice. Defined broadly as set of elements with connections (see also Chapter 2) it hardly has any limitations, at least as long human beings think in terms of objects and their properties and relationships. Besides hierarchies and graphs, several attempts to combine them have been proposed, combining the strong and clear order of hierarchies with the generality and flexibility of graphs.

Knowing that people like structure, this knowledge should be used to enhance communication. The partners for this communications could be teacher and pupils, two engineers, or a computer user interacting with an information system. The latter is the main aim of this thesis. By enhanced visualisation methods it is possible to make information displayable in a more human-friendly way. This is the basic prerequisite for making ever growing information masses understandable and manageable for human users.

This thesis describes JMFGraph, the Java Modular Framework for Graph Drawing, and is organised as follows. Chapter 2 summarises the most important facts and terms from graph drawing theory, basics which are important for the considerations in subsequent chapters. Graph drawing parameters (drawing conventions, aesthetics, constraints, and efficiency) are discussed. Methods for storing graphs, both in memory and in file systems are considered. Additional links and citations for further reading are given.

Using these basics, Chapter 3 introduces current approaches for graph drawing and visualisation. Representative algorithms are discussed in detail. Chapter 4 presents several particular algorithms developed for use in the hierarchical visualisation approach. Such visualisation concepts are also called Sugiyama-style algorithms. A selection of graph drawing and information visualisation programs is given in Chapter 5. Graph browsers, editors, and graph drawing systems are covered as well as toolkits and libraries for graph drawing.

The program JMFGraph is described in Chapter 6. This chapter introduces to the main concepts and the program's structure. Chapter 7 discusses selected details of the implementation. Algorithms especially developed for JMFGraph are documented in detail. Chapter 8 considers how this thesis and JMFGraph match the current trends in research and application development. Subjects for further development are suggested. In Chapter 9 final remarks on this thesis are noted.

Appendix A presents a user guide for the JMFGraph program. It covers installation and starting as well as an explanation of the user interface. In Appendix B the requirements for the program are specified as a user requirements document (URD). Appendix C presents specifications for the dot file format used by JMFGraph including specific adaptions. This thesis closes with the bibliography.

Chapter 2

Graph Drawing

Contents

Algorithms for information visualisation often depend on algorithms from graph theory and graph drawing. Therefore the basics and terminology of graph theory are of importance when dealing with visualisation. Section 2.1 is addressed to readers who are not very familiar with graph drawing theory. It is necessary to give precise definitions for the most important terms. In Section 2.2 some layout variations are considered which can help to make graphs more readable. An introduction to drawing conventions, aesthetics, constraints, and efficiency considerations. Section 2.3 gives an overview of how graphs can be stored in memory or in file systems. The main points are briefly summarised in Section 2.4, links and sources for further detailed information are listed.

2.1 Graph Drawing Terminology

2.1.1 Structures for Information

Some structures which are important in this thesis are explained below. A network is a structure which consists of nodes and links. Each link connects two nodes. Table 2.1 shows some examples for applications of the network paradigm and more specialised data structures. This table shows both degenerate and more obvious networks. A list, for instance, would probably not be seen as a network, but it can be modelled as such.

Structure	Example	Objects	Connections
List	linked list	items	successor, predecessor
Tree	family tree	persons	parent-child relations
	computer file system	files and folders	'contains' relations
Network	relational database model	entities	relationships
	molecule	atoms	bindings
	computer network	computer, server, workstation, terminal	wire, fibre cable
	World Wide Web (WWW)	document	hyperlink
	neural network	neurons	dendrites, axons

Table 2.1: Applications of the data structures list, tree, and network.

The term *network* denotes a set of objects which are somehow interconnected. This, of course, is a very general model and therefore widely applicable. When talking about networks, readers might immediately think of a computer network (for instance the Internet) or a neural network, depending on their personal background. Therefore one purpose of this section is to show the generality of the term "network".

Owing to this generality there are several sub-structures with more or less strict additional restrictions. Here the most common ones are described briefly. A *tree* is a network where exactly one path of connections can be found between any two objects. A *rooted tree* is a tree with one object specified as the root of the tree. Some authors call a tree which is not rooted a *free tree*. More abstract definitions for tree data structures are given in the next section. A *list* is a connected row of objects (see Table 2.1).

Some algorithms use more specialised data structures. For example linked hierarchies or compound networks. Such structures will be discussed when dealing with the respective algorithm.

2.1.2 The Graph

Definition 2.1 (Graph) *A graph is an abstract structure that is used to model information. Graphs are used to represent information that can be modelled as objects and connections between those objects. [dBETT99]*

From this definition the close relation between networks and graphs can be seen. Nevertheless, the terms should be distinguished. Here the network is the underlying information structure and the graph is an abstract model for it.

Figure 2.1 shows a very simple example for a graph: Three atoms, drawn as circles and labelled with their chemical symbols, building a molecule. In this graphic representation the atoms are the objects. The atom bindings, drawn as straight lines, are the connections between these objects. In graph theory the objects are usually called *nodes*. The connections are called *edges*.

Definition 2.2 (Node) *In graph theory a node is the abstract representation for an underlying object of the information structure being modelled. A different name for node, common in graph theory, is* vertex.

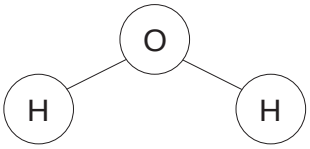


Figure 2.1: An undirected graph: H_2O molecule.

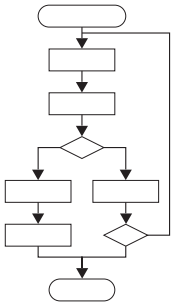


Figure 2.2: A directed graph: a flow chart.

Definition 2.3 (Edge) *An edge, or also* connection*, is the abstract representation for a relationship between two objects. Those objects are abstractly represented by an edge connecting the respective two nodes. Edges can be directed or undirected. In graph theory edges are also called* links *or* arcs*.*

There are many different properties of graphs. They help to classify and specify graphs. *Digraph* is simply an abbreviation for "directed graph". This means that the graph's edges are directed. In digraphs the connections are oriented and have a *source node* and a *destination node*. See for example Figure 2.2. Undirected graphs may be modelled as directed graphs with an arbitrary orientation of the edges. This orientation is simply ignored when drawing an edge.

A graph is called *simple* if it contains no self-loops and no multiple edges. A *self-loop* is an edge of which source node and destination node are the same. *Multiple edges* means that two nodes are connected by more than one edge. When all nodes are linked to each other the graph is said to be a *connected* graph. This means in different words there are no two nodes which are not connected by one or a series of edges. A graph is *planar* if and only if it can be drawn in the plane without edge crossings [CS96].

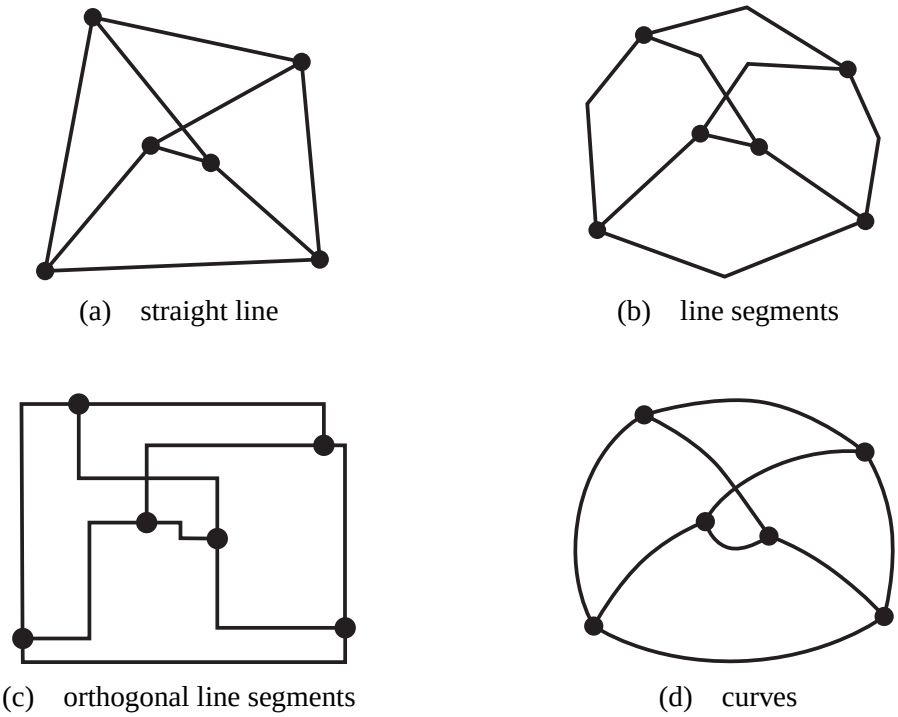
A directed graph is *acyclic* if it has no directed cycles. A *directed cycle* is a path that leads from one node along at least one edge back to the same node. The edges have to be followed according to their directions. An acyclic digraph with a single source s and a single sink t is called an *st-graph* [dBETT99]. In a digraph with cycles, a *feedback arc set* is a set of edges, such that if all the edges of such a set are reversed, then the graph is cycle-free.

Knowing these terms, a much simpler definition for the tree data structures can be given: A *tree* is a connected acyclic graph. And a *rooted tree* is a tree with one node specified as root. When representing as a digraph, all edges are oriented either away from or towards the root.

2.1.3 Graph Drawings

Being particular about terms, Figures 2.1 and 2.2 are not graphs, but drawings depicting graphs. The drawing of a graph is the graph's diagrammatic visualisation in the plane, i.e. in two dimensional space. A graph has infinitely many different drawings [dBETT94]. This is because the graph does not fix the position of its nodes and edges.

In graph theory nodes are depicted as points (see Figure 2.3). For visualisation purposes better representations are arbitrary symbols. They help to identify the underlying object. In applications it is common to draw nodes as boxes, circles, or simple shapes. See Figure 2.2 for example. Like in Figure 2.1 nodes are often labelled with informative text. Also edges can be drawn in a great variety of styles. The edge is abstractly defined as "simple open (Jordan) curve" [dBETT99, Tol96, dBETT94]. In simple words this definition says, "The edge can be any thin, continuous curve." Some of the



(a) straight line (b) line segments

(c) orthogonal line segments (d) curves

Figure 2.3: Examples for different edge styles.

most common edge drawing styles are illustrated in Figure 2.3. As for nodes, it is also common to label edges with additional information. All of these graphic representations—for edges as well as for nodes—have diverse applications. In some fields, history played an important role for picking a widely accepted graph style.

Not only graphs but also their drawings have certain properties. The drawing algorithm has to implement the desired visualisation of a given graph. A drawing of a graph is *planar* if it is drawn in two dimensions without edges crossing. As the relation between graph and drawing is not unique, planar graphs also have non-planar drawings. A *planar embedding* specifies the circular order of the edges around a vertex in a planar drawing. Hence, different drawings may have the same planar embedding. Note that a planar graph may have an exponential number of planar embeddings [CdBT$^+$92]. A *visibility representation* maps vertices into horizontal segments and edges into vertical segments that intersect only the two corresponding vertex segments [CdBT$^+$92].

The aim of graph drawing is to convey the properties of the underlying information structure as quickly and clearly as possible. For this purpose all kinds of positioning, alignment, symmetry, perspective, and valence techniques may be utilised. Such methods are discussed in the next section.

2.2 Parameters for Graph Drawing

It is a fact that a graph has infinitely many different drawings. Since our aim is to obtain an understandable graph, it is not sufficient to pick one of the many possible drawings randomly. Parameters are introduced which restrict the number of possible drawings, making it possible to choose from the great variety of graphical representations.

These restrictions affect mainly node placement and edge style. For one drawing several restric-

Drawing Convention		Explanation
Node placement	grid	Node positions, and in general also edge crossings and bends, are restricted to coordinates which are defined by a rectangular grid.
Edge style	straight line	The edges may only be straight lines without bends and curves.
	polyline	The edges may consist of one or more straight line segments.
	orthogonal	The edges may consist of straight line segments, but the segments must be either horizontal or vertical.
	edge ports	If the nodes are larger than simple points, the positions where an edge can start or end are restricted.
Edge routing	planar	A planar graph must be drawn planar, i.e. with no edges crossing.
	upward	In the drawing for a directed graph all directed edges have to point upward. Other directions, like downward or to the right, are equally common.

Table 2.2: Examples of drawing conventions for graphs drawings.

tions may be applied. In the following, parameters are discussed by which graph drawing algorithms and approaches may be distinguished. *Drawing conventions* (Section 2.2.1 and Table 2.2) define the basic demands for the desired drawing, for example where nodes may be placed or how edges have to be drawn and routed. *Aesthetics* (Section 2.2.2 and Table 2.3) are requirements which enhance readability, often including complex optimisation problems. *Constraints* (Section 2.2.3 and Table 2.4) are local requirements such as predefined positions of single nodes or special grouping of sets of nodes. *Efficiency* (Section 2.2.4) often is one of the most important points. The layout is expected to be calculated within a certain time which highly depends on the application. Acceptable periods of time might span from tenths of seconds to several hours. Since computation time increases with increasing number of elements, the two central questions are: “how long may the calculation take” and “how large may the displayed graph be”. In addition to this, memory consumption is often a very crucial point.

2.2.1 Drawing Conventions

The drawing conventions are the strongest restrictions. They have the highest priority and define the rules for drawing a graph. The principle is shown in Table 2.2. In this table, one rule is an example of node placement (node placement: grid). The other rules specify where edges may be laid and how they must look. An additional drawing convention would be to define how a node must be drawn. In contrast to aesthetics and constraints, drawing conventions are compulsory. They define the basic building blocks for the visualisation. Some authors also refer to drawing conventions as *graphic standards* [Tol96, dBETT94].

2.2.2 Aesthetics

Aesthetics concern properties of the drawing which can make it more readable and understandable. Table 2.3 shows a selection of common aesthetics. Most of them are optimisation goals which makes them computationally hard. Therefore it is generally necessary to compromise between running time

(see Section 2.2.4) and optimal readability [dBETT99, Tol96]. Graph drawing algorithms usually select more than one aesthetic, requiring a definition of precedence. This makes the aesthetics' "if it is possible" character obvious. It should be noted that aesthetics are subjective and may need to be tailored to suit personal preferences, applications, traditions, and culture [Tol96, dBETT94].

2.2.3 Constraints

Drawing conventions and aesthetics refer to the entire diagram, but constraints only affect part of the drawing or even just one node. Table 2.4 lists common constraints for graph drawing algorithms. Constraints can only operate in the space which the restrictions from drawing conventions and aesthetics leave open.

2.2.4 Efficiency

As already stated in Section 2.2.2 most aesthetics are computationally hard. This means that, depending on the size of the graph, the calculation time for the drawing can grow far beyond the period of time a user is willing to wait for the diagram. In such cases, heuristic algorithms can help to speed things up. The aesthetics adopted might be imperfect but are often good enough.

Efficiency is a crucial issue especially when dealing with interactive applications, which need real-time response.

2.3 Storing Graphs

When dealing with graphs on computers, the necessity to input, output, and store the graph is obvious. These demands can be distinguished according to the media. The graph has to be stored in memory in order to draw, display, and manipulate it. For storage on mass media, which also serves for information interchange with other applications, a different format is usually utilised.

2.3.1 Storing Graphs in Memory

Today, graphs in memory are usually modelled in an object-oriented way, mostly because of the popularity of object-oriented programming languages, but also because the graphs' natural structure can easily be remodelled by objects and references (see also Definition 2.1).

Theoretic and educational books usually propose two different approaches: the adjacency matrix and the edge list.

Adjacency Matrix

A two-dimensional array with dimensions $(|V|, |V|)$ is implemented, where $|V|$ is the number of nodes in the graph. The rows of the array stand for the start node, the columns for the end node. Figure 2.4 shows an example. The depicted graph has 5 nodes numbered from 0 to 4. The drawn edges are shown in the corresponding adjacency matrix.

Each cell of the array indicates the existence or non-existence of a possibly directed edge. For this purpose only one bit is required. Alternatively, numbers indicating the edge's weights can be stored in the array. Self loops are possible in this representation. For multiple edges additional measures are necessary.

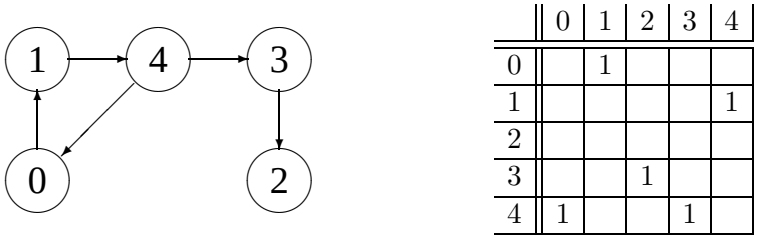
Aesthetic		Explanation
Area	used	The overall (rectangular) area of the drawing is minimised.
	aspect ratio	Attempt to uniformly fill a given aspect ratio for the drawing. (e.g. screen or paper format)
Node placement	symmetry	Symmetries of the graph should be obvious in the drawing. (e.g. tree symmetry)
	mental map	When changing an existing drawing, the nodes should keep their positions with respect to the entire drawing as well as to their neighbours. This helps the viewer to find out what changed and what remained unchanged.
Edge bends	total	The overall number of edge bends is minimised.
	maximum	The maximum number of one edge's bends is minimised.
	uniform	It is tried to distribute the necessary bends evenly to all edges.
Edge length	total	The sum of the lengths of all edges is minimised.
	maximum	The length of the longest edge is minimised.
	uniform	The deviation of the edges from the average edge length is minimised.
Edge crossings	total	The overall sum of edge crossings is minimised.
	maximum	The maximum number of crossing on one edge is minimised.
	uniform	The deviation of the edges from the average number of edge crossings is minimised.
Edge angle	angular resolution	Keeps the minimum angle formed by two edges incident on the same vertex from becoming too small.
	minimum angle	Keeps angles between edge segments from becoming too small.

Table 2.3: Examples of aesthetics for graph drawings.

Constraint		Explanation
Place node	centre	A selected part of the graph should be placed in the middle of the drawing.
	external	A selected part of the graph has to appear at the outer bounds of the drawing.
	cluster	Nodes which belong to the same defined group should be placed near each other.
	shape	Selected nodes have to appear in a defined shape in the drawing (for example sequence, tree, ring, ...).

Table 2.4: Examples of constraints for graph drawings.

Due to its high memory consumption of $|V|^2$, the adjacency matrix is only favourable for dense graphs. This can be seen from the fact that a non-existent edge consumes as much memory as an existing one. Undirected graphs use only half the space, because $a \rightarrow b$ is equivalent to $b \rightarrow a$. Some graph theoretic algorithms model an undirected edge by two directed parallel edges with opposite directions.



	0	1	2	3	4
0		1			
1					1
2					
3			1		
4	1			1	

Figure 2.4: The adjacency matrix for storing graphs in memory.

Edge List

Edge list implementations are usually somewhat slower than adjacency matrix implementations, but in contrast to the matrix only existing edges consume memory. Its memory consumption depends on the number of nodes ($|V|$) and the number of edges ($|E|$) and sums up to $|V| + |E|$. This can also be seen in Figure 2.5 where the same graph as in Figure 2.4 is depicted with an edge list representation.

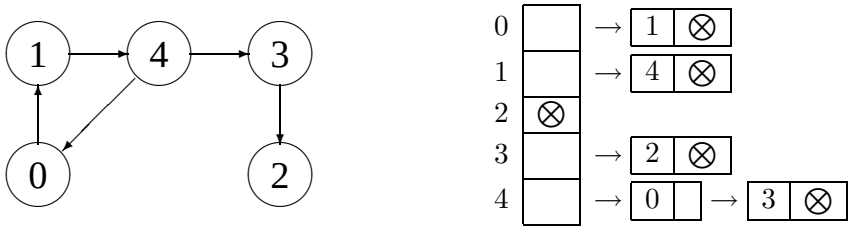


Figure 2.5: An edge list for storing graphs in memory. The symbol ⊗ marks the end of the respective edge list.

Advanced Graph Data Structures

Both the adjacency matrix and the edge list provide only the most basic functionality for storing graphs. For most algorithms dealing with graphs, additional information has to be stored. For example, a visited flag might be used to indicate which nodes have already been viewed. Or list and pointer structures might be introduced to speed up the answers for requests like "Which nodes are connected by edge e?"

2.3.2 Storing Graphs in the File System

Many current and emerging applications would benefit from the possibility to write graphs in a standardised format for information interchange. Despite this, no widely accepted graph format has yet evolved. Applications for graphs use many proprietary formats which cannot be applied universally. Those formats are often very specialised to favour efficiency. They provide definition of nodes as well

as interconnecting edges. Clustering nodes and labelling nodes and edges are often the most sophisticated features offered. Inclusion of objects (sub-graphs) or references are usually not supported.

At the recent International Symposium on Graph Drawing [GD'00] a Graph Data Workshop [BMN00] was held. Its aim was to help evolve a necessary standard. The participants identified requirements and formed a group to work out a proposal. The current status of this project is publicly available at the URL given for [BMN00]. An attempt to create a new standard for graph storage was taken at this year's Symposium on Graph Drawing [GD'01]. This attempt was not successful, instead a working group was suggested to formulate a future standard. Therefore going into much detail in this section would makes little sense. Some representative current formats will be discussed.

The graph drawing program Dot uses a proprietary format which is a rather simple and has only limited capabilities. In contrast to this, GraphXML builds upon the well known and widely accepted XML standard. This format is discussed in more detail here because of its highly flexible design.

Dot Format

Dot [Nor93] is a preprocessor for drawing directed graphs which can be drawn as hierarchies. It can output a variety of graphics and image formats. The input file is an attributed graph with nodes which must be uniquely labelled. Edges can be declared without identification. It also allows the construction of subgraphs. Additionally, several options for graphs or subgraphs, nodes, and edges can be applied, including size, font, colour, and other layout modifiers.

GraphXML

GraphXML is a graph description language in XML. It can be used as an interchange format for graph drawing and visualisation applications or packages [HM00, BPSM98].

GraphXML supports not only the declaration of a graph, it can also store a graph's drawing. Furthermore it is possible to attach application-dependent data and external references. Series of graphs may be ordered in time yielding a graph animation. GraphXML also offers the flexibility to introduce new features via XML's Document Type Definition (DTD) files. Nodes, edges, and graphs may be labelled, formatted, positioned, and provided with references and other arbitrary attributes. Graphs can be composed from different files at different locations in a network. The detailed specification and a parser for the format can be obtained from the URL given for [HM00].

A similar approach is taken in GML [Him97]. However, GraphXML's reference and multi-file capabilities are not included.

2.4 Summary

This chapter introduced some fundamental terms used in graph drawing, which will be used in the following chapters. There are special data structures like *networks* or *trees*. *Graphs* consist of *nodes* and *edges* which model the underlying information structure. Graphs can be *directed*, *simple*, *connected*, and *planar*. Directed graphs can be *acyclic* or *st-graphs*.

A *graph's drawing* is a diagrammatic representation in the plane. In connection with graph drawings, the terms *planar embedding* and *visibility representation* have been stated. Algorithms for graph drawing utilise certain parameters which are grouped into *drawing conventions*, *aesthetics*, and *constraints*. Considerations for *efficiency* are also of importance.

For further reading refer to [dBETT94, GT98, Wei98, dBETT99, Sha98] or one of the various other books in the field of graph drawing. The following sources are available through the Internet.

Good introductions to graph drawing can be found in [HMM01, M$^+$01], introductions to graph theory are presented for example in [Sas01, Cha01, Man01]. For tutorials and courses for graph drawing, graph theory, and information visualisation see [CT01, Cal01c, Cal01b]. The following glossaries are available on the Internet: [NIS01, Che01, TS01] for graph drawing and [Cal01a] for graph theory.

Having learned about the basics of graph drawing, the next chapter presents a collection of particular approaches and algorithms based on this theory.

Chapter 3

Graph Drawing Techniques

Contents

This chapter continues the considerations of the previous chapter. Known algorithms for graph drawing and visualisation will be presented and discussed. Graphs leave great freedom for arbitrary drawings. This is why a huge variety of very different algorithms has evolved. It is reasonable to gather these algorithms into distinct groups which share the same approach. Good overviews are also presented in [dBETT99] and [dBETT94].

Sections 3.1 to 3.3 present selected visualisation approaches with prominent visualisation algorithms. Section 3.4 discusses extensions to general graph drawing algorithms for information visualisation.

3.1 Hierarchical Visualisation

The hierarchical approach draws acyclic digraphs as layered downward (or upward) diagrams. The edges are drawn according to the polyline convention. This approach is generally split into four steps:

1. Layering.
   From the given acyclic digraph a *layered (acyclic) digraph* is produced. This is done by introducing a finite number of layers. Each node is then assigned to a layer obeying the convention

of downward drawing: edges only point from higher layers to lower ones. In a sub-step a *proper layered digraph* is generated. This means dummy vertices are inserted along those edges which span more than one layer. At the end of this layer assignment step, all edges reach down exactly one layer.

2. Crossing reduction.
   Up to this step only the ordering of the nodes concerning layers is defined. The nodes' order within a layer is determined in this step. The name "crossing reduction" already tells what purpose this part of the algorithm has. The nodes are placed to obtain a minimum of edge crossings.

3. X-coordinate assignment.
   For the final layout of the diagram the layers are taken as integer y-coordinates. For the nodes on the respective layers the order is already fixed, but this step may try to skilfully move nodes within layers a bit without changing the previously generated orders. This can also result in fewer bends in the final step.

4. Cleanup and drawing.
   Before actual drawing, the previously introduced dummy vertices are removed again. If other changes have been introduced in the first step to achieve necessary properties for this layout process, the obtained layout has to be modified accordingly, so that the final drawing shows the original graph without the temporary changes. Then the diagram can be drawn. Some authors include this step in step three, or avoid it by requiring an acyclic graph as input.

The demands for an acyclic digraph can also be met by some preprocessing. Cycles in the graph can be removed by reversing the direction of a minimum of edges. Of course, in the resulting final diagram the edges have to be corrected again. However, they then violate the convention for the drawing's downwardness. Furthermore, undirected graphs can be supplied with artificial directions for the edges. This is an additional way of influencing the automatically generated layout.

Optimal solutions for steps 2 and 3 are NP-hard. Algorithms following this approach have to deploy heuristics for dealing with large graphs. Otherwise they would only be applicable for very small graphs, because of the exploding computation time. For detailed discussion of applicable algorithms, please refer to Chapter 4.

The first proposal for a hierarchical approach was given in [STT81], and is discussed in the following section.

3.1.1 Sugiyama's Algorithm

Sugiyama and his co-authors were the first to present concrete implementation guidelines following the hierarchical visualisation approach in [STT81]. For an exemplary diagram drawn by the algorithm see Figure 3.1. A list of "readability criteria" defines the aims of the visualisation process:

A. Hierarchical layout of the vertices.
   Builds the most basic regularity of the drawing. While this point deals with the placement of the nodes, the following elements care about the tracability of paths.

B. Less crossings of edges.
   The crossing of edges is the major distracting factor when trying to understand a structure or following a path.

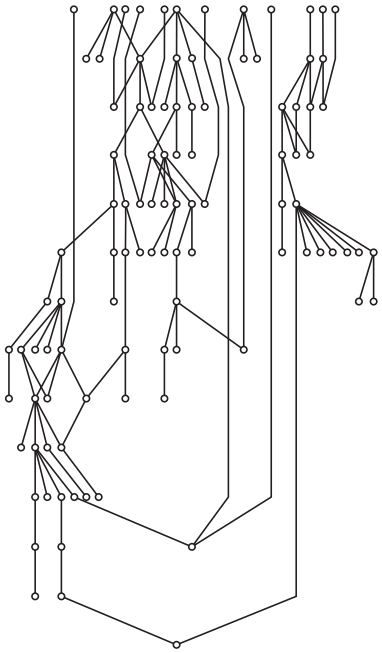


Figure 3.1: Example of Sugiyama's algorithm [STT81].

C. Straightness of edges.
   The less bends, the faster paths can be traced.

D. Close layout of connected vertices.
   It is desirable that paths are short.

E. Balanced layout of edges coming into and going from a vertex.
   This means that the structural information on branching and joining of paths is drawn clearly.

Two different strategies are proposed: a theoretical (or deterministic) method and a heuristic method. For both methods algorithms are shown. The theoretical method helps to analyse and explain the desired results, but for performance reasons a heuristic solution is presented. This makes it possible to deal with much larger graphs in reasonable computation time. Again, the visualisation process is subdivided into four steps:

1. Layering.
   A proper layered digraph is created. Cycles are removed by condensation which leads to a multilevel digraph. Long span edges are removed by inserting dummy nodes. All alterations are reversed again in the last step. For details see Section 4.2.1.

2. Crossing reduction.
   Due to the combinatorial nature of this problem, algorithms for only two levels are presented. Then the extension from a 2-level crossing reduction algorithm to cases for n-level hierarchies is shown. The given algorithms are discussed in detail later in this thesis: "Penalty Minimisation Method" (Section 4.3.1), "Barycentric Method" (Section 4.3.2), and the "Layer-by-Layer Sweep" (Section 4.3.3).

3. X-coordinate assignment.
   As already stated above, this step tries to avoid bends and to show symmetries. The only

allowed alteration to achieve this is moving nodes within their level of the hierarchy. The order of the nodes must not be changed, either. For a description of the proposed algorithms see the deterministic “Quadratic Programming Layout Method” (Section 4.4.1) and the heuristic “Priority Layout Method” (Section 4.4.2).

4. Cleanup and drawing.
   The alterations introduced in the first step are reversed and the graph is drawn automatically.

Finally, the paper presents empirical and statistical results. They show estimations for the calculation time and expectable quality of drawn diagrams. Also the large performance difference between the deterministic and the heuristic algorithms—in terms of running time—is confirmed.

Improved implementations of Sugiyama’s algorithm, the so-called Sugiyama-style algorithms, are presented later in this thesis: GRAB (Section 5.2.3), DAG (Section 5.1.1). For details about original and improved algorithms, see Chapter 4.

3.1.2 Method by Eades and Sugiyama

The following paper[ES90] continues the considerations of Sugiyama’s algorithm for drawing directed graphs. An ordered list of aesthetic criteria for the drawing’s understandability is stated. This sequence corresponds to the four steps of the algorithm. Each step tries to improve only the respective criterion. Other aspects of the layout are left free. References to descriptions for the proposed algorithms are included in the following list.

1. Arcs pointing upward should be avoided.
   - greedy cycle removal algorithm (Section 4.1.1)
   - divide-and-conquer cycle removal algorithm (Section 4.1.2)
2. Nodes should be distributed evenly over the page.
   - Coffman-Graham algorithm (Section 4.2.2)
3. There should be as few crossings as possible.
   - layer-by-layer sweep (Section 4.3.3)
   - Tutte algorithm (Section 4.3.3)
4. Arcs should be as straight as possible.

The authors state that their visualisation algorithm can draw graphs more readable than the original Sugiyama algorithm. The main reason for this is the better crossing minimisation in step 3. Additionally, the first two steps are designed to produce a layout in a way that crossing minimisation can be more successful. Unnecessarily long edges are avoided, too. A detailed description of these algorithms is contained in Chapter 4.

3.1.3 Drawing Compound Digraphs

A new feature to hierarchical visualisation of graphs is added by [SM91]. The paper considers “compound digraphs” with inclusion edges and adjacency edges. Inclusion and adjacency can be viewed as distinct digraphs on the same set of nodes. The inclusion structure is restricted to form a tree. A second restriction is given for the adjacency structure. Adjacency relations may “... not exist among ancestors and descendants in the inclusion tree” [SM91]. In the drawing (or “map”) the inclusion is depicted by a rectangle (or oval shape) including its subtrees. Nodes are also boxes or ovals. The adjacency edges are drawn in the polyline style or as curved lines. Figure 3.2 shows a generated layout for a compound digraph.

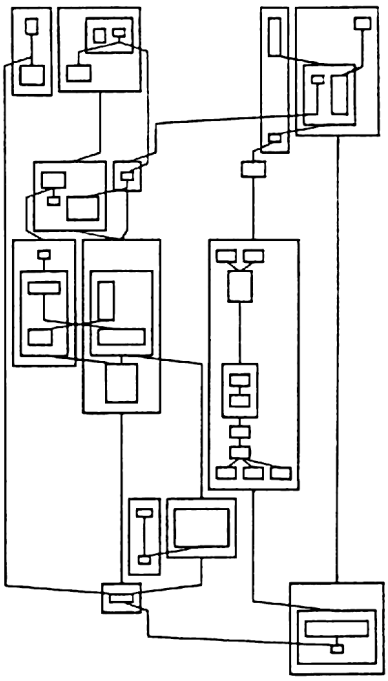


Figure 3.2: Visualisation of a compound digraph [SM91].

The visualisation algorithm conforms mostly to Sugiyama's hierarchical layout algorithm. Levels are numbered with sequences of integers to express the inclusion hierarchy, in the same way as sections in a document. A cycle removal algorithm is included in the hierarchisation step. Further supported features are: grouping, labelling, and an expand/collapse facility. The latter allows hiding detail by replacing an inclusion subgraph by a single node.

3.2 Force-Directed Visualisation

Force-directed algorithms are intuitive methods for creating straight-line drawings of undirected graphs [dBETT99]. A similar term in this context is force-directed placement. The central idea is to introduce a physical model for the graph. Having a closer look at this approach reveals its two parts.

1. Force model.
   The graph is translated to a physical model often consisting of springs, masses, and various fields. This is done with the aim of determining a stable state for the system between infinite expansion and infinite compression.

2. Minimum energy simulation.
   The prepared physical system is then put into a simulation process. The minimum energy configuration is of interest, since this is the most stable state of the system. Usually this state is linked with a reasonable amount of regularity.

In part 1 the algorithm replaces nodes and edges with physical representatives and forces are defined to interact with the components. Parameters of this model can be used to influence the graphic result. Techniques for part 2 are usually the product of numerical analysis rather than combinatorial algorithms [dBETT99].

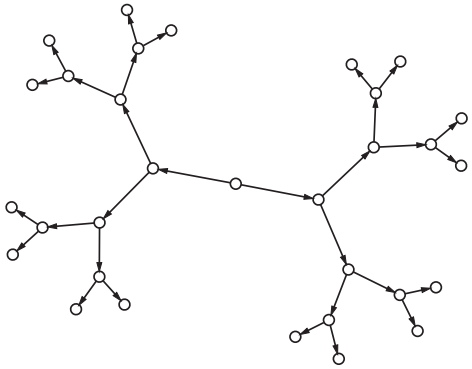


Figure 3.3: Final layout for a hierarchy drawn by the magnetic spring model [SM95].

The force-directed approach has several advantages. Algorithms can be implemented in a rather simple way. Nevertheless the produced diagrams are usually very nice and show good symmetry. Also an even distribution of the graph is obtained. One disadvantage, on the other hand, is that for large graphs part 2 may take considerable amounts of time.

Many variants of force-directed algorithms have been published. In the following sections, some applications and variations of this approach are discussed.

3.2.1 The Spring Model

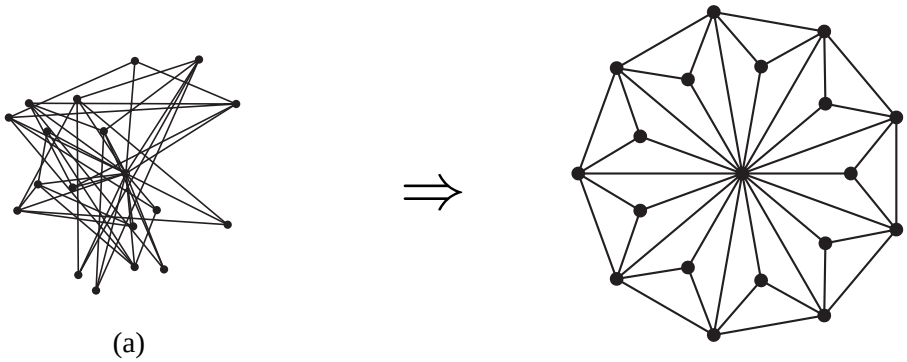
One of the early works on the force-directed visualisation (or placement) approach is [Ead84]. The proposed force model consists of rings, springs, and additional forces. Rings replace the nodes and springs the edges of the graph. The additional forces are attractive or repulsive forces between the rings. The minimum energy constellation is found as stated above (Section 3.2, step 2). Starting with an initial layout, the positions of the graph's elements are calculated iteratively according to the applied forces.

3.2.2 The Magnetic Spring Model

In [SM95] Sugiyama gives an extended model with magnetic springs and magnetic fields. A layout example can be viewed in Figure 3.3. This enables the algorithm to also handle directed or mixed graphs. The difference to Section 3.2.1 is that the springs may be magnetised. Adding this, directed edges can be modelled as directed magnetic elements that produce rotating forces when exposed to a magnetic field. Different magnetic fields are suggested: parallel, polar, concentric, and also compound fields.

3.2.3 Simulated Annealing

Simulated annealing [Coh97] goes a slightly different way to find a minimum energy situation. A problem of simple iterative algorithms is that a local minimum energy situation might be found rather than a global minimum. Simulated annealing tries to avoid those results by also allowing "uphill" moves. These are iteration steps which lead temporarily to higher energy potentials. By taking such iteration steps the algorithm hopes to get away from local minima to the one global minimum. The algorithm's name comes from the annealing process when a liquid cools down and crystallises. Figure 3.4 gives an example layout result.



(a) (b)

Figure 3.4: Example graph layout by simulated annealing.
Initial placement (a) and final layout (b) [Coh97].

3.2.4 Barnes-Hut Algorithm

The Barnes-Hut algorithm is a force-calculation algorithm [BH86]. It is used for calculation of forces within a large set of interacting objects (often referred to as the "N-body problem"). A characteristic of this algorithm is its adjustable precision. This gives the user the possibility to trade accuracy for run time. For very large sets of objects or for visualisation purposes this feature is highly desirable. In practice the algorithm is widely used in astrophysics, for example, and has been thoroughly parallelised. However, it can be applied as part of a force-directed visualisation algorithm as well. After devising an appropriate force model with a finite stable state, the Barnes-Hut algorithm can direct the minimum energy simulation.

The basic idea is to divide the object containing space into a quad-tree (or oct-tree for 3D calculation). For a quad-tree the viewed area is quartered in each refining step until each cell contains at most one object. The hierarchy of this refinement is the mentioned tree. Each of this tree's nodes—representing a certain area—stores the centre of mass and total mass of its contained objects. When the force, acting on one particular object has to be calculated, the algorithm can decide how deep to step down in the hierarchy. Depending on a given parameter a critical radius is calculated. This radius is proportional to the size of the considered sub-area in the quad-tree. If the distance between the particular object and the centre of mass of a sub-area is less than the critical radius, then the sub-area has to be refined to its child areas. Otherwise it is sufficient to use the whole area's calculated centre of mass. More detailed descriptions and program code is available on the Internet in great variety, for example [Sch94a, Wal98].

3.3 Other Geometric Visualisation Approaches

This section gives an overview of other known visualisation approaches. These approaches are general enough to allow major variations and additional aesthetics and constraints to meet individual demands.

3.3.1 Topology-Shape-Metrics Approach

The topology-shape-metrics approach constructs orthogonal grid drawings from undirected graphs. This approach allows homogeneous treatment of a wide range of aesthetics and constraints. Again,

the construction is divided into three consecutive steps. Each step takes the output from the previous one and adds some more restrictions.

1. Planarisation.
   The planarisation of the given graph produces a planar embedding (see Section 2.1.3). This step defines the *topology* of the graph's final representation. The topology of the drawing deals with the resulting areas between the edges and those area's relative order.

2. Orthogonalisation.
   This step takes the topology of the drawing as input. Now the orthogonalisation step defines the directions and bends of all edges. This introduces the *shape* for the drawing.

3. Compaction.
   The compaction step completes the drawing by applying the *metrics*. Here, in the last step, the actual positions of nodes and edge crossings are determined.

The Topology-Shape-Metrics Approach minimises the following aesthetic criteria in the given order: edge crossings, edge bends, drawing area. Other constraints and aesthetics can be introduced easily by extending the most appropriate of the three steps. For further reading about this graph drawing approach see [TdBB88].

3.3.2 Augmentation Approach

The idea behind the augmentation approach is to add edges for stronger connectivity. If enough edges are inserted, a maximal planar graph is obtained. This means that the resulting graph only contains triangular faces. Now one of the various algorithms for drawing triangulated structures can be applied. The algorithm, again, can be divided into three steps.

1. Planarisation.
   The same as in the previous algorithms. Dummy nodes replace edge crossings if the graph cannot be drawn planar.

2. Augmentation.
   In this step edges are added. The aim is to obtain a maximal planar graph. This means a planar graph consisting of only faces with three edges. The new edges are positioned in a way that the maximum number of edges incident to one node is minimised. This gives better conditions for the next step. For more flexibility it is also possible to add nodes, too.

3. Triangulation.
   The drawing is executed by representing each face as a triangle. For this step several applicable algorithms are known from graph theory. Finally the additional edges (and nodes) are removed again.

In general the result is a polyline drawing. Only if no dummy nodes have been inserted in the first two steps, all edges are straight lines. There also exist variations of this algorithm where fewer edges are inserted. In this case, the output from the augmentation step is not a maximal planar graph but a planar graph with certain connectivity.

3.3.3 Divide and Conquer Approach

The divide and conquer approach is widely known and used in algorithm design. It is very easy to understand how this approach works also for graph drawing. Basically, the graph is split into subgraphs which are drawn recursively. The resulting sub-drawings are combined to the entire drawing.

Implementation of the divide and conquer approach is simple, especially for trees or rooted trees. Therefore it is widely adopted. An algorithmic solution for a rooted binary tree would work like this:

1. Layer assignment.
   Like in the hierarchical approach (Section 3.1), the nodes are assigned to layers. A simple way to do this is to count the distance from the root.

2. Divide and conquer step.
   In the recursive part of the algorithm the following cases have to be handled.

   - If the subtree that should be drawn in one recursion is *empty* then do nothing.
   - If the subtree consists of a node with *no children*, then draw it trivially.
   - Else draw the two subtrees of the given subtree's root recursively. Place the sub-drawings at distance 2 from each other and place the root halfway between the roots of the subtrees.

This algorithm produces a straight-line grid drawing. It shows symmetries and gives isomorphic subtrees the same drawings.

3.4 Graph Decoration

Having heard about graph drawing approaches and visualisation algorithms, now the step to information visualisation will be taken. In Chapter 2 the elements of graphs and their drawings were discussed. Now these definitions have to be extended to new demands.

Information visualisation seeks to communicate as much information as possible. The name already tells that this is done visually, because human beings are very receptive of optical stimuli. Nodes do not convey much information if they all look the same. This is true for edges as well. The amount of information in a diagram can be increased greatly by adding textual and graphic components. Examples for additional information in a graph's drawing are:

- nodes and/or edges are labelled
- expressive symbols are used as nodes
- nodes are shapes filled with view of contents
- types of edges may be distinguished by different drawing styles
  (full/broken, fat/thin, curved/straight, decorated with arrows, etc.)

A broad overview of information visualisation efforts taken in this direction can be obtained from [Shn96, CMS98, Shn97, Fur98, CEG98, APW96]. Fairchild [Fai93] proposes a vector with different levels of detail for nodes. The visualisation application—or the user—can then decide which nodes are of interest. All other nodes are then displayed with less detail. This technique is an example of a distortion technique. Fairchild also gives a survey of several information visualisation applications.

The insertion of more information makes it necessary to enlarge the nodes. Since most graph drawing algorithms are designed using nodes of neglectable size, adaptions have to be introduced. Strategies and concepts to solve this problem are considered in [LE98].

3.5 Summary

Several graph drawing approaches have been presented in this chapter. The *hierarchical visualisation approach* and the *force-directed approach* were considered in more detail. Also, some specific information visualisation issues were covered. The different graph visualisation approaches discussed in this chapter mostly deal with two dimensional space, but virtually all of them can be extended to work in higher dimensional spaces. However, this is beyond the scope of this thesis.

Chapter 4

The Hierarchical Graph Visualisation Approach

Contents

After giving an overview of the currently used visualisation approaches, a closer look is taken at the hierarchical visualisation approach. Visualisation algorithms following this approach are often called "Sugiyama-style algorithms". This is because of the first proposal of such an algorithm by Kozo Sugiyama and others in [STT81] (see Section 3.1.1).

Since then a flood of improvements and alterations of this algorithm has been published. This chapter therefore tries to point out some of the most interesting variations to the parts of Sugiyama's algorithm. So this chapter presents both the originally proposed algorithms and improvements suggested by other authors. This chapter is subdivided into the hierarchical approach's four steps. Cycle removal is included as additional preprocessing step.

0. preprocessing:
   (a) cycle removal (Section 4.1)
1. proper hierarchy (Section 4.2)
2. crossing reduction (Section 4.3)
3. x-coordinate assignment (Section 4.4)
4. post-processing and drawing (Section 4.5)

The problems of steps two and three are known to be NP-hard. This means that the computation time grows rapidly as the graphs become larger. This is especially true for deterministic algorithms, which provide the one and only ideal result after execution. For NP-hard problems the best known strategy is to examine all possible solutions to find the desired one. This explains why deterministic algorithms might work on small sets of data, but dealing with larger structures is infeasible because of calculation time.

One way out of this situation is to lower demands. The nature of heuristic algorithms is to find a good solution without asking for the optimal one. An expansion from a single solution to a set of acceptable solutions is the result. Besides the increased probability of picking an acceptable solution out of the set of possible solutions, new algorithms can be applied. They use so-called "heuristics" which are able to find an arbitrary acceptable solution. Such heuristics can be implemented with much lower upper bounds of computation time than their deterministic counterparts.

According to [STT81] steps one and four are "self-explanatory". Nevertheless, some aspects will be considered here.

4.1 Cycle Removal

Limitations are always subject to work-around efforts. One of those efforts tries to overcome the restriction that only acyclic graphs can be drawn by the hierarchical approach. Sugiyama gave a first, more or less primitive, solution by replacing cycles by one node. Of course, this heavily influences the user's perception of the graph.

Other solutions trade the drawing convention of downwardness for greater flexibility. Only as many edges as possible should be drawn downward. This new degree of freedom can be made use of by reversing edges temporarily. By edge inversions the graph's cycles can be removed. When restoring the direction in the last step of the layout procedure, each inversion introduces an upward pointing edge. As few edges as possible should be reversed. The algorithms discussed in this section address how to pick the edges to be reversed.

Unfortunately, the problem of finding a cyclic graph's feedback arc set (see Section 2.1.2) with minimum cardinality is NP-hard and therefore not computable efficiently. There exist fast trivial algorithms, like for example the depth-first search, but they yield poor results. Better (heuristic) algorithms are shown in this section.

Extending the idea of temporary alterations, the applicability of a visualisation algorithm can be expanded, for example:

- *Undirected graphs* can be drawn by giving artificial directions to the edges. The orientations directly influence the layout of the obtained diagram.
- One way of applying directions to undirected graphs is creating a *rooted tree*. This is done by simply choosing a root out of the nodes. All edges are then oriented away from (or towards) the root.

It is clear that all the temporary changes have to be reversed later to restore the original graph. This is done in the final step of this approach (see Section 4.5).

4.1.1 The Greedy Cycle Removal Algorithm

This, also rather simple, greedy algorithm is mentioned in [ES90]. It is "... based on the intuition that nodes of large out-degree should appear near the top of the drawing" [ES90]. The nodes are sorted with respect to their out-degree. The node with the most outgoing edges gets the first position. Then a feedback arc set can be determined by finding all the edges going back in the nodes' order. The authors state that this algorithm can be implemented consuming $\mathcal{O}(|V| + |E|)$ run time, where $|V|$ denotes the number of vertices and $|E|$ the number of edges (or arcs). In practice its run time is competitive with the trivial search algorithm but provides better results.

4.1.2 The Divide-and-Conquer Cycle Removal Algorithm

Similar to the greedy approach, this recursive algorithm also deals with the out-degree, and is also stated in [ES90]. Again, the nodes are sorted with respect to their out-degree or, in other words, labelled with integer values from 1 to $|V|$. The recursion is defined as follows:

1. If the given graph has no edges, then assign labels arbitrarily.
2. If the number of vertices is odd, then the node with the highest out-degree is called graph G_1. All remaining nodes and all edges connecting them are called graph G_2. Recurse on G_1 and G_2 providing the appropriate range of labels.
3. If the number of vertices is even, then partition the nodes into two sets of equal cardinality, G_1 and G_2. All out-degrees of the first set must be greater or equal to all out-degrees of the second set. Recurse on G_1 and G_2 providing the appropriate range of labels.
4. Finally, those edges pointing from a lower to a higher level belong to the resulting feedback arc set.

The authors claim that this algorithm can be implemented with time complexity of:

$$\mathcal{O}\left(\min\left\{|V|^2,\ (|V| + |E|)\log|V|\right\}\right)$$

Thus for sparse graphs (where $|E|$ is $\mathcal{O}(|V|)$) it is slower than the greedy algorithm 4.1.1. In practice it takes up to four times as long on graphs with up to 400 nodes. However, tests have shown that it regularly obtains results with 20% fewer up-edges. [ES90]

4.2 Layering Algorithms

Layer assignment is a necessary preprocessing step for the hierarchical approach. Good layering can avoid unnecessarily long edges and thus save computation time. In addition to assigning each node to a layer, the input graph is also made "proper", which means that long edges are broken by additional nodes. The aim is to achieve only edges between adjacent layers.

4.2.1 Layer Assignment Method by Sugiyama

Sugiyama's layer assignment method proceeds as follows:

1. In [STT81, page 111] the input data is specified as "... a given set of directed pairwise relations among elements of a system." In other words: the expected structure for input is a directed graph. Regarding cycles the paper notes, "If the digraph ... has cycles, it is condensed" [STT81]. So Sugiyama's "cycle removal algorithm" simply replaces every cycle by a node. For more sophisticated solutions see Section 4.1.

2. The root or the roots of the hierarchy are put to the highest level. The roots are characterised by having no incoming edges.

3. Then they are removed so that new roots result. This procedure is iteratively executed until all nodes are assigned to levels. The number of resulting levels is equal to the maximal path length form a root to a leaf. This ensures that the drawing convention, forcing all edges to point down, is obeyed. It is easy to show that this strategy only works for acyclic digraphs.

4. In general the hierarchy is not proper yet. If edges reach farther than between adjacent layers, virtual or dummy nodes are temporarily inserted. Each edge gets a new dummy node at each layer it crosses.

Alternatively, the positioning can also be run from the lowest layer up. Starting at the top results in highest possible levels for the nodes. Starting at the bottom puts all nodes to their lowest possible position.

A different approach for the same result is the *longest-path layering algorithm* (see for example [ES90]). The nodes are assigned to layers in linear time. This is done by calculating the minimum length of a path to a sink. This length corresponds to the layer.

4.2.2 Coffman-Graham Algorithm

This algorithm was originally developed as a heuristic for the multiprocessor scheduling problem by E. Coffman and R. Graham [CG72]. It is proposed as a solution to the layer assignment problem in [ES90]. The Coffman-Graham algorithm fits the diagram on a drawing area with predefined width.

Basically, it works as follows. The nodes are labelled with integer numbers. These numbers serve as a kind of priority when actually placing the nodes. Determining the labels, the nodes with the least incoming edges get the highest priority. The placement step simply fills each level sequentially starting from top level. If a level has received its maximum number of nodes, the algorithm continues with the level below. For each placement the node with the highest priority is taken, but all its ancestors must have been placed before.

The Coffman-Graham algorithm does not work on graphs with transitive edges or cycles. This means that edges which bypass other edges, and generally other paths in the graph, have to be removed in a preprocessing step.

4.3 Crossing Reduction Algorithms

As stated in Section 3.1.1, Sugiyama's algorithm seeks to avoid edge crossings. Crossing lines make paths difficult to trace and this makes a drawing difficult to understand. Again, a deterministic and a heuristic solution are proposed. Both algorithms are capable of reducing the edge crossings between

only two layers where one layer's node order is fixed. Therefore an additional strategy (Section 4.3.3) is given for extending the algorithms to handle multiple layers.

Furthermore, several improved algorithms for this problem are shown. Section 4.3.6 actually presents a fast algorithm for counting crossings. It can be incorporated in crossing minimisation algorithms to improve computation time.

4.3.1 Penalty Minimisation Method (PM)

The PM method is a deterministic 2-level crossing reduction algorithm proposed in [STT81]. The suggested strategy to expand this algorithm for n-level hierarchies, the layer-by-layer sweep, is shown in Section 4.3.3. Basically the PM method considers the increase or decrease of crossings for the swap of pairs of nodes. If minimising swaps exclude themselves mutually, the swap with the least "cost" is dropped. For determining the swaps that should be dropped the so-called *penalty digraph* is deployed. The calculation can be sketched as follows:

1. The penalty digraph is built. It shares the nodes of the given graph. The edges represent a swap increasing the number of crossings. Edge labels show the amount of the increase.

2. All the strongly connected components are found. That means all sets of nodes which are circularly connected by edges. Strongly connected components can only have three or more nodes. This follows from the definition of the penalty graph.

   If no strongly connected components are found, the penalty digraph is cycle-free. The optimising swaps do not conflict. The algorithm can continue with the final step 4. If there are conflicts, they have to be resolved first:

3. Each strongly connected component has to made cycle-free by inverting edges. The edges to be inverted are determined skilfully to minimise the number of crossings. Each inversion introduces crossings according to the respective edge's label.

4. The constraints for the solution can be obtained by following the penalty graph's edges. Missing edges mark do not care order. The graph could theoretically be completed by inserting undirected edges between not connected nodes labelled with zero. All paths visiting all nodes without violating edge directions are acceptable solutions. All such solutions are equally good with respect to the number of crossings.

4.3.2 Barycentric Method (BC)

BC method is the heuristic algorithm for crossing reduction described in [STT81]. Like the PM method it is a 2-level algorithm and can be extended by the algorithm described in Section 4.3.3. The reordering follows the barycentres[1] which average the position of the downward (or upward) neighbours. Downward or upward depends on whether the upper or the lower level should be changed. The nodes are ordered from the smallest to the largest barycentre. If two barycentres are equal the initial order remains unchanged.

Additionally, a second phase is recommended due to good empirical results. The overall strategy is outlined in the following.

[1] A node's barycentre, which is a synonym for centre of mass, denotes the average position of that node's adjacent nodes. The barycentre can be calculated with respect to both or only one adjacent layer.

Phase 1: Both layers are improved consecutively by sorting their nodes from smallest to biggest barycentre. Nodes with equal barycentres keep their mutual order.

1. The upper level is reordered.
2. The lower level is reordered.
3. Phase 1 is repeated if the previous iteration introduced changes to the layer's orders. Otherwise, or also if the number of iterations in phase 1 attains a predefined maximum, phase 2 is entered.

Phase 2: Both layers are consecutively changed by rearranging nodes with equal barycentres.

4. Reorder upper level's nodes.
5. Reorder lower level's nodes.
6. If both of these actions do not destroy the barycentric order of the respective other layer then terminate the calculation.
7. If the number of iterations in phase 2 attains a predefined number, then terminate calculations. Otherwise re-enter phase 1.

The maximum number of iterations, for the case that not all crossings can be removed, is set to 4 in [RDM+87]. Unless "... the previous pass made a dramatic reduction in the number of edge crossings, in which case an additional pass is made" [RDM+87], see also Section 5.2.3. In empirical tests for computation time the BC method performed significantly better than the PM method. Performance with respect to finding the minimum crossing layout was also satisfying.

A slight variation to the BC method is given in [GNV88] (see Section 5.1.1). Instead of barycentres "weighted medians" are used. For odd numbers of adjacent nodes on the lower (or upper) level the coordinate of the median is taken. If the number is even, then the weighted median lies between the left and the right median's position. The actual position between those nodes is biased "... toward the side where neighbour nodes are more tightly packed" [GNV88].

Besides that a specialised procedure for determining the initial order of the level's nodes is deployed. Phase one is executed using the weighted medians. Another change is introduced in phase two: Pairs of nodes adjacent on the same layer are transposed if this reduces the number of crossings. For doing this a heuristic was developed to find local optima.

4.3.3 Layer-by-Layer Sweep

In [STT81] this algorithm is presented for making the two 2-level crossing reduction algorithms applicable for large hierarchies. Generally speaking, this algorithm takes any algorithm for 2-level crossing reduction as a sub-algorithm. The only restriction is that one specified level must remain unchanged. This demand is necessary for applying the sub-algorithm iteratively forming a sweep.

The following mathematical formula describes the aim of the algorithm:

$$minimise\ f(z) = z_1 + z_2 + \cdots + z_{n-1}$$

where z_i represents the number of crossings introduced by the nodes on levels i and $i+1$ in their respective order. Now the algorithm tries to reduce the factors z_i one by one. Although they are not completely independent, because all but the first and the last level influence two of the z-terms. This difficulty can be gotten around by the following simple algorithm.

1. Arbitrary initial orders for each level's nodes are given.

2. Reduce the crossings between the first two levels by applying the 2-level sub-algorithm. The upper level is fixed. Only the lower level's node's order may be changed.
3. Repeat step 2 one level lower and so on. Go on repeating down through all levels.
4. Reduce the crossings like in step 2 for the lowest two levels, but now only the upper level's order may be changed.
5. Repeat step 4 one level higher and so on. Repeat stepping up through all levels.

This procedure is called a down-up procedure because of the consecutive sweeps first down and then up. The whole down-up procedure is iterated until one down-up procedure makes no change to the layout. In addition to this, a maximum number of iterations is given initially.

For the BC method, phase 1 is executed as described above. Phase 2 is started after the down-up procedure in phase 1 terminated.

A slight alteration is suggested in [RDM+87]. They state that, for some graphs, node positions were unstable. Thus they propose to use the average of the upper and lower barycentre after the first down-up procedure is completed.

In [ES90], the Tutte algorithm was applied as an "... old graph drawing method first used by Tutte ...". It is claimed to produce results similar to those of the barycentric method. This is achieved without a layer-by-layer sweep. Technically speaking, the Tutte algorithm iteratively solves a system of sparse linear equations. This process is very similar to the sweep strategy. The algorithm fixes the positions of top layer and bottom layer. Then equations for the nodes' x-positions are established. This is done in a way that each node is placed at the weighted average of its neighbours' x-coordinates. The positions are weighted by the considered node's in-degree and out-degree.

4.3.4 Using Sifting for k-Layer Straight-Line Crossing Minimisation

"... Sifting ... is a heuristic for dynamic reordering of decision diagrams used during logic synthesis and formal verification of logic circuits." [MSM99]

Sifting is proposed for crossing minimisation of, both one-sided 2-level graphs and k-level hierarchies in [MSM99]. The presented algorithm first picks one node and sifts it before picking another. For one-sided 2-level graphs, only the modifiable level's nodes are picked in a specified order. Suggested orders are from left to right, sorted by in-degree, and random order. If sifting one node takes $\mathcal{O}(n)$ the algorithm runs in $\mathcal{O}(n^2)$ for the entire graph.

For k-layer problems, a global order of all nodes is used by a method called "Global Sifting". Here the paper recommends sorting the nodes by in-degree. Furthermore, a strategy is given which regularly reverses the nodes' order in each iterative step. The order is also reversed if no improvement is possible by sifting one node. This event is called a fail. The number of fails is limited by a definable constant, exceeding this constant terminates the algorithm.

The actual sifting, both for the one-sided and for the k-level version, takes one movable node. All other nodes are fixed. The node is moved by swapping it with its neighbour on the same layer. By doing this, the node is moved to the right through the layer until it becomes the rightmost node. Then it is swapped to the left until it becomes the leftmost node. While doing this, the respective crossing numbers are computed. Thus, the minimum crossing number is known after the right-left procedure. Finally, the considered node is moved to the right again until it reaches the position with the local crossing minimum.

Experimental results have shown good performance. The algorithm outperformed layer-by-layer sweep algorithms, previously known for good performance, by far. This holds for layered graphs

with three layers or more. Visualisations of graphs with different sizes and densities were examined to obtain this result.

Improvements to this algorithm are given in [GSBM00]. The alterations mainly concern the sifting process. Implementation details to save computation time are presented. The authors suggest implementing a three-dimensional crossing matrix. This measure speeds up computation of the number of crossings in each step. When swapping two nodes, the data structure's respective columns have to be interchanged.

Secondly, lower bound sifting is used to reduce the number of iterations while sifting a node. This is achieved by calculating the lower bound for the number of crossings. While sifting, this lower bound for moving the node further is efficiently updated after each swap operation. If this value exceeds the current minimum number of crossings, further movement in the same direction cannot improve the result. Thus, the swap direction can be reversed immediately, omitting remaining steps to the end of the layer.

Experimental results showed that it is possible to reduce the run time by a factor of 20 or more, without a loss of quality.

4.3.5 Vertex-Exchange Graph

An interesting concept for multi-level crossing minimisation is presented in [HK99]. The authors construct a data structure to deal with crossings in proper layered digraphs. The nodes of this "vertex-exchange graph" are ordered pairs of the levelled graph's nodes which share the same layer. The vertex-exchange graph's nodes are connected if the corresponding original nodes are connected by two non-adjacent edges. Those connections are marked with '-' if the original edges cross, with '+' otherwise. The paper states that planarity can be detected by ensuring that the vertex-exchange graph contains no cycles with an odd number of '-' connections. Based on the given crossing counting algorithm, a crossing minimisation algorithm is proposed. The solution is designed as a minimisation problem using integer linear programming (ILP).

4.3.6 An $E \log E$ Line Crossing Count Algorithm

The authors of [WM99] propose a fast crossing count method for improvement of crossing minimisation algorithms. They state that the second step is the bottleneck of all Sugiyama-style layout algorithms. This solution should enable running a hierarchical layout algorithm in $\mathcal{O}(n \log n)$ time, hence handling graphs with tens of thousands of nodes or more should be possible on current computers.

The crossing calculation is done in a sweep for each adjacent pair of layers. All edges connecting these layers are divided into groups with respect to the current scan position. The crossings are calculated by those groups' cardinalities. Also alternate level crossings and same level crossings are handled separately. To calculate same level crossings efficiently, a data structure called an accumulator tree is used. The paper provides proofs for its statements and experimental results for calculation time.

4.4 X-Coordinate Assignment Algorithms

The purpose of this step is to make the drawing readable for the user. The previous step already provides a layout with few crossings, but for obtaining a human understandable diagram additional work is necessary. Features which help users to grasp the structure of the graph are symmetry, balance, and lines with few or no bends. Those features are taken into account when moving the levels' ordered

nodes to actual positions. Of course, the order of the nodes must remain unchanged. Special attention is paid to dummy nodes, because they introduce edge bends.

4.4.1 Quadratic Programming Layout Method (QP)

The QP method is the deterministic algorithm proposed in [STT81]. Mathematically the strategy is described by an objective function and a set of constraints. The *objective function* represents an optimisation problem. The calculation instruction is given by the formula:

$$minimise\ f = cf_1 + (1-c)f_2, \qquad 0 < c \leq 1$$

Or in words: The factors f_1 and f_2 are weighted by c. The value of c allows to adjust the valence between f_1 and f_2. Here f_1 is a measure for the closeness of connected nodes with respect to the x-coordinate. The factor f_2 measures the balance of the incoming and the outgoing edges incident to nodes in the layout.

The following *constraints* are applied: The x-position of the first node in the first level is predefined. And the difference between the x-coordinates of two consecutive nodes must be at least a given positive minimum distance. For dummy nodes an additional constraint says: If two dummy nodes are connected by an edge, they must have the same x-coordinate.

The final solution has to be the layout with the minimum value for f without violating the given constraints. As already stated, finding this solution is NP-hard. This algorithm is only feasible for rather small graphs. Thus, Sugiyama introduced the PR method for graphs of more reasonable size.

4.4.2 Priority Layout Method (PR)

The heuristic PR method was proposed in [STT81]. In contrast to the QP method, it generates a readable layout, not necessarily the optimal one. This more generous demand can bring the benefit of much shorter computation time.

The basic idea is to improve one level after another in a sequential way. By so-called “priority numbers” the changes in inter-level node placement are determined. The algorithm can be outlined as follows:

1. The nodes on each level are put to initial positions of equal distance according to their order.
2. Positions in each level are improved in three sweeps through levels: 2...n, n-1...1, t...n (where $2 \leq$ t $\leq$ n-1). The first and the last sweep are called down procedures, the second is the up procedure.
3. In each level node positions are determined according to their priority numbers. Dummy nodes have the highest priority. Priorities of the other nodes are the number of upward pointing (for down procedures) and downward pointing (for up procedures) edges.
4. In principle improvement is done by minimising the distance between the upper (lower) barycentre and the considered node under the following conditions:
   (a) Positions of all nodes are integers and distinct for all nodes at the same level.
   (b) The order of nodes within a level must not be changed.
   (c) On each level the positions of all contained nodes are improved starting from highest to lowest priority. If an improvement would violate one of the previous two conditions, this improvement is skipped.

4.4.3 A Fast Layout Algorithm for k-Level Graphs

Another algorithm for the coordinate assignment step is given in [BJL00]. The layout is constructed similarly to Sugiyama's original algorithm, but, additionally, the total length of short edges is minimised level-wise. The authors state a running-time of $\mathcal{O}(\overline{m}(\log \overline{m})^2)$ where $\overline{m}$ denotes the number of edge segments in the graph's drawing. All long edges' segments except the outermost ones are drawn vertically. A precondition is that long edges may not cross at inner segments. This can be ensured in a preprocessing step.

The algorithm consecutively executes in three steps. The first assigns x-coordinates to dummy nodes—also called virtual vertices. After that the original nodes are positioned. And, eventually, the y-coordinates of the levels are determined. Step 1 of this procedure generates the final x-coordinates for the dummy nodes as follows. Keeping each level's node permutation, the average between the leftmost and the rightmost position is stored for the dummy nodes x-coordinates. Leftmost and rightmost positions are determined obeying the minimum distance between nodes and the demand for vertical line segments between dummy nodes.

Original nodes are positioned in the second step. For each level, groups of original nodes are examined. They lie in between dummy nodes and/or the level's borders. First fixed sequences are determined by comparing the sequence's available space to the minimum node distance. Then the hierarchy is traversed down and up by a dynamic array calculation algorithm. Also a sophisticated method for finding the optimal positions is explained.

The final step introduces y-coordinates for the hierarchy's levels. The distances between two levels are determined by limiting the edges' gradients. This limitation has to be satisfied by the longest segment between each pair of levels.

4.5 Post-processing and Drawing

Before the actual drawing of the diagram, some post-processing usually has to be done. The temporary alterations introduced in step 1 have to be reversed again. Otherwise the drawn graph would not match the graph originally given as input. For Sugiyama-style algorithms dummy nodes have to be removed creating long edges with bends. Temporarily reversed edges must be reconstructed again. This introduces upward pointing edges. For other algorithms following this approach additional work might be necessary in this step.

For drawing, the coordinates determined in steps one to three are simply used to place the graphical representations of the nodes. The edges connect the positioned nodes as defined in the given graph structure. Also some fine-tuning may be introduced in this step. For example considering node sizes for avoiding overlaps or widening small intermediate spaces which makes edge layout difficult. This task is often not very simple. Especially if nodes have non-uniform sizes or shapes. If the selected edge representation needs additional work like path routing, this is also managed in this final step.

Chapter 5

Graph Drawing Applications and Packages

Contents

This chapter presents applications related to graph drawing and information visualisation. In contrast to previous chapters actual programs and systems will be presented. Of course, these examples build upon and benefit from the theory already covered.

Section 5.1 presents graph drawing programs which are designed to produce static visualisations of graphs. Such programs support no user interactivity. Section 5.2, in contrast, covers applications which provide dynamic interaction with the user. They are often called browsers. Graph editors also allow manipulating a given graph. Section 5.3 concentrates on systems which aim at providing particular information visually. Visualisation paradigms are adapted to maximise the visual bandwidth,

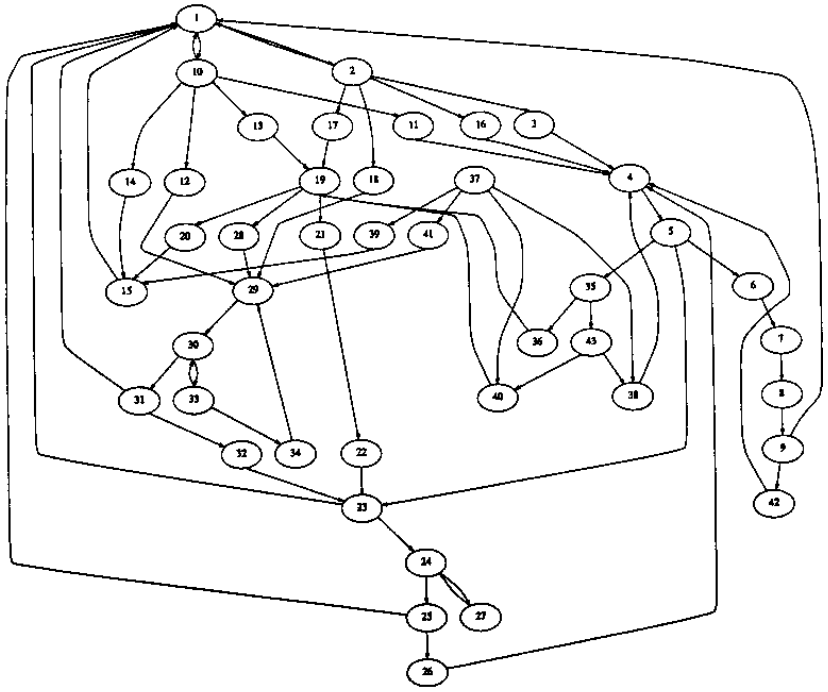


Figure 5.1: Example diagram from DAG (“dynamic world model” from J. W. Forrester) [GNV88]. We will call this graph JWF1.

often by dropping the traditional graph metaphor of objects and connecting lines. Finally, an overview of diverse current packages and libraries to support graph drawing is given in Section 5.4.

5.1 Batch Graph Drawing

The applications viewed in this section are “one-shot” or “batch” drawing programs. They are only partly designed to output to computer monitors and usually create files containing the computed drawing. Such drawings can be stored as a raster image or a vector graphics file. This strategy outsources viewing and navigational functionality to image viewer programs which generally support zooming and panning the viewed area.

Such static strategies are mostly favourable for very large graphs. Layout calculation may take a while, but the resulting image is preserved for further use. The same is true for complex graphs which have to be drawn optimally. The output files are also convenient for printing and for inserting into documents. Besides those applications listed below, many algorithmic implementations of the various graph drawing approaches discussed in Chapter 2 are available.

5.1.1 DAG—A Program that Draws Directed Graphs

DAG is a Sugiyama-style graph drawing program proposed in [GNV88]. For input, a list of nodes and edges is required. Optional drawing instructions—concerning node layout, labels, and spacings—can be given. Output is written to a PIC or PostScript file. A layout produced by DAG is shown in Figure 5.1. As can be seen from the figure, DAG creates layered diagrams with edges drawn curved as B-splines. Multiple edges are supported, and nodes and edges can be labelled. The paper describes the B-spline control points’ placement in detail. The input file is a text file following a specified concise syntax. Some layout commands may be applied to nodes. Edges can be modified by some

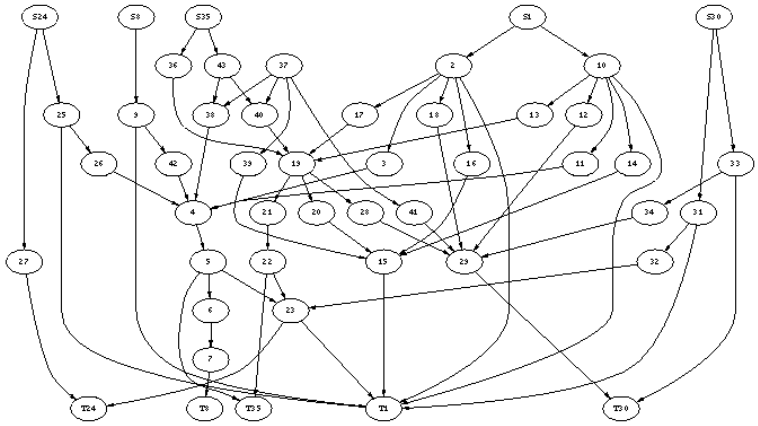


Figure 5.2: Dot drawing of the graph shown in Figure 5.1.

options, for example weights which influence the layout. Constraints can force selected nodes to appear at the same level or in sequential order.

Cyclic graphs are made cycle-free in an initial step by inverting conflicting edges. These are found by a depth-first search. The layering is done by a specially developed combinatorial algorithm in $\mathcal{O}(E)$ time using $\mathcal{O}(E)$ memory. For crossing minimisation the original algorithm with an enhanced weight function ("weighted median") is used (see Section 4.3.2).

5.1.2 Dot

Dot is the hierarchical graph drawing component of the Graphviz toolkit. For Graphviz see also Section 5.4.2 and [GV99, GD'01]. Dot implements a hierarchical algorithm for graph drawing. Edges are usually drawn as curves. The layout example in Figure 5.2 shows a representative example. Although similar to Figure 5.1, which shows the same graph, differences are obvious. This figure has been created using additional constraints to force selected nodes to appear in the same hierarchy level. Source nodes, marked with labels starting with 'S', share the top layer. The bottom layer is populated by the drain nodes. Their labels start with 'T', hinting at the method for drawing st-graphs.

Dot reads graph descriptions as an ASCII file which can also include constraints and layout commands. Different styles and labels for nodes and edges are supported, as well as clustered and nested graphs.

5.2 Graph Browsers and Editors

Nowadays, the expression "browser" is used every day for the programs for viewing the contents of web sites. A graph browser is similar to a web browser in some points: nodes (corresponding to individual web pages) can be viewed, edges (links between web pages) are shown and can be followed. However, web browsers usually show only one node at a time. Graph browsers try to give a good impression of the whole structure. If the structure is too large, than at least a portion of the environment around the part in focus is shown.

This section also takes a look at graph browsers with editing capabilities (Section 5.2.3 and Section 5.2.4). They allow interactive graph drawing: the user has the final decision as to where to put

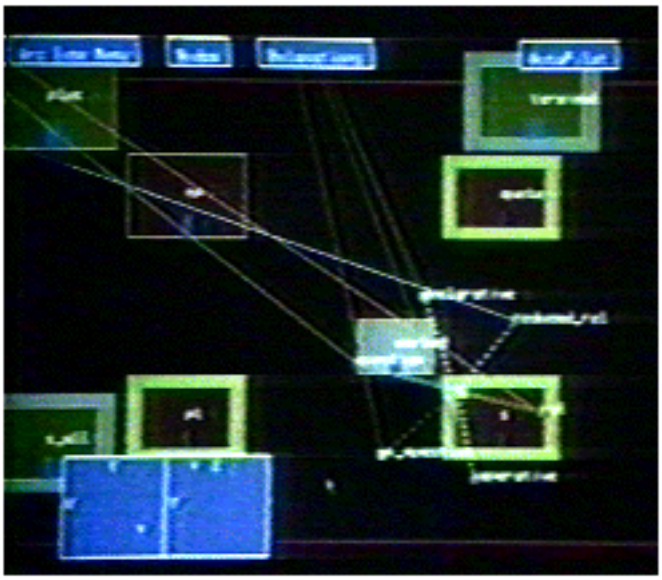


Figure 5.3: Screen shot from a SemNet knowledge base visualisation.

the graph's elements, but is assisted by the program which gives layout suggestions and provides powerful layout functions and tools. The term "graph manipulator" describes a program which enables the user to introduce changes to a given information structure. Such changes may include the alteration of labels or node contents, but also inserting and deleting nodes and edges. However, this added functionality also brings new problems, which are discussed in Section 5.3.

5.2.1 SemNet

SemNet was an exploratory research project undertaken to advance the understanding of problems facing both users and developers of large knowledge bases [Fai93].

The developers did not try to find an optimal visualisation for networks, they gave the user the possibility to easily modify the visualisation for individual demands. The visual presentation is based on a 3D space, where the entities are placed with respect to their properties. The user may navigate through this space to have a closer look at individual entities, or to obtain an overall view from a certain perspective. The result is a fish-eye view based on three-dimensional perspective. SemNet is suggested to be implemented using 3D graphics hardware to directly convey the 3D space metaphor. According to their momentary importance, the elements can be placed closer to or farther away from the user. The example in Figure 5.3 gives a rough impression.

The project's central objectives can be expressed by the following three hypotheses:

- A user must recognise the identities of individual elements,
- A user must recognise the relative position of an element within a hierarchical context, and
- A user must recognise explicit relationships between elements

Users of SemNet have a rich collection of navigation techniques for moving the viewpoint, techniques for experimenting with different methods and parameters for positioning the knowledge elements [Fai93].

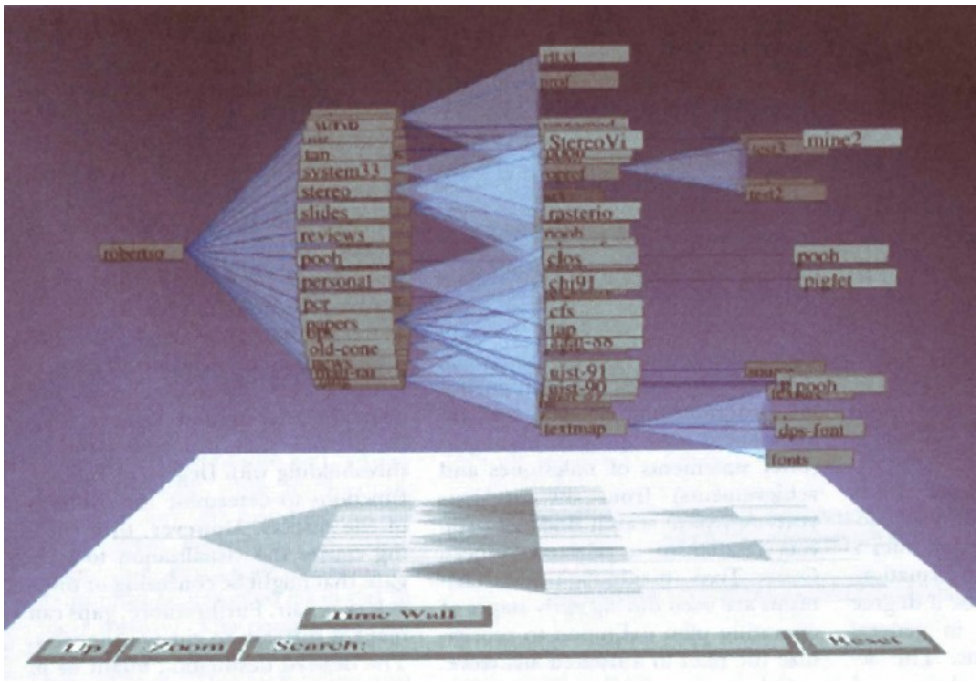


Figure 5.4: Cone Tree view of a library hierarchy.

5.2.2 Cone Trees

The Cone Tree visualisation paradigm [RMC91] was developed by Xerox PARC. It displays hierarchical information as navigable 3D cones. Each cone represents one parent in the tree (cone top) with all connected children. The children are arranged along the cone's base. The user can rotate these cones to view particular children more closely, or shift the visible cones to move the region of interest. Focused elements are rotated to the front automatically. All operations are shown as smooth real-time animations. The animations are timed to last one second which ensures efficient work without loosing context. The structure is scaled to fit on the screen. Besides that, it is possible to modify the visual appearance of the Cone Trees.

An example of a Cone Tree view is given in Figure 5.4. The three dimensional effect is reinforced by the white ground plane catching the shadow of the suspended cone construction. This plane also carries control elements for user interaction.

The strength of this approach is in giving users an understanding of inter-object complexity. It is also very useful in helping users to understand the connections between nodes and how nodes fit into a global context [Fai93].

5.2.3 GRAB—A Browser for Directed Graphs

In 1986 Rowe and his co-authors published a description of the graph browser GRAB [RDM$^+$87]. GRAB was designed as a general purpose browser for directed graphs with editing facilities. Figure 5.5 shows a screen shot of a visualisation layout by GRAB. Obviously the hierarchical visualisation approach is followed. This is also indicated by the requirement for a directed graph as input. The layout is automatically constructed. Additionally, the user has the possibility to introduce manual modifications interactively. The simple interface for browsing consists of a zooming facility to select the size of the visible part of the drawing area. The position of the actually displayed part can be controlled by scrollbars at the bottom and at the right of the graphical user interface.

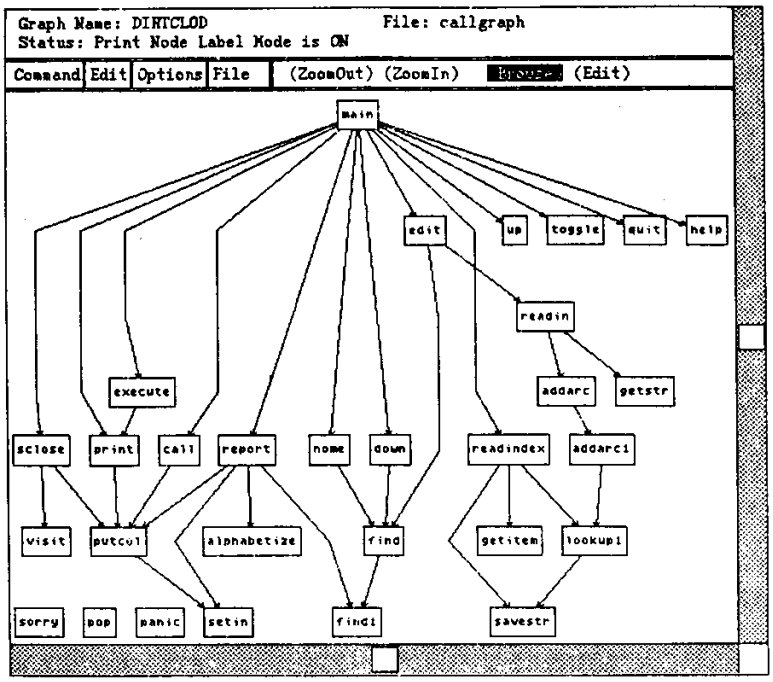


Figure 5.5: A call-graph displayed in a GRAB browser window [RDM+87].

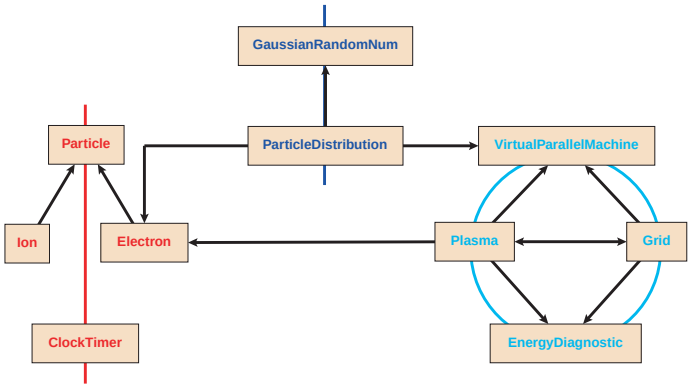


Figure 5.6: GLIDE example graph layout with applied constraints [RMS97].

GRAB is stated to use a variation of Sugiyama's algorithm (see Section 3.1.1). Modifications concern cycles in the given graph and layout improvements. One reason for choosing this algorithm out of several candidates was the moderate run time which results from the deployed heuristics. The authors were "... willing to sacrifice layout quality for run time performance" [RDM+87].

In contrast to Sugiyama's original algorithm cycles are not condensed but removed by temporarily inverting edges. This is done during the first step (building a proper layered digraph) every time a conflicting edge is detected. Fine-tuning with respect to the layout is done by making nodes variable sized. Level distances are increased until all edges have a gradient of at least $\frac{2}{3}$. Requiring this, almost horizontal edges are avoided.

5.2.4 GLIDE—Graph Layout Interactive Diagram Editor

The GLIDE system [RMS97, GD'01] is an interactive constraint-based editor for drawing small- and medium-sized graphs (50 nodes or fewer). It organises the interaction in a more collaborative

manner than in previous systems. GLIDE uses a physical simulation to satisfy the constraints under a mass spring model. Animating the simulation helps users to see what effect their actions have. The application permits user intervention at any time. [RMS97]

Several interactive layout mechanisms are provided. Constraints may be applied which partly restrict further manipulation. Automated functionalities for improving the layout include avoiding node-node and node-edge overlap, alignment, equal spacing, and showing symmetry. Nodes can be constrained to form sequences, clusters, T-shapes, zones, or hub shapes. In the example shown in Figure 5.6, the blue nodes are aligned vertically, the red part is laid out to show symmetry, and the aqua coloured nodes form a hub structure. Horizontal alignment and equal spacing between the different coloured sub-graphs have been constrained in earlier steps.

GLIDE is designed to help users to interactively create aesthetically pleasing diagrams more quickly. It is useful for small and medium-sized graphs. The user has to give rough positions for the node. After that, the user may apply rules for the visualisation. GLIDE then refines the rough node positions via physical simulation according to the given constraints. The simulation is run simultaneously with the user interaction.

5.3 Information Visualisation Applications

Algorithms for information visualisation try to improve the amount of information visualised in a diagram or interactive simulation. Approaches range from modified 2D graph representations, as already discussed, to abstract three-dimensional virtual worlds. A landscape metaphor's implementation is discussed in Section 5.3.5.

The central purpose of those efforts is making detail (focus) visible while preserving a global view (context). The usual problem is that large data spaces should fit on computer display screens with very limited area. Presenting detail and global structure in two different pictures is only a sub-optimal solution. Most approaches deal with different forms of distortion. The whole diagram space is distorted to show different elements in varying detail. Common distortion methods include the following:

- *Graphical Distortion.*
  The most popular is the *fish-eye view*. It simulates the effects of optical lenses to magnify parts of interest on an overview map. Unfortunately, physically correct implementations destroy the common rectangular coordinate system. So the presented pictures look somewhat messy. This problem can be overcome by the orthogonal fish-eye view [MELS95].
  An approach with distortion applied to only one dimension is presented in Section 5.3.1. A different approach, presented in the same visualisation package, is mentioned in Section 5.2.2.
- *Semantic Density Distortion.*
  This technique distinguishes between contents of more or less importance. The elements of interest are presented large and in detail, the very unimportant ones may even be omitted. An implementation is shown in Section 5.2.1. An application implementing this technique is presented in Section 5.3.2 (see also [Fai93]).
- *Statistical Abstraction.*
  Statistical abstraction—the calculation of averages, medians etc.—is a somewhat different kind of distortion working with numeral data. The same principle holds true: Detail is hidden while global properties are made visible. Methods for going into detail can also be provided. Section 5.3.3 discusses an example application.

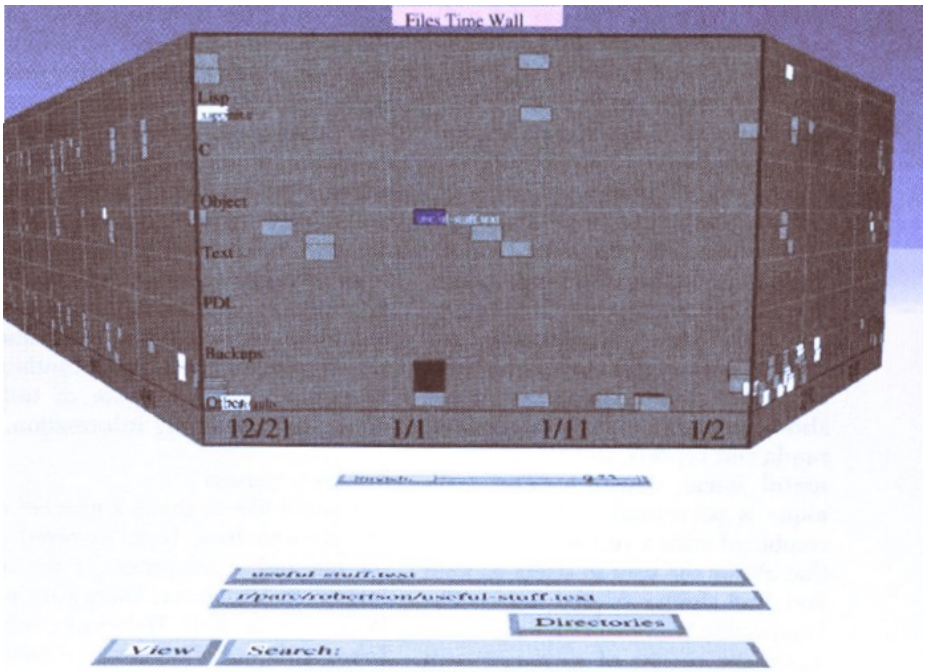


Figure 5.7: The Perspective Wall.

Another issue in information visualisation is preserving the user's "mental map" during steps of interaction. Section 5.3.4 discusses implementation guidelines.

5.3.1 Perspective Wall

The Perspective Wall is a visualisation metaphor for linear information, proposed in [MRC91]. The basic idea can be explained by viewing Figure 5.7. All objects to be visualised have their defined position on the horizontal rubber band. The band is often labelled with time, representing a calender. Different tracks help to distinguish the entries which additionally have properties like colour, length, and partly labels. The rectangular front part is shown in detail, whereas the two ends of the band are stretched back to the left and the right. This distorts the regions to either side of the main (front) part. So the central information can be viewed closely while keeping an overview of the entire band. Like Cone Trees (Section 5.2.2), the whole three-dimensional object is based on a white ground plane, upon which user interface elements are also placed.

5.3.2 Tree Browsing

The browser presented in [AC96] is designed to visualise tree structures. The authors state that trees are widely used in computer systems. On the other hand, the limitations forced by the tree definition makes drawing trees a much easier task than drawing general graphs. This advantage gains in importance, when applied to larger data sets. Strategies applied to cope with such large structures base on simple methods for increasing or decreasing level of detail in parts of the tree. Such parts may be leafs, sub-trees, or arbitrarily selected path segments.

The tree browser lets the user define the importance of all elements. A set of interaction techniques allows increasing and decreasing detail. Navigation in this case means telling the program what is how important. The generated view is always drawn to fit on the screen. Hence, the tree's global structure is always visible. The more elements are omitted, the more space is available for the ones

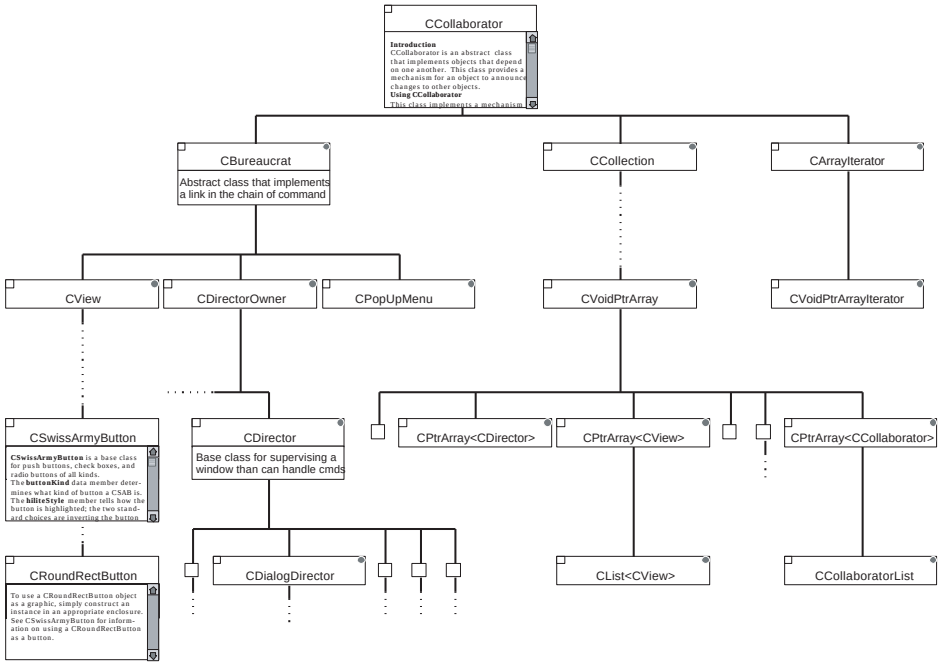


Figure 5.8: C++ class library section illustrated by the tree browser described in [AC96]. Used with permission.

viewed in detail. Figure 5.8 presents examples for the applied metaphors. The nodes, drawn as rectangles, are drawn in different levels of detail. In the least level of detail they are only small squares. In this particular application the highest level of detail shows a manual page in a separate text viewer window (not shown in the figure). Of course, those node views have to be predefined with respect to the underlying contents. The user can directly change between them via mouse actions. For node representations with levels of detail see also [Fai93]. The dotted lines indicate omitted parts in this tree. These sub-trees are said to be pruned. The authors also talk about vertical and horizontal “telescoping” in this connection.

5.3.3 MGV—Massive Graph Visualiser

MGV is an integrated visualisation and exploration system for massive multi-digraph navigation. The implemented techniques are applied to multi-graphs defined on vertex sets with sizes ranging from 100 million to 250 million vertices, data sets that do not fit in a typical computer’s main memory.

MGV is implemented as a C-computational engine (server) and a Java-3D visualiser (client). It provides a drill-down zoomable interface together with a collection of multi-linked views [AK00].

The nodes of the navigated digraphs have to be organised in a predefined tree. This is important for removing and adding detail while navigating. The representation’s visual appearance can be modified by plugging in arbitrary visualisation modules. Those modules include 2D-array diagrams as well as 3D-“multi-comb” visualisations. Incorporating geographic information is also possible. Very useful, especially for very large graphs, are several statistical functions supporting the hiding of detail. Figure 5.9 gives an example: data from telephone calls is depicted in the star-map view, superimposed upon geographic information.

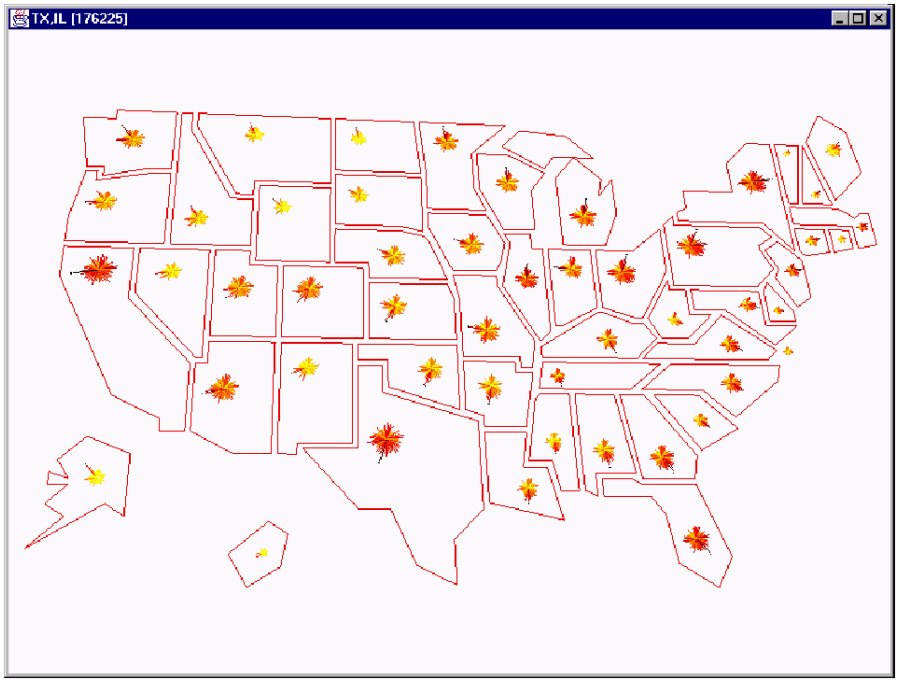


Figure 5.9: Visualisation by MGV with statistical representation incorporating additional geographic information [AK00].

5.3.4 Layout Adjustment and the Mental Map

In [MELS95], the authors distinguish between layout creation and layout adjustment. The difference is that with layout creation a picture is drawn from a given graph description. If only one node is added or removed from this graph description, a repeatedly created layout might look significantly different. This is a problem especially in interactive graph editors. If, in the worst case, the layout changes completely after every action, the user will be more occupied trying to regain comprehension of the presented structure than by actually introducing changes actively. The user's "mental map" is destroyed with each major change in layout and has to be rebuilt.

The mental map is described by three different models. The first, called *orthogonal ordering*, tries to help the user by preserving the pairwise horizontal and vertical order between nodes. *Proximity relations*, the second mental map model, considers the distance between nodes. Nodes that are close to each other should remain close. Thirdly, the drawing's *topology* is an issue. The resulting areas should keep their relative positions.

Based on rectangular nodes and straight edges, several approaches for adjusting graph layouts while preserving the user's mental map are suggested. One considered method is the force-scan algorithm. Much like the force-directed graph visualisation approach (see Section 3.2), force-scan applies simulated forces to overlapping nodes which move them apart. The movement happens along the line through the two nodes' centres. The article also discusses perspective display methods such as fish-eye mapping, orthogonal fish-eye mapping, and biform mapping. Example applications are discussed for most algorithms.

5.3.5 Information Pyramids

Information Pyramids, described in [AWP97, Wol98], is a 3D visualisation system for hierarchies. It implements a landscape metaphor. Figure 5.10 presents a view of a hierarchical structure. The base

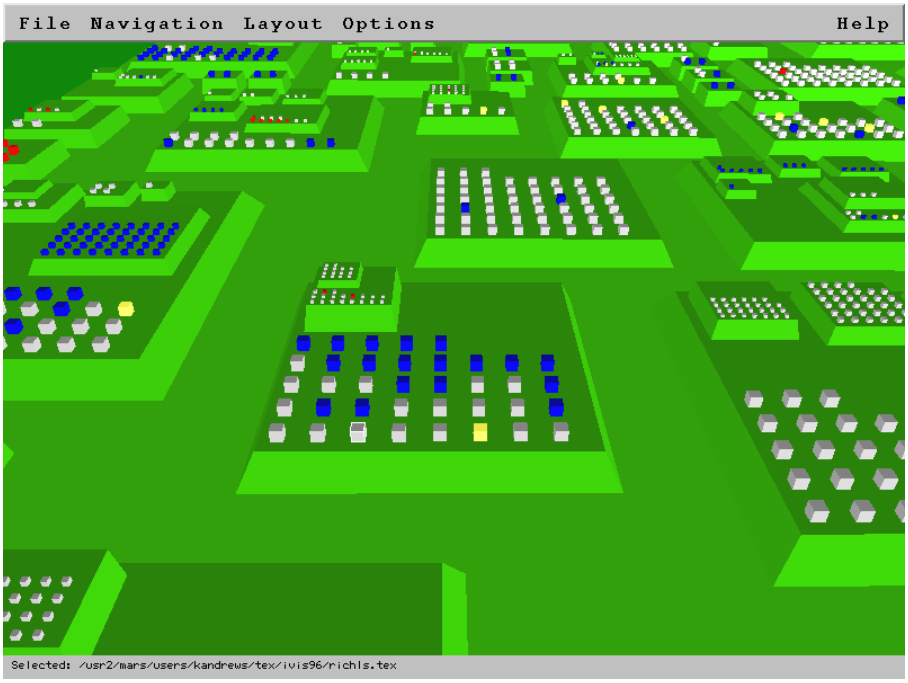


Figure 5.10: An Information Pyramids landscape view window [Wol98]. Used with permission.

represents the tree's root. All internal nodes are depicted as stacked square plateaus on a plane. The tree's leafs are drawn as small coloured boxes. The boxes' bases are sized proportionally with respect to their respective plateau's dimensions. Colour may be used to indicate certain properties, as can the leaf objects' height. For file systems, file age could be mapped to colour and file size to height.

An advantage of this strategy, compared to other hierarchy visualisation methods, is the compact layout. It avoids empty spaces which are unavoidable in general 2D tree layouts. Also, the third dimension is used more efficiently than in previous systems using the landscape metaphor. The pyramid grows in height keeping the area consumed by the base level (= root) constant. Algorithms for positioning the sub-levels as well as the leaf objects are presented. The size of sub-level plateaus is determined by its sub-tree's depth and contents. The placement of those plateaus is done minimising the remaining empty space. If the available area is too small, the leaf objects are drawn smaller. Also, several strategies for providing additional information by leafs' placement are suggested. For example one- or two-dimensional sorting or heuristics for making dependencies between objects visible through proximity. Hence, the Information Pyramids hierarchy visualisation approach leaves great freedom to add more visual means for conveying information.

5.4 Toolkits, Libraries, and Graph Drawing Systems

Graph drawing packages can be roughly classified into the following groups:

- Toolkits.
  Toolkits for graph drawing offer services to application programs via an application programming interface (API). Such services include storing, laying-out, and drawing graphs. Often the graph layout algorithms are fully automated to make applying them as easy as possible. Default parameters which can be altered to needs are common. Most toolkits also provide batch programs or server applications which can be used to separate the whole process of graph layout.

The layout program is simply called on demand. It returns the finished graph layout when terminating. When using a graph drawing server, this procedure can even be moved to a different computer. Some examples out of available graph drawing toolkits are given in Sections 5.4.3 and 5.4.4, and 5.4.2.

- Libraries.
  In contrast to toolkits, graph drawing libraries only provide data structures, algorithms, and algorithmic sub-steps for application developers. Applications using such libraries are responsible themselves for creating and drawing a final graph layout. The provided code, mostly organised in object-oriented classes, can be linked or imported into an application which needs graph drawing facilities. Such a library is described in Sections 5.4.5. Besides that, most toolkits also provide a graph drawing library, which is implemented for internal use, anyway. Section 5.4.1 describes a suggested design for a graph drawing application or system based on a graph drawing library.

- Systems.
  Graph drawing systems are, similar to graph manipulators, tools to help the user create aesthetically pleasing drawings of graphs. The difference is that graph drawing systems include many powerful graph layout algorithms. They are either applied automatically when useful or upon request by the user. The user can influence properties such as node and edge styles. Positions may be altered manually and rich navigational possibilities are offered. For an implementation see Section 5.4.7.

5.4.1 Parametric Graph Drawing

In [BdBL95] a new approach for graph drawing is presented called ALF for automatic layout facility. The basic idea is to include a great variety of algorithms in an algorithm base. Then the appropriate ones can be chosen for a particular graph. The algorithms should be adjustable for special needs. Reuse of common sub-steps is considered. The sub-division for Sugiyama's algorithm is shown in Figure 5.11.

Algorithms are characterised by their capabilities: necessary class and properties of the input graph, properties introduced to the output graph, and an estimation for the time to compute the layout.

ALF takes a graph and some requirements from the user (for example minimise crossings, obey aspect ratio, or draw edges as straight lines). Also, an upper bound on computation time may be given. These requirements can be weighted to take into account their relative importance [BdBL95]. Then the appropriate algorithms are determined and applied. Finally, the user is offered a set of finished drawings from which to choose the most pleasing. An implementation of the automatic layout facility is included in the program Diagram Server, written in C++.

5.4.2 Graphviz

Graphviz is an open source set of graph drawing tools for Unix or Windows (win32), including a web service interface [GV99]. Graphviz provides efficient algorithms for creating readable drawings of graphs. Two graph drawing approaches are supported, hierarchical layout of trees and acyclic digraphs and virtual physical layouts of undirected graphs by the spring model.

The toolkit contains a set of customisable batch programs. They read graph description and layout options from file and command line. The resulting drawing is output to file, too. The package includes a script-customisable graphic interface as well as a base library for graph tools. The particular components include:

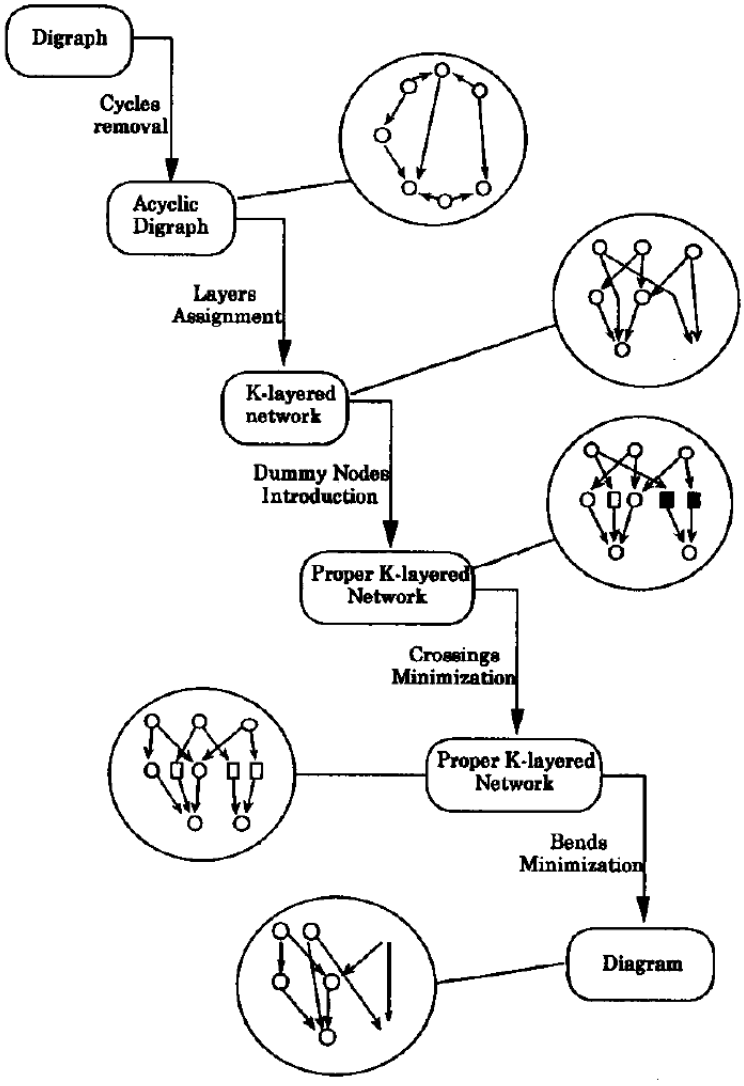


Figure 5.11: ALF's sub-steps implementing Sugiyama's algorithm and intermediate graph data [BdBL95].

- *dot* comprises the package's hierarchical part. The graph description is given as an ASCII text file obeying dot's simple graph description language. Dot can output to file in both vector and raster graphics formats. For more detail see Section 5.1.2.
- *neato* represents the physical spring model approach. It has the same input and output features as dot above.
- *lefty* is a two-view graphics editor for technical pictures.
- *dotty* is a lefty script for directed graphs.
- *lneato* is a lefty script for undirected graphs.

5.4.3 GDToolkit

GDToolkit [GDT98] (also known as GDT) is a Graph Drawing Toolkit designed to efficiently manipulate several types of graph, and to automatically draw them according to many different aesthetic criteria and constraints [GDT98].

The GDToolkit includes a graph API (GAPI) and an object-oriented C++ library of graph classes including a wide set of graph algorithms. Technically speaking, the algorithms are implemented as methods in the appropriate data structure's class. They may be inherited by derivation. This strategy makes algorithms only available where they can be applied usefully. On the other hand, a disadvantage of this implementation is that data types and algorithms are closely coupled. In a truly object-oriented design those functionalities should be separated. Otherwise it is very hard to exchange algorithms or to apply the same algorithm to different data types.

Another central part of the toolkit is the Batch Layout Generator (BLAG). It reads graph descriptions and configuration information from file and writes the computed layout back to file again. This output file contains the coordinates for the diagram's objects. BLAG is designed to work as a server under the control of a client application.

Besides choosing a graph drawing algorithm, constraints such as forcing positions or node alignment are supported. A personal trade-off between speed and layout quality may be introduced. The toolkit was tested to handle various types of graphs with up to 1000 nodes.

5.4.4 Tom Sawyer's Graph Layout Toolkit (GLT)

Tom Sawyer Software sells a Graph Layout Toolkit (GLT) providing programming interfaces to graph layout services [Tom01, GD'01]. These services include features like:

- Automatic graph layout
- Automatic edge label placement
- Incremental layout
- Complexity management techniques for nesting, navigating, folding, and hiding graphs
- Support for edge ports

GLT is implemented in C++. Specific interfaces and manual sets are available for the Java, ANSI C++, and ANSI C programming languages. The Graph Layout Toolkit has been ported to a number of UNIX, Microsoft Windows, Apple Macintosh, and OS/2 operating systems and compilers. It can be utilised by applications via two integration options: direct inheritance and loosely coupled integration. GLT scales to run efficiently whether graphs contain 5 or 5,000 objects.

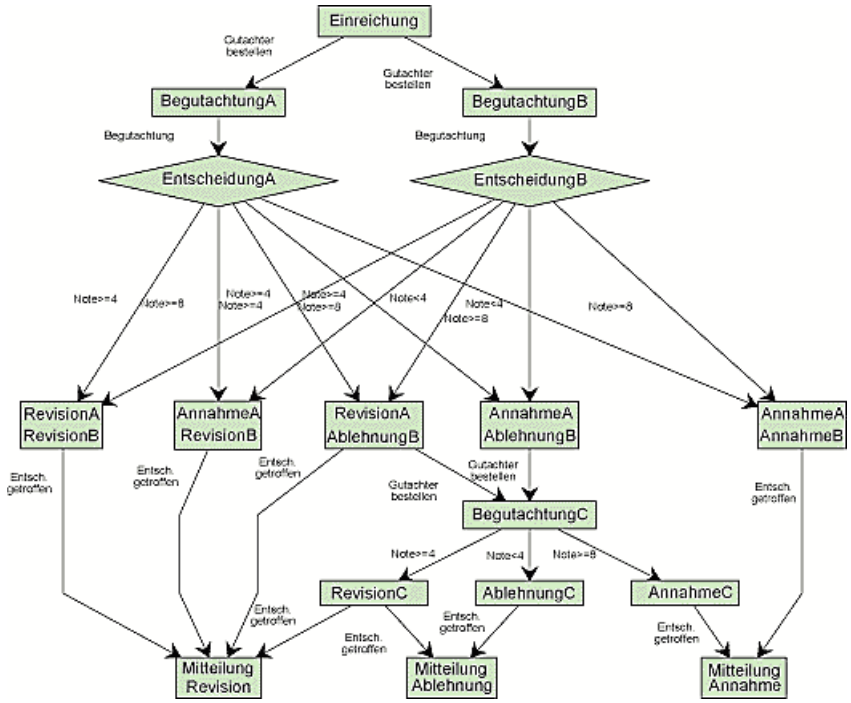


Figure 5.12: Example hierarchisation by AGD [AGD97].

Users may choose among four different automatic graph drawing styles (hierarchical, orthogonal, circular, and symmetric). Incremental layout and automatic label positioning help to create attractive diagrams. GLT's navigational features include a graph nesting and navigation system allowing drill-down, folding, collapsing, hiding, and partitioning graph elements and sub-graphs. Graphs can be stored persistently and output may be written to PostScript.

GLT has no graphics dependencies. Therefore applications retain complete flexibility for user interface development across multiple platforms. Additionally, Tom Sawyer offers a Graph Editor Toolkit (GET) to speed up implementation time. GET, building upon GLT, enables users to rapidly integrate custom diagram editor technology into their applications. The supported development environments include JDK (and Swing), MFC, and ActiveX.

5.4.5 AGD—A Library of Algorithms for Graph Drawing

AGD, the library of Algorithms for Graph Drawing is a C++ library for automatic graph drawing based on LEDA (see Section 5.4.6). Criteria for aesthetic drawings include regular and well-balanced distributions of objects, few edge crossings, and short object connecting edges with few bends. AGD offers a broad range of two-dimensional graph-drawing algorithms (implemented in C++) and provides tools for implementing new and innovative layout algorithms [AGD97, GD'01].

The current version (AGD V1.1) provides graph drawing functions like planarisation, compaction, crossing minimisation, and area adaption. Figure 5.12 gives an example of a diagram drawn by AGD with a Sugiyama-like hierarchical layout algorithm. Further features are calculation of planar and orthogonal embeddings, bend minimisation, edge and node labelling, and cluster graphs. AGD utilises the efficient data structures and algorithms provided by LEDA.

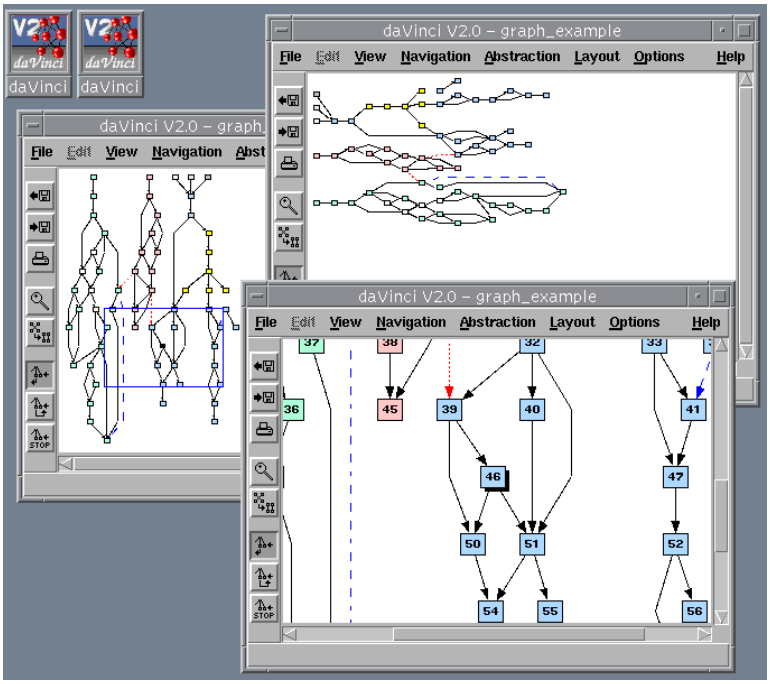


Figure 5.13: Screen shot from a daVinci session [FW94].

5.4.6 LEDA—Library of Efficient Data types and Algorithms

LEDA is a versatile, extendible, platform-independent C++ class library [LED90]. This library offers a rich set of data-types like graphs, sequences, dictionaries, trees, points, flows, matchings, segments, shortest paths etc. All components are documented with respect to their time and space complexity. LEDA also provides windowing and visualisation facilities (windows, menus, buttons, dialogue boxes, etc.).

5.4.7 daVinci

daVinci [FW94] is an X-Windows visualisation tool for drawing directed graphs automatically in high quality. The “daVinci Presenter” provides full automatic graph layout with a rich set of node and edge styles like colour, shape, pattern, icon, etc. [daV92]. In the current version “daVinci Presenter 3.0” provides automatic incremental graph layout with interactive fine tuning for a given layout. Interactive abstractions (for example collapsing sub-graphs or hiding edges) are comfortable features for navigation. Output can be written to screen or as an EPS-file. The package includes an API for client applications and an interactive graph editor for demonstration.

The example (Figure 5.13) shows a screen shot from the earlier version 2.0. Technically speaking, daVinci implements a version of the Sugiyama algorithm with many extras and layout extensions, one of which is incremental drawing.

5.5 Summary

Many very different programs have been shown and their advantages and disadvantages have been pointed out. Graph drawing applications have been divided into batch drawing programs for static pictures of graphs, graph browsers and editors with a special section on information visualisation, and finally development packages like libraries and toolkits as well as complete graph drawing systems. For detailed information about graph drawing programs not mentioned in this thesis see, for example, [Fai93] or [HMM00].

Chapter 6

The Architecture of JMFGraph

Contents

After discussion of research and theoretical results, this chapter will discuss practical results from JMFGraph. This program has been designed and implemented to explore theoretical considerations and for empirical efficiency testing. JMFGraph is implemented in Java (version 1.3) and provides a graphical user interface based on Swing. No third-party libraries are used for graph handling or layout creation.

As common in the object-oriented Java programming language, JMFGraph is divided into several modules called packages. Figure 6.1 shows a sketch of the hierarchical package structure. In the following these packages are described briefly.

jmfgraph
: This main package contains classes for central coordination. One of these is the main class "JMFGraph" which starts the program. Also the static class "InstallReg" is placed here which holds information about which installed modules and components should be offered to the user for choice. See Section 6.6.

jmfgraph.inputmode
: As explained later, the graph source handling is organised modularly. Figure 6.1 reveals that JMFGraph currently has two input modules called "demo" and "dot". An input module is stored in its own sub-package and contains methods to choose, open, and read a particular source. Input Modules are instantiated by a special factory in this package. JMFGraph graph source input modules are described in Section 6.1. Local edge and node representations are also stored in the respective input module package.

jmfgraph.graph
: The graph package is responsible for graph data structures. The current program provides a multi-linked edge map implementation based on Java's TreeMap to speed up graph processing. This graph structure is also utilised in the drawing structures. See also Section 6.4.

jmfgraph.drawing
Everything concerning the visual graph representation is placed in this package. Classes for general and for layered drawings are provided here. Data structures for handling drawings as well as paintable representations for nodes and edges which are available for all graph source input modules. Representations are also instantiated by the mentioned factories. See also Section 6.2.

jmfgraph.layout
The layout includes the graph layout algorithms which are stored in the sub-package "algorithm". Algorithms and sub-algorithms are instantiated by the algorithm factory in this layout package. Layout algorithms may make use of an arbitrary number of sub-algorithms which may also be used by other algorithms. See also Sections 6.3 and 7.1.

jmfgraph.userinterface
User interface holds classes for the Swing interface including classes for display, such as a menu bar, a zoomable panel, and an options frame, as well as interaction facilities for reception of mouse and keyboard events.

jmfgraph.io
Declares interfaces for input and output graph streams.

jmfgraph.util
For universal helper classes.

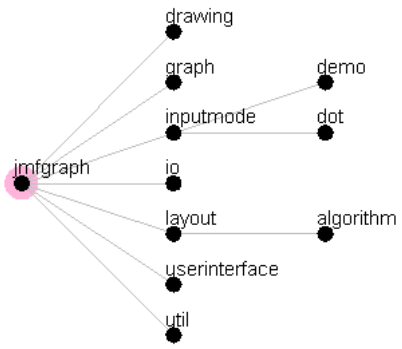


Figure 6.1: JMFGraph module structure visualised by JMFGraph.

6.1 Graph Source Input Modes

JMFGraph uses graph source input modes as packages which adapt the program to a specific graph source. Possible examples for such sources are a file system to view files, links, and folders graphically. Or, with the restriction that hyperlinks cannot be followed backwards, an input mode for browsing web pages by following links is conceivable.

Besides providing methods for picking and reading a particular source, information about available and currently selected representations is connected with this input mode. Instantiable modes are registered in the InstallReg class. The input mode factory creates instances which are requested providing the mode name. Each input mode module itself provides a factory which produces the mode specific components.

The currently implemented input modes include the “demo” mode which provides three different graphs and the “dot” mode which can read a graph description file. This file has to obey the grammar rules of the dot graph grammar described in Appendix C. The other input mode, demo, lets the user choose from three available graphs for demonstration purposes: K_5, $K_{3,3}$, and a general small demo graph with cycles.

6.2 Node and Edge Representations

Node and edge representations are useful tools to change a drawing’s look. The user may interactively choose from the installed representations. The diagram is updated immediately. The representation may even make use of arbitrary information added to the graph elements when they are read in from source. For example, icons or interactive graph elements could be introduced using this feature.

Representations may be local to one input mode or available for all nodes. Instantiation, whether local or global, is generally controlled by the input modes’ factories. Information about which mode allows which representations for nodes and edges is included in InstallReg.

6.3 Layout Algorithms

JMFGraph currently provides two Sugiyama-style hierarchical graph layout algorithms, a standard and a focus-based version. More detailed descriptions for the implementations are given in Section 7.1. The algorithm may be modularly altered or enhanced by replacing its algorithmic sub-steps. New algorithms may be introduced: hierarchical ones as well as algorithms following a different graph visualisation approach. Of course, some algorithms need special data structures and drawing implementations. Any additional representations for edges and nodes may have to be provided in addition to the new algorithm.

To provide greater flexibility, a set of interfaces is defined which describe the available features a particular algorithm or sub-algorithm provides. Interfaces are included for the following features: view mode (local / global), stepping ability, and orientation.

Local View The local view mode lets the user browse through the graph by supporting a focus, which is centred in the graph display. The user may pick any visible node as focus by simply clicking it with the mouse. Besides centreing this node the layout algorithm is recalculated for the new centre of interest. Only the visible portion of the graph is laid out in order to speed up user interaction. The visible part is determined by considering the graph, the focus, the current window size, and the current scale factor.

Global View In contrast to the local view, the global view mode always lays out the whole graph. If part of the drawing does not fit in the display area, the visible part may be viewed using scroll bars. Initially, when switching to global view mode or when opening a graph, the scale factor is adjusted to make the whole graph visible. Regardless of the window size, the drawing size is computed from the layout only.

Step Mode The step mode enables the user to view intermediate results in cases where an algorithm consists of multiple sub-steps. This functionality may be turned on and off. When on the algorithm is executed step-by-step. At any stage, the navigational functionalities such as browsing, scrolling, and changing orientation are fully available.

Orientation Orientation supports the four directions up, down, left, and right. Implementing algorithms do not have to support all directions. It is left to the particular algorithm to handle the orientation commands forwarded from the user interface.

An algorithm may implement both the local and the global interface to support fast local layout as well as focus-adjustable global layout.

6.4 Internal Data Structures

The graph data structures implemented in JMFGraph are designed to favour computational efficiency over low memory consumption. The graph elements are heavily linked to speed up various kinds of graph traversals. Nodes, for example, contain information about their incident edges, and edges hold information about their end nodes. Additionally, container structures for both nodes and edges can be obtained from the graph data structure.

Drawings are always based on a graph which should be displayed. In addition to this, a drawing also contains an own graph. This drawing internal graph holds the part of the original graph which is currently visible. Besides that, the internal graph is used to hold layout information like position, size, and graphical properties.

6.5 The User Interface

The JMFGraph user interface consists of two Swing frames. The main diagram frame is responsible for displaying the created graph layout on a re-sizable panel. It supports zooming the display and browsing the graph. This is possible by setting the focus to any visible node which is then moved to the centre and the drawing is updated. The focus is visualised by a red outline.

Additionally, an options frame is provided. This second frame can be opened on demand and may be used in parallel with the drawing frame. It offers choices for input modes, algorithms, and representations to be used. Some of the options frame's functionality is also accessible via the main frame's menu bar. For more detail and screen shots, see the JMFGraph user guide in Appendix A.

6.6 InstallReg

The static class file InstallReg in the main package has static lists containing information about the currently available modular components. These lists are utilised to offer the user a choice list of available options. The following static string lists are contained in InstallReg:

- List of input modes.
- List of layout algorithms.
- List of applicable node representations (for each mode).
- List of applicable edge representations (for each mode).

The latter two are two-dimensional lists providing one list for each input mode in the respective list.

Chapter 7

Selected Details of the Implementation

Contents

As an example, Figure 7.1 shows the K_5 graph. It is laid out using focussed layering in horizontal orientation. Figures 7.2 and 7.3 show two views of JMFGraph displaying the sample graph JWF1. The source code for the displayed graph is printed in Appendix C. Section 7.1 describes the focus-based graph layout algorithm implemented in JMFGraph. The second, standard, algorithm differs only on one sub-algorithm. Section 7.2 concentrates on the layout algorithm mentioned in the previous section. It has been especially developed to support "browsing" exploration of graphs.

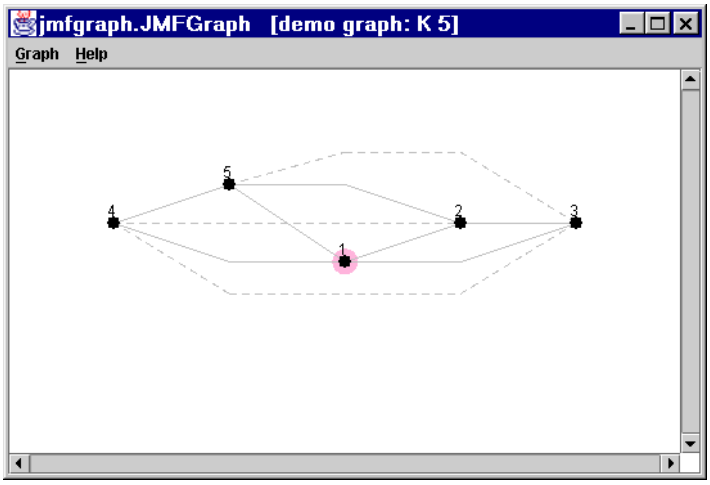


Figure 7.1: K_5 graph visualised using focussed layout.
The graph's edges are provided with orientations to make the graph directed.

7.1 Sugiyama-Style Focus-Based Graph Layout

The main aim of the focus-based layout is supporting the browsing technique for navigation. The user may explore even a very large structure by setting a focus repeatedly to visible nodes. The

program reacts by presenting a new layout with the new node in focus centred and other nodes placed considering the connectedness to the focussed node. So the layout is recalculated on each navigational move. To speed up things, only the visible part is determined an laid out.

As explained in Section 3.1, graph layout algorithms following the hierarchical approach are usually divided into four steps. The following list explains how each respective sub-problem is approached in JMFGraph.

1. Layering.
   The first step is implemented using the new layering algorithm explained in detail in the next section (Section 7.2). The layering is constructed around one focussed node. A pre-processing step is used to remove self-loops which would irritate the algorithm. The resulting layering is made proper by a separate subsequent step which also marks edges as "forward pointing" or "backward pointing". An example of the focus-based layout is shown in Figure 7.2.

   The second algorithm implementation differs in this step. It uses a conventional "Longest Path Layering" algorithm, see also Section 4.2.1. The necessary cycle removal is achieved implicitly by executing two passes where the first generates a valid but, if cycles were contained, very stretched layout. The second then uses the resulting graph from the first pass without cycles. In addition to horizontal and vertical orientation, this algorithm is capable of layering the graph's nodes topmost/leftmost or bottommost/rightmost. See Figure 7.3 for a vertical bottommost drawing.

   Both layering algorithms finally decompose polyline edges into multiple straight edges. This is necessary for the following steps.

2. Crossing reduction.
   Crossing reduction is done by an implementation of the "Sifting" algorithm described in Section 4.3.4 [MSM99]. The original algorithm is enhanced by a mixed pass technique where, in defined alteration sequences, either both or only one neighbour layer is considered for calculating the difference in crossings when a node is moved within the layer.

3. X-coordinate assignment.
   The algorithm implemented for this step is the original "Priority Layout Method" proposed with the original algorithm by Sugiyama (see Section 4.4.2) [STT81]. A special modification is introduced to this algorithm to enhance symmetry display capabilities. If in one layer more than two subsequent nodes are assigned the same priority, the middle one is increased to be laid out first. This yields aesthetically pleasing symmetries especially for tree structures with multiple child elements.

4. Cleanup and drawing.
   The final step is capable of introducing either horizontal or vertical orientation to the final layout. Furthermore, all layers and rows of positions are moved apart depending on the contained nodes' sized to keep the space between nodes constant. Nodes of differing sizes can also be handled correctly.

   Implementing step mode (see Section 6.3) made an additional finishing algorithm necessary because the drawing has to be displayed before the actual finishing step has been executed. A very trivial algorithm which directly maps layers and positions to cartesian coordinates is sufficient for this purpose.

7.2 Focussed Layering

The focussed layering algorithm was designed to meet the demands of an application for graph browsing. The aim was to concentrate the closely linked neighbourhood around the focussed node to explicitly show the relations between the focussed node and its environment. The first idea was to use the well known Longest Path Layering once for each direction. But the resulting "hour-glass" diagram is not as helpful as it might seem at first. The smaller problem is that nodes might appear in both halves of the drawing. The much bigger difficulty is that all directed paths not crossing the focussed node are missing. Adding those paths later is possible, but is very time consuming and also produces poor layouts because entire layers possibly have to be shifted. For example, if a side path is longer than its end nodes are apart in the layers of the hour-glass diagram.

Basically, what the algorithm does is making a sweep from the centred focussed node away to the borders of the drawing scope. This is very straight-forward, because the algorithm wants to emphasise the focussed node's environment. So the closer a node is connected to the focus, the earlier it gets the possibility to take good position. The algorithm only deals with directed edges and ignores undirected ones, which may be present nevertheless. The algorithm's structure is described in the following pseudo-code.

FocussedLayering

1. Set up two sweep lines, one following the directed edges (the forward SL) and the other stepping the other way (the backward SL). Both are initially filled with the focus node. The area between those sweep lines is called the sweep area (SA).
2. Make alternating steps with these two sweep lines. Start with the forward SL to favour the forward direction.
3. For each sweep line step do the following:
   (a) Find all nodes which lie one edge in front of the current sweep line.
   (b) Empty the current SL adding the contained nodes to SA. All edges connecting the new nodes with the previous SA nodes are added to the drawing. Move the SL to the next layer and fill it with the nodes found.
   (c) Remove nodes which are inside the SA or on a SL (including this one to avoid edges inside the layer).
   (d) Update SA (see below).
4. The algorithm finishes if both SLs are empty or a predefined number of layer steps is exceeded.

Update Sweep Area

This sub-algorithm uses an additional sweep line, the centre SL.

1. Fill the centre SL with the nodes of the outer SL that has just been moved.
2. Start iterative stepping inside the SA.
   (a) Fill the centre SL with all neighbour nodes of the previous centre SL following the current sweep direction. Nodes which are member of SA or an outer SL are not added.

(b) Remove nodes from the SL which follow other nodes in the SL in the current sweep direction.

(c) If nodes have been found mark this layer as changed.

(d) Change direction if no new nodes have been found and no layer ahead is marked as changed or if the bounding SL has been reached.

3. Stop if two direction changes immediately follow each other.

Updating the sweep area after each SL step is necessary because otherwise only a forward and a backward tree would be the result of this layering. It is crucial that this update happens as soon as new graph components for the SA are found, so the nodes consecutively found by the outer sweep lines can be put to a final layout. If this does not happen, it is, in general, very difficult and time consuming to insert the missing paths into the double-tree drawing. Nodes or even whole layers would have to be shifted if an outer path connecting the two sides is found which is longer than the respective nodes are apart.

7.3 Results

To find out about the usefulness of this new approach Figures 7.2 and 7.3, showing the same graph, can be compared. Besides the different orientations, which are not a matter of algorithms, the drawings differ in qualities like compactness, number of crossings, path tracability, and many more. The most important common and differing properties are discussed in the following.

Both drawings have no backward edges, although this is not the case in general. Also, both drawings were layered freely. This means that no additional information on where to put which node was used. It is important to mention, that both drawings only differ in the layering algorithm used. All other algorithmic steps, like crossing reduction and positioning, were accomplished by the same algorithms. Such test settings best emphasise the differences introduced by the algorithmic parts of interest.

Figure 7.3 was layered with a conventional Longest Path Algorithm. From the options panel can be seen that the selected orientation is vertical with all nodes at the bottommost position. This means that all sinks are at the lowest layer. The number of crossings is 97 in this drawing, and as already stated there are no backward edges. The node $S8$ is provided with the focus mark, but the focus has no influence on the resulting layering.

Other than in Figure 7.2, laid out with the Focussed Layering. Node 4 is made the centre of the drawing by setting the focussing to it. The drawing is laid out to maximise tracability of paths crossing the focus node 4. The layout is very compact in this case, because node 4 is very central in the given graph structure. The number of crossings, 80, is less than in the other layout. Being designed for graph browsing, the focussed approach has an additional benefit. The number of available drawings is virtually equal to the number of nodes. The user may choose any, for example the one looking most central to him, to create a balanced and compact drawing.

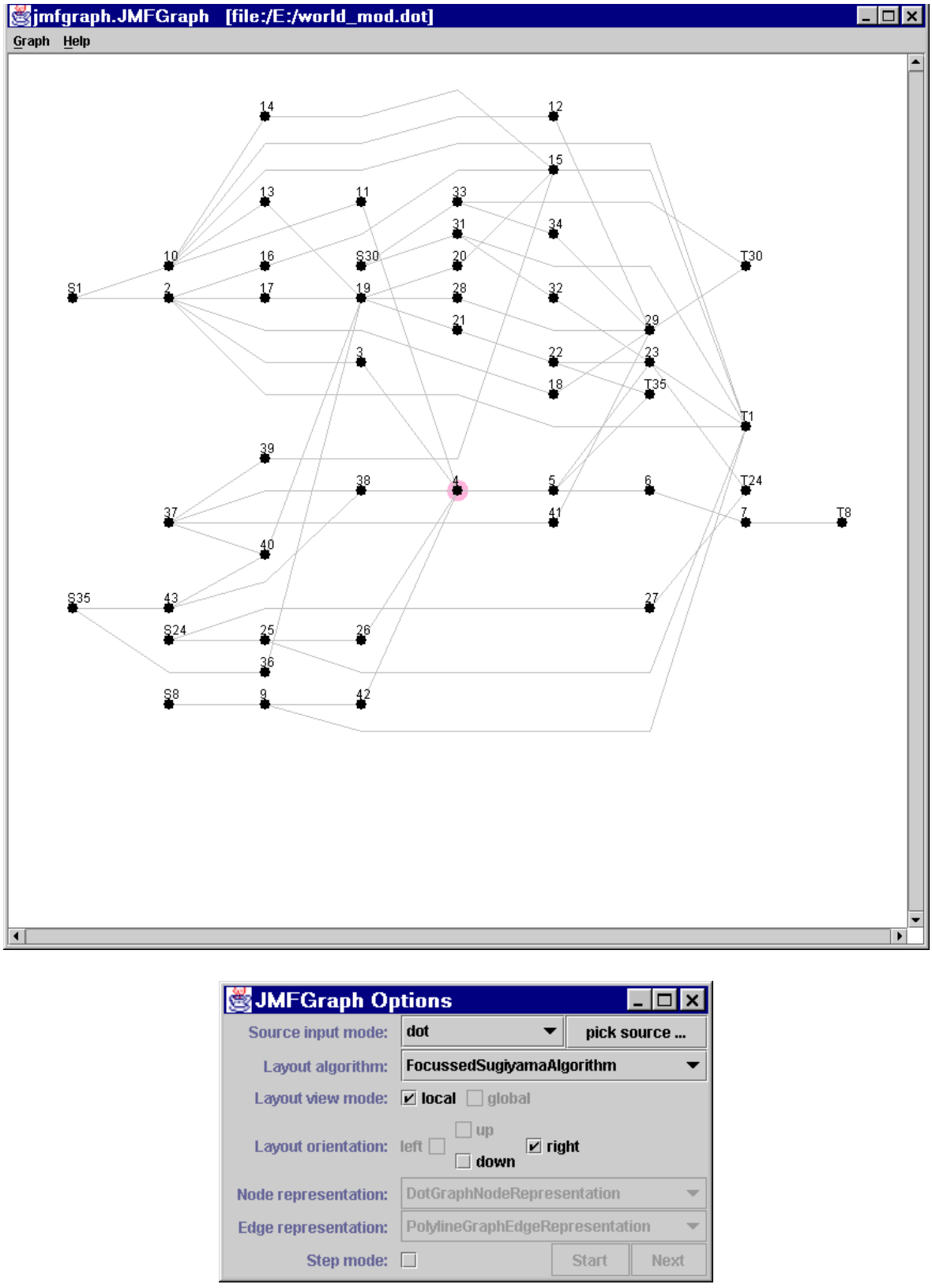


Figure 7.2: JMFGraph visualisation of JWF1 using focus-based layout. The node with label “4” is the current centre of focus.

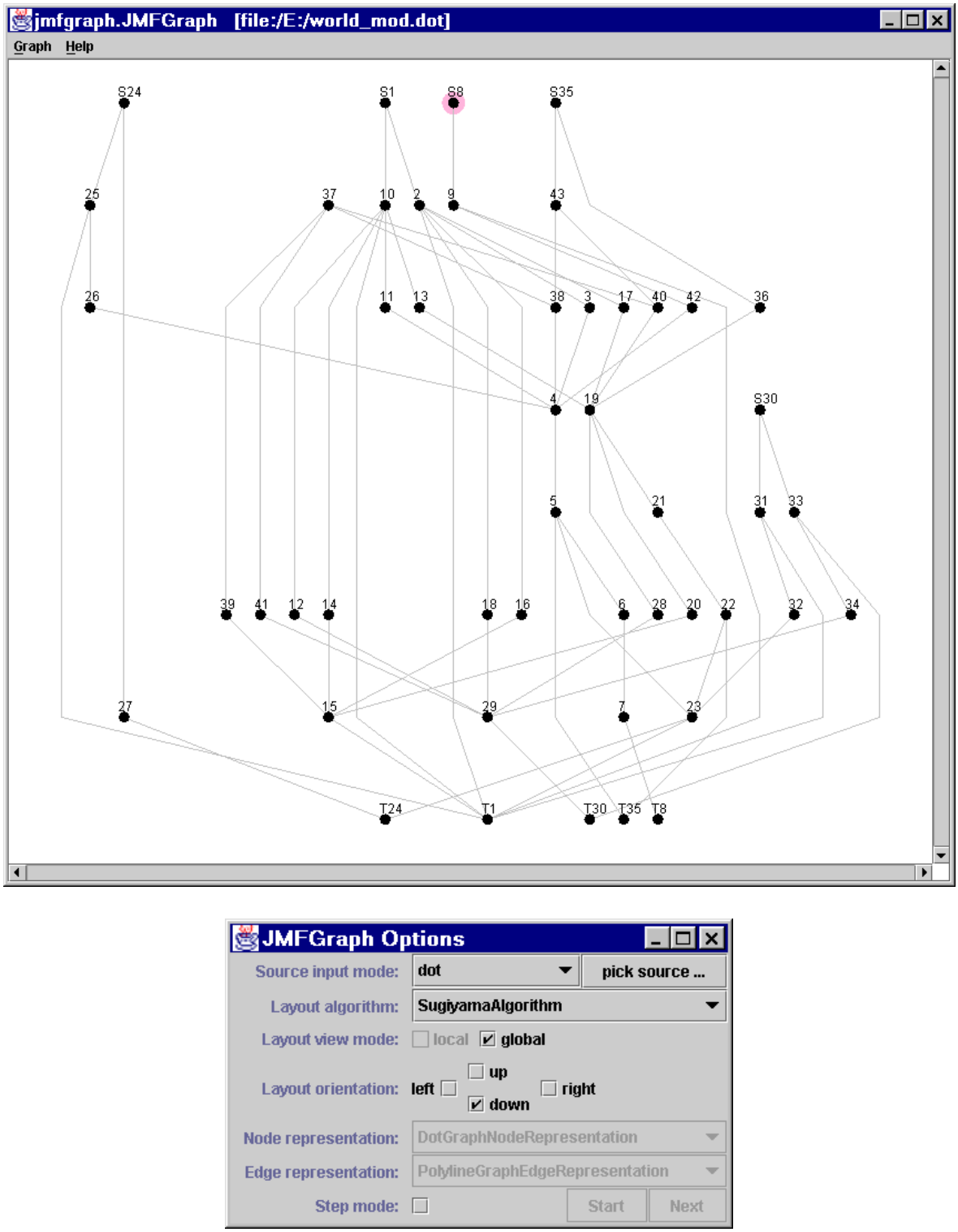


Figure 7.3: JMFGraph visualisation of JWF1 using non-focus bottommost layout.

Chapter 8

Outlook

8.1 General Trends

It is rather surprising that current research mainly concentrates on two types of graph drawing application. One type is the batch layout calculation described in Section 5.1 where static graph visualisations are generated to be printed or stored as a picture or drawing. Besides that, the second type of intensively researched graph drawing applications is the field of interactive and iterative layout creation applications. Such approaches were considered in Section 5.2. The user is responsible for giving the layout generation processes hints and advice and so the final layout is a cooperation of human and computer.

Much less research work is currently spent on the type of application that JMFGraph belongs to: interactive graph browsers. Besides [Sch98, And96] (and [AC96] for trees) almost no application or publication dealing with the focus-based interactive browsing of large graphs was found in the field of graph drawing, where in this case interactive means that the user is not interested in modifying the presented layout. Let us call it ad-hoc automatic graph layout.

Nevertheless, the field of ad-hoc automatic graph layout creation seems to be very interesting and promising. As the amounts of information available are ever increasing. Therefore nothing seems more helpful then creating tools which can create useful and navigable representations without additional efforts for the user. So, if a pleasing layout is created automatically, the user can completely concentrate on the contained information.

In other words, such algorithms could be applied and modularly exchanged very universally. And furthermore, if for example the XML [BPSM98] standard emerges as promised, enormous new application possibilities will arise. Virtually each XML document, or also sets of XML documents, could be converted to a XML graph representation like GraphXML (see Section 2) by appropriate XML converters (i.e. Extensible Style Language—XSL [XSL01]). This means the more documents are available in the XML format, the mightier would be a universal XML enabled ad-hoc graph drawing tool with a set of XSL input filters.

8.2 Ideas for Future Work

Although there could be much improvement, this section tries not to concentrate on the user interface too much. It is clear that this program, written in the scope of a master's thesis, can in no way compete with products developed by institutes or even commercial organisations over years with the help of many people. Especially since everything in JMFGraph has been written from scratch. So the main

results should be seen in the particular implementation of the layout algorithm and in the structure of object-oriented framework.

Nevertheless some points concerning the user interface are collected here briefly. Besides node labels, also edge labels should be supported. In this connection, the possibility to pick edges as focussed elements should be considered. Edge representations with arrows in addition to full and dashed edge drawing style should be implemented. Some layout algorithms cannot work without arrows. They also make distinction between directed and undirected edges more obvious if both appear together in one diagram. Though the main navigation mode for JMFGraph is the browsing mode, it should be completed by a panning mode where the drawing is painted without a focus. Of course this enhancement would make adaptions to the layout algorithms necessary. Therefore interfaces are prepared for algorithms to declare their capabilities (as implemented for the step mode). Finally, users certainly would appreciate the possibility to influence distances between layers and between nodes in general.

A method to restrict applicable edge representations not only by the graph source mode but also by the graph layout algorithm would be useful. It is obvious, that some algorithms can only work with certain types of edges. For example a force-directed algorithm makes no sense with curved or polyline edges. Being particular, the input mode called “dot” should be renamed as “file mode”. So any kind of graph representation file, for example also XML graph files, could be handled by this mode, provided appropriate filters are installed.

The dot parser, together with the respective representations, should be enhanced to support all dot grammar features that are not yet supported. Examples for this are node or edge attributes like shape, colour, and stroke, subgraphs and several options and command for layout constraints.

The implementation of methods for handling self loops, multiple edges, and non-connected components would be useful. Of course, any additional layout algorithm or sub-algorithm for the hierarchical layout implementation would be an enrichment.

Currently, there are no animated transitions from one view to the next. This feature could be added in a separate module creating intermediate pictures between two calculated graph layouts. A trivial implementation would move elements on straight lines. Although even such simple could help the user following drawing transitions, optimising such animated transitions is still a researched issue.

When considering bigger and/or more complex graphs, it would be useful to separate layout calculations from the GUI processes. This should be done by putting this part into a thread of its own. Certainly, both threads have to be synchronised in a proper way to prevent the user interface from freezing if the layout calculation takes more than noticeable amounts of time. Also a more efficient hit detection for mouse clicks on the drawing space should be included to speed up interaction times for large graphs.

Chapter 9

Concluding Remarks

JMFGraph is an extendible framework for graph visualisation programmed in Java 1.3 using a Swing-based graphical user interface. The current implementation includes to versions of a specially adapted Sugiyama-style hierarchical graph layout algorithm. One of which supports navigation via a focus-based browsing technique. Graphical graph element representations include basic node and edge representations. The node representation may be labelled with a character string.

An ASCII file parser is included which reads the dot graph file syntax's main features. It reads in graph information from files selected by the user. JMFGraph is open to incorporate additional graph layout algorithms, including those following other than the hierarchical visualisation approach. The design also supports additional representations for nodes and graphs.

The aim of this thesis was to create a universally extendible graph browsing tool which creates layouts automatically without user intervention. Besides displaying entire graphs, the design also supports the design of input modes which only load sub-graphs necessary for the current scope. This makes it possible to display data from large databases or, for example, structures on the Internet.

Appendix A

User Guide

Contents

Section A.1 describes how to start JMFGraph from the command line and which options may be given. In Section A.2 the principles of source input modes are discussed. The following sections explain the graphical user interface (GUI) of JMFGraph. The main display frame is presented in Section A.3. The separate frame for setting options is explained in Section A.4. Finally, Section A.5 describes how to install the program or modular program components.

A.1 Starting JMFGraph from the Command Line

JMFGraph is provided as an executable jar file "jmfgraph.jar". When executing this file with a proper installation of Java 1.3, the empty graph display frame will appear. The user then has to choose a graph source input mode. This can be done either via the menu "Graph—Mode" or from the options frame (see Section A.4). An example starting command is shown below.

```
java -jar jmfgraph.jar
```

The program should be started from a shell in foreground mode, because JMFGraph outputs useful information and action feedback for the user. For example the current scale factor or information on which step was executed in step mode.

A.2 Graph Source Input Modes

If no input mode has been given as command line parameter, the display frame appears empty. As already described in the previous section, the user has to pick an input mode before a graph can be displayed. Such an input mode module combines capabilities for the following tasks:

- User interface for selecting a particular graph source (for example a file chooser dialog.)
- Streams to communicate with a selected graph source.

- Currently selected representations for nodes and edges.
- Additionally, applicable node and edge representations are defined for each input mode. (See also A.5.)

Currently, there are two source input modes implemented. The input mode *demo* is a very simple mode for demonstration purposes. Selecting a graph source in this input mode is not possible because the demo graphs are encoded inside the demo classes.

Input Mode *dot* is more powerful. It can load graph information from a text file which may be selected via a file chooser dialog. The file has to be formatted according to the dot graph file format described in Appendix C.

A.3 Graph Display Frame

If no input mode has been given as command line parameter, the display frame appears empty. As already described in the previous sections, the user has to select a input mode before a graph can be displayed. Figure A.1 shows an example view of the graph display frame showing a graph. Here the user has the following possibilities of interaction:

- Usual frame components.
  Some of the usual frame interaction methods are:
  - Resizing the frame by dragging the frame border.
  - Iconifying, maximising, restoring, and closing the frame with the menu and icons in the frame title bar.
  - Picking commands from the frame's menu bar.
  - System global keyboard short-cuts, for example to close the frame and quit the program.

  The frame's title bar is inscribed with the main class' name and a string identifying the displayed graph's source.

- Zooming in and out.
  Using the keyboard the scale factor of the display can be changed. The diagram is enlarged by typing `Page-Down` or `+`, keys `Page-Up` and `-` shrink the drawing. The `Page-` keys make larger steps, and the step size for all those keys can be increased by holding `Shift` and/or `Ctrl` and decreased by holding `Alt`. Additionally, the key `Home` resets the scale to the initial value.

- Picking a focussed node.
  Looking at Figure A.1, the node in the centre of the display area is marked by a red border. This is the notation for the focussed node. The focussed node is always centred in the drawing area. The user can move this focus by clicking any other visible node with the mouse. The new focus, again, is moved to the centre and the graph is laid out respecting the new focus.

- Panning over the static drawing.
  JMFGraph supports a global view mode which lets the user explore the diagram by moving the visible section with horizontal and vertical slide bars or by arrow keys. (See also Section 6.3 for a detailed description.)

Representations can be installed modularly (see Section A.5) and arbitrarily chosen for display by the user (see below). In the current implementation nodes all look the same and have the same size.

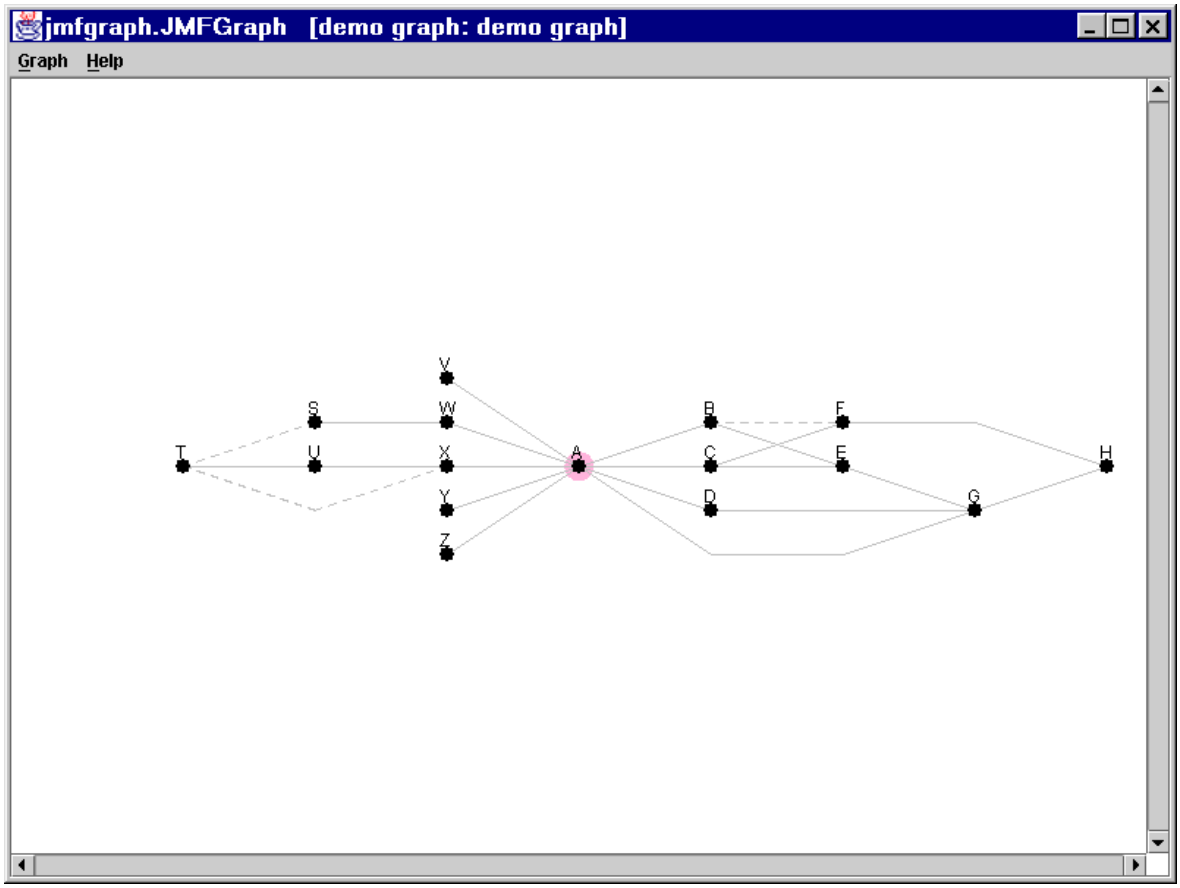


Figure A.1: JMFGraph main graph display frame.

Only the label indicates the node's identity. Edges are currently drawn without arrow heads showing their direction. Instead, edges pointing in the drawing direction (from left to right) are drawn as full lines. Other (backward) edges are drawn dashed to distinguish them.

A.4 Options Frame

The options frame provides the user with various GUI components to make choices or issue commands to the program. An example is shown in Figure A.2. At start-up most components are empty or deactivated. After selecting a graph source input mode from the uppermost drop-down list, those components are filled with the settings from the chosen mode. Alternatively, the same list of input modes is available in the main frame's menu bar as sub-menu "Graph→Mode". The button "pick source . . . " (which is also present as menu entry "Graph→Pick source . . . "in the graph display frame) activates the input mode's procedure to pick a particular graph source to be displayed. This command is automatically activated when the input mode is changed by the user.

Below that, a layout algorithm can be chosen from the list of installed layout algorithms. In the current implementation two versions of Sugiyama's algorithm are available. The layout algorithm can be chosen independently of the graph source input mode. If the selected layout algorithm supports stepping, the check box at the bottom of this frame can be checked. This makes it possible to view intermediate layout results produced by the layout algorithm. Clicking the button "start" tells the algorithm to stop after its first step. "next" advances to the next intermediate result until the final

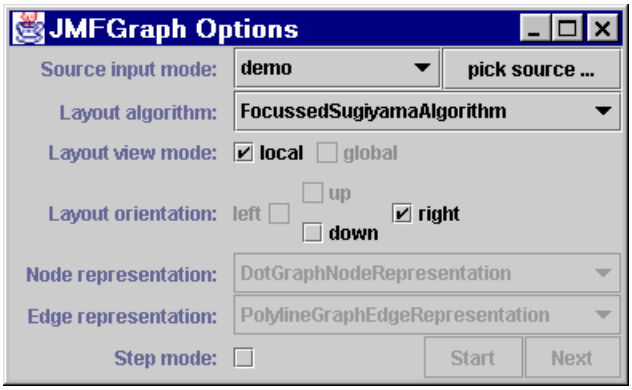


Figure A.2: JMFGraph options frame.

layout is reached. A short description on each step is printed to the shell where JMFGraph is running.

Representations for nodes and edges can be picked from the respective drop-down lists in the frame's centre. The provided alternatives depend on the selected input mode. Currently there is no restriction considering the selected layout algorithm. So selecting an inappropriate representation, for example for edges, might result in a displeasing layout. Notice that drop-down fields are disabled if no or only one element is available, because no choice can then be made.

A.5 Installation

The JMFGraph program is simply installed by copying the executable jar archive to an arbitrary directory. To be executed, the executing machine must have Java 1.3 installed. A few more steps are necessary to install additional layout algorithms, graph source input modes, or graph element representations. Of course, all such modules have to conform the JMFGraph interfaces.

- Installing additional layout algorithms:
  The properly coded layout algorithm has to be put into the package "jmfgraph.layout.algorithm". Also the static class "jmfgraph.InstallReg" has to be updated by adding the algorithm's class name to the list of algorithms.

- Installing additional graph source input modes:
  The whole new input mode package has to be added as sub-package to "jmfgraph.inputmode". Then the input mode has to be registered by adding its name to the input mode list in "jmfgraph.InstallReg". Also the representation lists in this class have to be extended by sets of representations for nodes and edges which are applicable. Here global representations and representations local to this input mode may be used (see also below).

- Installing additional representations for edges or nodes:
  Representations may be added either globally or locally for one input mode. The class file has to be added to the package "jmfgraph.drawing" to add it globally. For local representations, the respective mode package "jmfgraph.inputmode.<mode>" has to be updated. The static class "jmfgraph.InstallReg" has to be updated in either case by adding the representation's class name. It has to be updated in each input mode's node/edge representation list where it should be available. Of course, local representation can only be used in input modes in which they are local.

Appendix B

User Requirements

Contents

JMFGraph is a visualisation framework for graphs. Its purpose is to serve as a basis for a graph drawing package. Sub-algorithms, for example for cycle removal or crossing minimisation, can be inserted modularly to form the processing sequence(s) for the desired visualisation method.

B.1 Layout Processing

As already stated, the visualisation algorithms have to be supplied as modular sub-algorithms. The visualisation process is assembled from them through a special class. Designing different paths through the process is possible.

JMFGraph includes an implementation of Sugiyama's algorithm. Heuristic sub-algorithms are chosen, but not necessarily the original ones. The algorithm can deal with graphs under the following limitations:

- The graph may have undirected and directed edges, but only directed edges are drawn.
- The graph need not be connected, but only the connected component containing the first node (or, if supported, a node marked as focus) is shown in the diagram.
- Self loops and multiple edges are not supported. Self loops are removed before layout creation and a warning is printed. Multiple edges are treated as a one single edge.
- Cycles are allowed, but create backward pointing edges. In the drawing such backward edges are distinguishable from forward edges.
- The program has to provide appropriate interfaces to add layout algorithms and representations for nodes and edges modularly.

B.2 Data Source Interface

Interaction with the data source for the visualisation is defined as read-only. The open design of this interface allows adding more input drivers for other data sources. The program includes input drivers for:

- Graphs in ASCII Dot files.
  A subset of Dot's syntax is supported, see also [Nor93] and Section 2.3.2. This input driver is mainly used for testing.
- Demo Graphs.
  Hard-coded graphs for demonstration purposes.

B.3 User Interface

The user can interact with the program at run time via a graphical user interface. Also some options can be preset via command line at start-up. For browsing and navigating the user has the following possibilities:

- Zooming: variation of the displayed contents' size.
- Focussing: the selected node is centred.
- Stepping: layout algorithms can be instructed to stop after each layout step making intermediate results visible.
- Options: modifications of layout properties may be requested via a separate "Options Frame".

Standard input devices for these actions are keyboard and mouse.

B.4 System Specification

The implementation uses Java version 1.3. The graphical user interface is based on Swing.

Appendix C

The Dot File Format

Dot is a graph layout tool contained in the Graphviz package (see Section 5.4.2). It loads the graph to be laid out from a text file following a simple syntax. JMFGraph's dot input mode includes a file parser which can read a sub-set of dot file functionality. In Table C.1 the dot file grammar from [Nor93] is given. The used notation is explained in Table C.3.

graph	→ *[***strict***]* ⟨**digraph** | **graph**⟩ *id* { *stmt-list* }
stmt-list	→ *[stmt [***;***] [stmt-list]]*
stmt	→ *attr-stmt* | *node-stmt* | *edge-stmt* | *subgraph* | *id* **=** *id*
attr-stmt	→ ⟨**graph** | **node** | **edge**⟩*[* **[***attr-list***]** *]*
attr-list	→ *id* **=** *id [attr-list]*
node-stmt	→ *node-id [opt-attrs]*
node-id	→ *id [***:** *id]*
opt-attrs	→ **[***attr-list***]**
edge-stmt	→ ⟨*node-id* | *subgraph*⟩ *edgeRHS [opt-attrs]*
edgeRHS	→ *edgeop* ⟨*node-id* | *subgraph*⟩ *[edgeRHS]*
subgraph	→ *[***subgraph** *id] stmt-list* | **subgraph** *id*
id	→ *any alphanumeric string not beginning with a digit, may include underscores* |
	→ *a number* |
	→ *any quoted string possibly containing escaped quotes*
edgeop	→ **–>** *... in directed graphs,*
	→ **––** *... in undirected graphs*

Table C.1: Original dot file grammar from [Nor93] with adaptions.

JMFGraph introduces a few alterations listed in Table C.2. Some of them, like comma separated attribute lists and comments, were noticed in files available from the Graphviz homepage. Attribute lists may use a comma as separator. Graphs as well as digraphs may contain both typs of edges. C-like comments are supported. Ignored dot grammar features are also listed in Table C.2. Unsupported features are ignored and cause no errors.

For convenience, the text files containing the dot graph information should be named with the file extension ".dot". Files with names not conforming this suggestion can be opened by unselecting the corresponding file filter in the file chooser dialog. An example dot file is listed in Figure C.1 showing the source code for the graph shown in Figures 7.2 and 7.3.

```
/* Sample graph JWF1 taken from */
/* www.graphviz.org */

/* Sub-graph edges have been replaced */
/* by standard edges */

digraph world {
size="7,7";
    {rank=same; S8 S24 S1 S35 S30;}
    {rank=same; T8 T24 T1 T35 T30;}
    {rank=same; 43 37 36 10 2;}
    {rank=same; 25 9 38 40 13 17 12 18;}
    {rank=same; 26 42 11 3 33 19 39 14 16;}
    {rank=same; 4 31 34 21 41 28 20;}
    {rank=same; 27 5 22 32 29 15;}
    {rank=same; 6 23;}
    {rank=same; 7;}

    S8 -> 9;
    S24 -> 25;
    S24 -> 27;
    S1 -> 2;
    S1 -> 10;
    S35 -> 43;
    S35 -> 36;
    S30 -> 31;
    S30 -> 33;
    9 -> 42;
    9 -> T1;
    25 -> T1;
    25 -> 26;
    27 -> T24;
/*  2 -> {3 ; 16 ; 17 ; T1 ; 18} */
    2 -> 3;
    2 -> 16;
    2 -> 17;
    2 -> T1;
    2 -> 18;
/*  10 -> { 11 ; 14 ; T1 ; 13; 12;} */
    10 -> 11;
    10 -> 14;
    10 -> T1;
    10 -> 13;
    10 -> 12;
    31 -> T1;
    31 -> 32;
    33 -> T30;
    33 -> 34;
    42 -> 4;
    26 -> 4;
    3 -> 4;
    16 -> 15;
    17 -> 19;
    18 -> 29;
    11 -> 4;
    14 -> 15;
/*  37 -> {39 ; 41 ; 38 ; 40;} */
    37 -> 39;
    37 -> 41;
    37 -> 38;
    37 -> 40;
    13 -> 19;
    12 -> 29;
    43 -> 38;
    43 -> 40;
    36 -> 19;
    32 -> 23;
    34 -> 29;
    39 -> 15;
    41 -> 29;
    38 -> 4;
    40 -> 19;
    4 -> 5;
/*  19 -> {21 ; 20 ; 28;} */
    19 -> 21;
    19 -> 20;
    19 -> 28;
/*  5 -> {6 ; T35 ; 23;} */
    5 -> 6;
    5 -> T35;
    5 -> 23;
    21 -> 22;
    20 -> 15;
    28 -> 29;
    6 -> 7;
    15 -> T1;
    22 -> T35;
    22 -> 23;
    29 -> T30;
    7 -> T8;
    23 -> T24;
    23 -> T1;
}
```

Figure C.1: Dot source file for the sample graph JWF1 shown in 7.2 and 7.3. The commented-out lines are replaced by multiple edge definition lines, because the current parser implementation does not support the subgraph construct. It shows the “dynamic world model” from J. W. Forrester which has become a popular sample graph.

attr-list → *id* **=** *id* [**,** *attr-list*]	
edgeop → **– >** | **– –**	
/* *comment* ***/**	*everything enclosed by these terminals is ignored*
ignored	*the following features are ignored:*
	– strict
	– subgraphs
	– attr-stmt
	– attr-list, opt-attrs
	– ports

Table C.2: JMFGraph modifications to the dot file grammar.

terminals	*are written upright and bold*
nonterminals	*are written italic and gray*
[optional]	*double-line brackets enclose optional items*
⟨*grouping*⟩	*angle brackets denote grouping where necessary*
| *alternatives* |	*alternatives are separated by vertical bars*

Table C.3: Notation for dot file grammar in Table C.1 and Table C.2.

Bibliography

[ABW98] James Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external memory graph algorithms. In *European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, pages 332–343. Springer-Verlag, 1998.

[AC96] Mark D. Apperley and Michael Chester. Tree browsing. Working Paper 13, The University of Waikato, Department of Computer Science, Private Bag 3105, Hamilton, New Zealand, July 1996.

[AGD97] AGD-library, a library of algorithms for graph drawing. Algorithmic Solutions, 1997. `http://www.algorithmic-solutions.com`.

[AGKN99] James Abello, Emden R. Gansner, Eleftherios Koutsofios, and Stephen C. North. Large scale network visualization. *SIGGRAPH Newsletter*, 33(3):13–15, August 1999.

[AK00] James Abello and Jeffrey Korn. Visualizing massive multi-digraphs. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2000)*, 2000.

[And96] Keith Andrews. Information systems and the internet. In Hermann Maurer, editor, *HyperWave: The Next Generation Web Solution*, chapter 2, pages 9–18. Addison-Wesley, May 1996. `http://www.iicm.edu/hgbook`.

[APW96] Keith Andrews, Michael Pichler, and Peter Wolf. Towards rich information landscapes for visualising structured web spaces. In *Proc. 2nd IEEE Symposium on Information Visualization (InfoVis'96)*, pages 62–63, San Francisco, California, October 1996. `ftp://ftp.iicm.edu/pub/papers/ivis96.pdf`.

[AWP97] Keith Andrews, Josef Wolte, and Michael Pichler. Information Pyramids: A new approach to visualising large hierarchies. In *IEEE Visualization'97, Late Breaking Hot Topics Proc.*, pages 49–52, Phoenix, Arizona, October 1997. `ftp://ftp.iicm.edu/pub/papers/vis97.pdf`.

[BdBD00] Paola Bertolazzi, Giuseppe di Battista, and Walter Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE Transactions on Computers*, 49(8):826–840, August 2000.

[BdBL95] Paola Bertolazzi, Giuseppe di Battista, and Giuseppe Liotta. Parametric graph drawing. *IEEE Transactions on Software Engineering*, 21(5):662–673, August 1995.

[BGT96] S. Bridgeman, Ashim Garg, and Roberto Tamassia. A graph drawing and translation service on the www. In *Proceedings of the Symposium on Graph Drawing* [GD'96].

[BH86] J. Barnes and P. Hut. A hierarchical $\mathcal{O}(n \log n)$ force calculation algorithm. *Nature*, 324, December 1986.

[BJL00] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. A fast layout algorithm for k-level graphs. In *Proceedings of the Symposium on Graph Drawing* [GD'00], pages 229–240.

[BMN00] Ulrik Brandes, M. Scott Marshall, and Stephen C. North. Graph data workshop report. In *Proceedings of the Symposium on Graph Drawing* [GD'00], pages 407–409.

[BP90] Karl-Friedrich Böhringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of CHI'90*, pages 43–51, 1990.

[BPSM98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Recommendation, World Wide Web Consortium, February 1998. `http://www.w3.org/TR/REC-xml.html`, `http://www.w3.org/XML/`.

[Bra94] Franz J. Brandenburg. Designing graph drawings by layout graph grammars. In *Proceedings of the Symposium on Graph Drawing* [GD'94], pages 266–269.

[Cal01a] Chris K. Caldwell. Graph theory glossary. The University of Tennessee an Martin, 2001. Brief explanations of the most important terms. Useful for beginners. `http://www.utm.edu/departments/math/graph/glossary.html`.

[Cal01b] Chris K. Caldwell. Graph theory tutorials. The University of Tennessee an Martin, 2001. List of tutorials concerning particular graph theoretic problems. `http://www.utm.edu/departments/math/graph/`.

[Cal01c] Chris K. Caldwell. An interactive introduction to graph theory. The University of Tennessee at Martin, 2001. Interactive course which requires registration. `http://www.utm.edu/cgi-bin/caldwell/tutor/departments/math/graph/intro`.

[CdBT+92] R. F. Cohen, Giuseppe di Battista, Roberto Tamassia, Ioannis G. Tollis, and Paola Bertolazzi. A framework for dynamic graph drawing. In *Proceedings of the eighth Annual Symposium on Computational Geometry, Berlin, Germany*, pages 261–270. ACM, 1992.

[CEG98] Stuart K. Card, Stephen G. Eick, and Nahum Gershon. Information visualization. CHI'98 Tutorial, 1998. `http://turing.acm.org/sigchi/chi98/cp/?show=103`.

[CG72] E. Coffman and R. Graham. Optimal scheduling for two-processor systems. In *Acta Informatica 1*, pages 200–213, 1972.

[Cha01] Matthew Chalmers. Information visualisation. University of Glasgow, 2001. Papers for 6 parts course "Information Visualisation". `http://www.dcs.gla.ac.uk/~matthew/lectures/IS3/`.

[Che01] Bill Cherowitzo. Graph and digraph glossary. University of Colorado at Denver, 2001. Presentation of most important graph theoretic terms. `http://www-math.cudenver.edu/~wcherowi/courses/m4408/glossary.htm`.

[CM89] Mariano P. Consens and Alberto O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *Hypertext '98 Proceedings*, pages 269–292, November 1989.

[CMS98] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufman, April 1998. ISBN 1-558-60533-9.

[Coh97] Jonathan D. Cohen. Drawing graphs to convey proximity, an incremental arrangement method. *ACM Transactions on Computer-Human Interaction*, 4(3):197–229, September 1997.

[CS96] Robert Cimikowski and Paul Shope. A neural-network algorithm for a graph layout problem. *IEEE Transactions on Neural Networks*, 7(2):341–345, March 1996.

[CT01] Isabel F. Cruz and Roberto Tamassia. Tutorial on graph drawing. Tufts University / Brown University, 2001. Introduction on how to visualize a graph in two parts: The algorithmic approach (Tamassia) and the declarative approach (Cruz). `http://www.cs.brown.edu/people/rt/gd-tutorial.html`.

[daV92] daVinci Presenter, an X-Window visualization tool for drawing directed graphs automatically in high quality. b-novative, 1992. `http://www.daVinci-Presenter.de/`.

[dBETT94] Giuseppe di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Algorithms for Drawing Graphs: An annotated Bibliography*. Brown University, June 1994. Comprehensive overview of graph drawing approaches, algorithms, and literature. `ftp://wilma.cs.brown.edu/pub/papers/compgeo/gdbiblio.*`.

[dBETT99] Giuseppe di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999. ISBN 0-13-301615-3.

[dBGL95] Giuseppe di Battista, Ashim Garg, and Giuseppe Liotta. An experimental comparison of three graph drawing algorithms. In *Computational Geometry, Vancouver, B.C. Canada*, volume 11, pages 306–315. ACM, 1995.

[dBTT98] Giuseppe di Battista, Roberto Tamassia, and Ioannis G. Tollis. Area requirement and symmetry display in drawing graphs. In *Proceedings of the fifth annual symposium on Computational geometry*, pages 51–60. ACM, 1998.

[dBvR95] G. A. M. de Bruyn and O. S. van Roosmalen. Drawing execution graphs by parsing. In *Proceedings of the Third Workshop on Parallel and Distributed Real-Time Systems*, pages 113–122. IEEE, 1995.

[DG98] David P. Dobkin and Emden R. Gansner. A path router for graph drawing. In *Symp. Computational Geometry (SCG'98) - Minneapolis, Minnesota, USA*, pages 415–416. ACM, 1998.

[DGKN97] David Dobkin, Emden R. Gansner, Eleftherios Koutsofios, and Stephen C. North. Implementing a general-purpose edge router. In *Proceedings of the Symposium on Graph Drawing* [GD'97]. ISBN 3-540-63938-1.

[DH96] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, October 1996.

[Ead84] Peter Eades. A heuristic for graph drawing. In *Congressus Numerantium 42*, pages 149–160, 1984.

[EFK00] Markus Eiglsperger, Ulrich Fößmeier, and Michael Kaufmann. Orthogonal graph drawing with constraints. In *Proceedings of the eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 3–11, 2000.

[ES90] Peter Eades and Kozo Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–437, 1990.

[EW93] Stephen G. Eick and G. Wills. Navigating large networks with hierarchies. In *Proceedings of Vis'93*, pages 204–210. IEEE, 1993.

[F+00] D. Ferraiolo et al. Scalable vector graphics (SVG) 1.0 specification. Working Draft, World Wide Web Consortium, March 2000. `http://www.w3.org/Graphics/SVG/Overview.html`.

[Fai93] Kim Michael Fairchild. Information management using virtual reality-based visualizations. In Alan Wexelblat, editor, *Virtual Reality: Applications and Explorations*, chapter 3, pages 45–74. Academic Press, 1993. ISBN 0-12-745045-9.

[FHH+90] M. Froman, T. Hagerup, J. Haralambides, M. Kaufnamm, F. T. Leighton, A. Simvonis, E. Welzl, and G. Woeginger. Drawing graphs in the plane with high resolution. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, volume 1, pages 86–95. IEEE, 1990.

[FPF88] Kim Michael Fairchild, Steven E. Poltrock, and George W. Furnas. SemNet: Three-dimensional representations of large knowledge bases. In Raymonde Guindon, editor, *Cognitive Science and its Applications for Human-Computer Interaction, Hillsdale, New Jersey*, pages 201–233. Lawrence Erlbaum, 1988. ISBN 0-12-745045-9, `http://panda.iss.nus.sg:8000/kids/fair/webdocs/semnet/semnet-1.html`.

[FS91] George W. Furnas and Ben Shneiderman. TreeMaps: A space-filling approach to thte visualisation of herarchical information. In *Proceedings of CHI'91*, pages 401–408. ACM, 1991.

[Fur98] George W. Furnas. Course: SI 619 "Information Visualization". University of Michigan, 1998. `http://madison.si.umich.edu/Courses/98F619/619CourseInfo.html`.

[FW94] M. Fröhlich and M. Werner. Demonstration of the interactive graph-visualization system daVinci. In *Proceedings of the Symposium on Graph Drawing* [GD'94], pages 266–269.

[FZ94] George W. Furnas and J. Zacks. Multitrees: Enriching and reusing hierarchical structure. In *Proceedings of CHI'94*, pages 330–336. ACM, 1994.

[GD'94] GD'94. *Proceedings of the Symposium on Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*. Springer, 1994.

[GD'96] GD'96. *Proceedings of the Symposium on Graph Drawing*, volume 1190 of *Lecture Notes in Computer Science*, Berkley, California, USA, September 1996. Springer. ISBN 3-540-62495-3.

[GD'97] GD'97. *Proceedings of the Symposium on Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, Rome, Italy, September 1997. Springer. ISBN 3-540-63938-1.

[GD'99] GD'99. *Proceedings of the Symposium on Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, Stirin Castle, Czech Republic, September 1999. Springer. ISBN 3-540-66904-3.

[GD'00] GD'00. *Proceedings of the Symposium on Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, Colonial Williamsburg, VA, USA, September 2000. Springer. ISBN 3-540-41554-8, `http://www.cs.virginia.edu/~gd2000/`.

[GD'01] GD'01. *Proceedings of the Symposium on Graph Drawing*, Lecture Notes in Computer Science, Vienna, Austria, September 2001. Springer. `http://www.ads.tuwien.ac.at/gd2001/`.

[GDT98] GDToolkit. University of Rome, August 1998. `http://www.dia.uniroma3.it/~gdt/editablePages/main_index.htm`.

[GK88] David Gedye and Randy Katz. Browsing in chip design database. In *25th ACM/IEEE Design Automation Conference*, pages 269–274, 1988. paper 20.3.

[GNV88] Emden R. Gansner, Stephen C. North, and Kiem-Phong Vo. DAG—a program that draws directed graphs. *Software-Practice and Experience*, 18(11):1047–1062, November 1988.

[GSBM00] Wolfgang Günther, Robby Schönfeld, Bernd Becker, and Paul Molitor. k-layer straightline crossing minimization by speeding up sifting. In *Proceedings of the Symposium on Graph Drawing* [GD'00], pages 252–258.

[GT98] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. Wiley, 1998. ISBN 0-471-19308-9.

[GV99] Graphviz. AT&T Labs - Research, 1999. `http://www.research.att.com/sw/tools/graphviz/`, `http://www.graphviz.org/`.

[Him96a] Michael Himsolt. GML: A portable graph file format. Technical report, Universität Passau, 1996.

[Him96b] Michael Himsolt. The Graphlet system. In *Proceedings of the Symposium on Graph Drawing* [GD'96], pages 233–240.

[Him97] Michael Himsolt. GML—Graph Modelling Language, 1997. `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/`.

[HK99] Patrick Healy and Ago Kuusik. Vertex-Exchange Graph: A new concept for multi-level crossing minimisation. In *Proceedings of the Symposium on Graph Drawing* [GD'99], pages 205–216.

[HM00] Ivan Herman and M. Scott Marshall. GraphXML—an XML-based graph description format. In *Proceedings of the Symposium on Graph Drawing* [GD'00], pages 52–62. `http://www.cwi.nl/InfoVisu/GraphXML/`.

[HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, January 2000.

[HMM01] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualisation and navigation in information visualisation. Centre for Mathematics and Computer Sciences (CWI), Amsterdam, 2001. A survey on graph visualization and navigation techniques, as used in information visualization. `http://www.cwi.nl/InfoVisu/Survey/StarGraphVisuInInfoVis.html`.

[HS94] David Harel and Meir Sardas. Randomized graph drawing with heavy-duty preprocessing. In *ACM VI'94, Bari, Italy*, pages 19–33, June 1994.

[Kan92] Goos Kant. Drawing planar graphs using the lmc-ordering. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 101–110, 1992.

[KGC89] S. M. Kaplan, S. K. Goering, and R. H. Cambell. Specifying concurrent systems with D-grammars. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 20–27. Society Press, 1989.

[KY93] H. Koike and H. Yoshihara. Fractal approaches for visualising huge hiererachies. In *Proceedings of the Symposium on Visual Languages (VL'93)*. IEEE, 1993.

[LA94] Y. K. Leung and Mark D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126–160, 1994.

[LE98] Wei Lai and Peter Eades. Routing drawings in diagram displays. In *Proceedings of the 3rd Asia Pacific Symposium on Computer Human Interaction*, pages 291–296, 1998.

[LED90] LEDA—Library of Efficient Data types and Algorithms. Algorithmic Solutions, 1990. `http://www.algorithmic-solutions.com`.

[M+01] Petra Mutzel et al. Automatic graph drawing. Vienna University of Technology, 2001. Introduction to different fields of application for graph drawing and their respective demands. `http://www.apm.tuwien.ac.at/research/graphDrawing.html`.

[Man01] Visualization course material. University of Manchester, University of Edinburgh, 2001. Collection of course supplements. `http://www.mcc.ac.uk/hpc/cluster_computing/IMPACT/training/visualisatio%n/`.

[Mau96] Hermann Maurer, editor. *HyperWave: The Next Generation Web Solution*. Addison-Wesley, May 1996. `http://www.iicm.edu/hwbook`.

[McC97] C. McCreary. *Visualizing Graphs with Java (VGJ) Manual*, 1997. `http://www.eng.auburn.edu/department/cse/research/graph_drawing/manual/vgj_manual.html`.

[MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:183–210, 1995.

[MGR99] Tamara Munzner, François Guimbretiére, and George G. Robertson. Constellation, a visualization tool for linguistic queries from MindNet. In *Proceedings of the IEEE Symposium on Information Visualization (InfoViz'99)*, pages 132–135, 154, 1999.

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

[MP92] Seth Malitz and Achilleas Papakostas. On the angular resolution of planar graphs. In *ACM STOC - Victoria, B.C., Canada*, volume 24, pages 527–538, 1992.

[MRC91] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The Perspective Wall: Detail and context smoothly integrated. In *Proceedings of CHI'91*, pages 173–179. ACM, 1991.

[MSM99] Christian Matuszewski, Robby Schönfeld, and Paul Molitor. Using Sifting for k-layer straightline crossing minimization. In *Proceedings of the Symposium on Graph Drawing* [GD'99], pages 217–224.

[Mun97] Tamara Munzner. H3, laying out large directed graphs in 3D hyperbolic space. In *Proceedings of the IEEE Symposium on Information Visualization (InfoViz'97)*, pages 2–10, 114, 1997.

[NIS01] Dictionary of algorithms, data structures, and problems. National Institute of Standards and Technology (NIST), 2001. Large list of explained graph theory terms. `http://hissa.nist.gov/dads/terms.html`.

[Nor93] Stephen C. North. Dot abstract graph description format, August 1993. `http://www.research.att.com/~north/cgi-bin/webdot.cgi/dot.txt`.

[OW88] R. H. Overmars and E. Welzl. New methods for computing visibility graphs. In *Proceedings of the Symposium on Computational Geometry*, volume 4, pages 164–171. ACM, 1988.

[PdBA+92] J. Paredaens, J. Van den Bussche, M. Andries, M. Gyssens, and I. Thyssens. An overview of GOOD. *ACM SIGMOD Record*, 21(1):25–31, March 1992.

[PSS94] Jànos Pach, Farhad Shahrokhi, and Mario Szegedy. Applications of the crossing number. In *Computational Geometry, Stony Brook, NY, USA*, volume 10, pages 198–202. ACM, June 1994.

[PT98] Achilleas Papakostas and Ioannis G. Tollis. Interactive orthogonal graph drawing. *IEEE Transactions on Computers*, 47(11):1279–1309, November 1998.

[Pur98] Helen C. Purchase. The effects of graph layout. In *Proceedings of the Australasian Computer Human Interaction Conference*, pages 80–86, 1998.

[RDM+87] Lawrence A. Rowe, Michael Davis, Eli Messinger, Carl Meyer, Charles Spirakis, and Allen Tuan. A browser for directed graphs. *Software-Practice and Experience*, 17(1):61–76, January 1987.

[RGHC97] P. J. Rodgers, R. Gaizauskas, K. Humphreys, and H. Cunningham. Visual execution and data visualisation in natural language processing. In *Proceedings of the Symposium on Visual Languages (VL'97)*, pages 342–347. IEEE, 1997.

[RK97] P. J. Rodgers and P. J. H. King. A graph rewriting visual language for database programming. *Journal of Visual Languages and Computing*, 8(6):641–674, December 1997.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: Animated 3D visualizations of hierarchical information. In *Proceedings of CHI'91*, pages 189–194. ACM, May 1991.

[RMS97] Kathy Ryall, Joe Marks, and Stuart Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of the 11th annual ACM Symposium on User Interface Software and Technology, Banff, Alberta, Canada (UIST'97)*, pages 97–104. ACM, 1997. `http://www.cs.virginia.edu/~glide/`.

[Rod98] P. J. Rodgers. A graph rewriting programming language for graph drawing. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 32–39, 1998.

[Sas01] Graph theory: An introduction. University of Saskatchewan: Department of Computer Science, 2001. Graph theory introductions split up into parts for, both, beginners and advanced students. `http://www.cs.usask.ca/resources/tutorials/csconcepts/graphs/tutorial/`.

[SB92] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In *Proceedings of CHI'92*, pages 83–91. ACM, 1992.

[Sch94a] Robert Schreiber. The Barnes-Hut algorithm. RIACS, 1994. `http://www.npac.syr.edu/hpfa/hpff2/html/section2_6_1.html`.

[Sch94b] A. Schürr. Rapid programming with graph rewrite rules. In *Proceedings of the Symposium on Very High Level Languages (VHLL), Santa Fe*, pages 83–100. USENIX, October 1994.

[Sch97] A. Schürr. BLD—a nondeterministic data flow programming language with backtracking. In *Proceedings of the Symposium on Visual Languages (VL'97)*, pages 398–405. IEEE, 1997.

[Sch98] Jürgen Schipflinger. The design and implementation of the Harmony session manager. Master's thesis in Telematics, Graz University of Technology, Institute for Information Processing and Computer Supported New Media, December 1998.

[Sha98] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis, Java Edition*. Prentice-Hall, 1998. ISBN 0-13-660911-2.

[Shn92] Ben Shneiderman. Tree visualization with Tree-Maps: 2-d space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. 1996 IEEE Symposium on Visual Languages*, pages 336–343, Boulder, Colorado, September 1996. IEEE Computer Society. `ftp://ftp.cs.umd.edu/pub/papers/papers/3665/3665.ps.Z`.

[Shn97] Ben Shneiderman. Course: CMSC 828/838 "Information Visualization". University of Maryland, 1997. `http://www.otal.umd.edu/Olive/Class/`.

[SM90] Kozo Sugiyama and Kazuo Misue. "good" graphics interfaces for "good" idea organizers. In *Third International Conference on Human-Computer Interaction, Cambridge, UK*, volume IFIP TC13, pages 27–31, August 1990.

[SM91] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, June 1991.

[SM95] Kozo Sugiyama and Kazuo Misue. Graph drawing by the magnetic spring model. *Journal of Visual Languages and Computing*, 6:217–231, 1995.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, February 1981.

[Tat97] Junichi Tatemura. Visualizing document space by force-directed dynamic layout. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 119–120, 1997.

[TdBB88] Roberto Tamassia, Giuseppe di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, January 1988.

[Tol96] Ioannis G. Tollis. Graph drawing and information visualization. In *ACM Computing Surveys*, volume 28A, 1996. `http://www.utdallas.edu/~tollis/SDCR96/TollisGeometry/`.

[Tom01] Tom Sawyer graph layout toolkit. Tom Sawyer Software, 2001. `http://www.tomsawyer.com/glt/index.html`.

[TS01] Tom Sawyer glossary. Tom Sawyer Software, 2001. Very brief explanations in short words. `http://neoteny.eccosys.com/~deivu/portfolio/tom/before/glossary.html`.

[TTV91] Roberto Tamassia, Ioannis G. Tollis, and Jeffrey S. Vitter. Lower bounds and parallel algorithms for planar orthogonal grid drawings. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 386–393. IEEE, 1991. (Extended Abstract).

[Wal98] John Wallin. Description of the Barnes-Hut algorithm. George Manson University, Physics & Astronomy, 1998. `http://www.physics.gmu.edu/~large/lr_forces/desc/bh/bhdesc.html`.

[Wei98] Mark Allen Weiss. *Data Structures and Problem Solving using Java*. Addison-Wesley, 1998. ISBN 0-201-54991-3.

[WM99] Vance Waddle and Ashok Malhotra. An $e \log e$ line crossing algorithm for levelled graphs. In *Proceedings of the Symposium on Graph Drawing* [GD'99], pages 59–71.

[Wol98] Josef Wolte. Information Pyramids: Compactly visualising large hierarchies. Master's thesis in Telematics, Graz University of Technology, Institute for Information Processing and Computer Supported New Media, 1998.

[XSL01] The w3c's xsl page, 2001. `http://www.w3.org/Style/XSL/`.