The Graph Visualization System (GVS)

A Flexible Java Framework for Graph Drawing

Wolfgang Prinz

The Graph Visualization System (GVS)

A Flexible Java Framework for Graph Drawing

Master's Thesis

at

Graz University of Technology

submitted by

Wolfgang Prinz

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

21st March 2006

Advisor: Ao.Univ.-Prof. Dr. Keith Andrews

Das Graph Visualization System (GVS)

Ein flexibles Java Gerüst zum Zeichnen von Graphen

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Wolfgang Prinz

Institut für Informationssysteme und Computer Medien (IICM),
Technische Universität Graz
A-8010 Graz

21. März 2006

Diese Arbeit ist in englischer Sprache verfasst.

Betreuer: Ao.Univ.-Prof. Dr. Keith Andrews

TUG Technische Universität Graz
Erzherzog-Johann-Universität

Abstract

A graph describes relationships between entities and is usually represented by a set of nodes (entities) and a set of edges (relations) between the nodes. Metadata such as labels or weights are often associated with the elements of a graph.

The field of graph drawing, part of the wider field of information visualization, seeks to visualize the abstract information contained within a graph for the human observer. A drawing of a graph is a graphical image, in two or three dimensions, which reflects the graph's topology and characteristics as closely as possible. Applications of graph drawing include social network and web site visualization, transportation network maps, and document cluster analysis.

The Graph Visualization System (GVS) is a modular, flexible, and extensible framework for graph drawing implemented in Java. GVS provides implementations of some of the standard layered and force-directed graph drawing techniques. In addition, GVS is designed to be used as a demonstrator tool when teaching graph drawing methods. To this end, each of the implemented algorithms is divided into its constituent parts, which can be stepped through (and undone) interactively.

Kurzfassung

Ein Graph beschreibt Beziehungen zwischen Elementen und besteht gewöhnlich aus einer Menge an Knoten (Elemente) und einer Menge an Kanten (Beziehungen) zwischen diesen Knoten. Oft werden durch Metadaten zusätzlich Beschriftungen oder Gewichte an Elementen des Graphen angebracht.

Das Visualisieren von Graphen, als Teilgebiet der Informationsvisualisierung, versucht die in Graphen enthaltene abstrakte Information für den menschlichen Betrachter aufzubereiten. Das Bild eines Graphen kann sowohl zwei- als auch dreidimensional sein. Es versucht die Topologie des Graphen und dessen Charakteristik wiederzugeben. Anwendungen hierfür sind die Visualisierung von sozialen Netzwerken oder Internetseiten, die Darstellung von Verkehrslinien oder die Analyse von Dokumentclustern.

Das Graph Visualization System (GVS) ist ein modulares, flexibles und erweiterbares Java Programmgerüst zur Visualisierung von Graphen. GVS beinhaltet bereits die Implementierungen einiger hierarchischer sowie einiger "force-directed" Techniken. Zusätzlich wurde GVS als Demonstrationsprogramm zur Veranschaulichung jener Techniken für den Lehrbetrieb entwickelt. Diesem Rechnung tragend kann jeder Algorithmus durch Interaktion mit dem Anwender schrittweise abgearbeitet und analysiert werden.

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.

Acknowledgments

First of all I would like to thank my parents Horst and Ingrid for their unrestricted and life-long support. In addition, I want to thank my sister Christine for being patient with me all the time and enduring my, in her opinion, boring talks about graph drawing and mathematics. Further thanks go to Harald Axmann, Edmund Ehrlich, Marco Haid, and of course Helmut Pauer (now Helmut Mader), who accompanied me through my study. Simon Jantscher has to be thanked for politely asking about the progress of my work from time to time. I want to thank my supervisor Keith Andrews for his support and constructive criticism during my work. The appearance of the user interface of GVS was particularly influenced for the better through his advice.

This work is dedicated to my old school friend, Christian Perktold, who died tragically in an avalanche in March 2006.

Wolfgang Prinz
Graz, Austria, March 2006

Credits

- Figure 1.1 is an excerpt from the "Grazer Straßenbahnlinienübersicht mit Umsteigemöglichkeiten" provided by the Grazer Verkehrsbetriebe (public transport systems of Graz) via their web page http://www.gvb.at/home/z-files/Plan.pdf.
- Figure 1.3 was taken from http://en.wikipedia.org/wiki/Image:Minard.png.
- All screenshots presented in this work were made from local installations of the corresponding software and were created by the author himself.

Contents

List of Figures

Chapter 1

Introduction

" When you make a journey, always start with the first step. "

[Proverb.]

In their most general form, graphs describe relationships of any kind between entities of any kind. Graphs are very common not only in science but also in nearly every part of everyday life. Finding meaningful ways of expressing graphs and transporting the information they contain is key. The process of creating a graphical representation of the information in a graph is called *graph drawing*. This chapter introduces the field of graph drawing. Applications for graphs and graph drawing in general are described and the importance of automatic graph drawing is discussed. Furthermore, the embedding of graph drawing in the wider field of information visualization is presented and the scope of this thesis is defined.

1.1 Applications of Graphs and Graph Drawings

Describing relationships between entities, and thereby describing these entity-relationships by graphs, is a very common feature in computer science and indeed all fields of science. For example, they are typically used to describe database architectures. A hierarchically structured sub-form of entity-relationships, the tree, is omnipresent in file system explorers. Trees and their graphical representations can be found in many computer programs. Another important data structure, the sequential list, is in fact a degenerated tree and hence a graph as well. The graphical representations of trees and lists are the foundation of every modern Graphical User Interface (GUI) (see [Sun, 2001a] and [Andrews, 2006a]).

Another, more concrete example for the applicability of general graphs are computer networks. The World Wide Web (WWW) is a very large assembly of server entities connected to each other by the internet (see Section 3.5). In psychology, social networks describe relationships between people or groups of people (see Section 3.9.2). Biological networks link biological properties of a sample together in order to find out their overall connections. Many further forms of networks are found in many sciences. The actual distinction between a graph and a network, as understood in this thesis, is only that a network usually has an associated, well defined semantic. Because of their close affinity, graphs and respectively their drawings can easily be used to describe networks.

Entity-relationship models are not only restricted to computer science or the sciences, but can be found in nature and everyday life as well. A typical example are train network maps (see [Spence, 2001]). In contrast to a typical street map, which positions elements according to their geographical location, a train network map only shows the stations and their connections. Geographic distances between train stations or the actual train tracks are not preserved, because they are of minor importance to the train user. Figure 1.1 shows the tram route map in the city of Graz. The distances between stations on the

map and the directions and bends of the routes do not reflect geographical conditions, but are deformed in order to make the whole map more compact and readable.

The examples of graphs and graph drawings presented above are only a small excerpt from the vast range of possible applications. The large number of professional software packages available for graph drawing already underlines its importance. The graph drawing packages presented in Chapter 3 are only a small excerpt themselves. Specialized software exists for applications in biology, chemistry, physics, medicine, engineering and social sciences, which use graph drawing to visualize complex information.

One direct application of graph drawing techniques is InfoSky [Andrews et al., 2002; Kienreich et al., 2003]. InfoSky displays the contents of a structured information space, which could, for example, be a collection of documents, an encyclopedia, or a news archive. Using the metaphor of a galaxy of stars in the sky, each information entity, for example a document, is represented by a single star in the InfoSky universe. Related entities form star systems, related star systems form galaxies and so forth. The user can navigate through this information galaxy like an astronomer would explore the real universe. The information galaxy is an example of graph drawing by force-directed placement (see [Kienreich et al., 2003] and Section 5.1).

Figure 1.2 shows the InfoSky demo application [Know-Center, 2006]. The dataset visualized by the InfoSky demo application is the "Computers" branch of the `dmoz` open directory project (see `http://dmoz.org`). This particular branch consists of over 140.000 documents organized by a structure of over 7.000 directories. InfoSky uses nine levels of hierarchy to visualize this amount of data. It took two hours to calculate the force-directed layout for this large dataset. The user can explore the information galaxy interactively by browsing into the star systems that represent the different topics.

For many applications, it is important to provide insight into the graph or network's structure for the human user. It is desirable that important features, central players, and key terms can be comprehended easily at a glance. Unfortunately, very dense graphs and networks tend to become unmanageable and unreadable, because too much information is presented on too small an area. On the other hand, if the drawn image is too large, the viewer will find it difficult to perceive the information. This already introduces one of the most severe problems in graph drawing: the issue of scalability (see Section 2.3). In InfoSky, this problem is solved by zooming into the information galaxies [Kienreich et al., 2003]. Figure 1.2 shows only a small portion of the whole information universe graph — just large enough that the user can comprehend an overview of the selected topic. Presenting the whole graph in that level of detail would not fit on the display. Designing visualizations so that they can be understood easily by humans is the goal of information visualization.

1.2 Information Visualization

The examples in the previous section demonstrated the need for feasible techniques for representing large amounts of information to the user. The field of computer science involved in analyzing the interplay between humans and computer is called *Human-Computer Interaction (HCI)* [Andrews, 2006a]. HCI develops interfaces between human users and computers that take account of human psychological and physical conditions. Since the primary sense of the human body is the sense of sight, most information can be transported using this channel. The field of computer science that deals with visualizing information for the human user is called *information visualization*. Since graphs and their drawings are such a general and expressive concept, they play a key role in information visualization as well.

Information visualization transforms abstract data or concepts into more easily understood representations. The goal of information visualization is to gain insight into data or concepts that were hidden before. Often the information is hidden simply by the huge amount of data available. Thus, information visualization can also be seen as a converter between the underlying data and the human perception of it. An intuitive illustration of the power of information visualization was given by Spence [2001]:

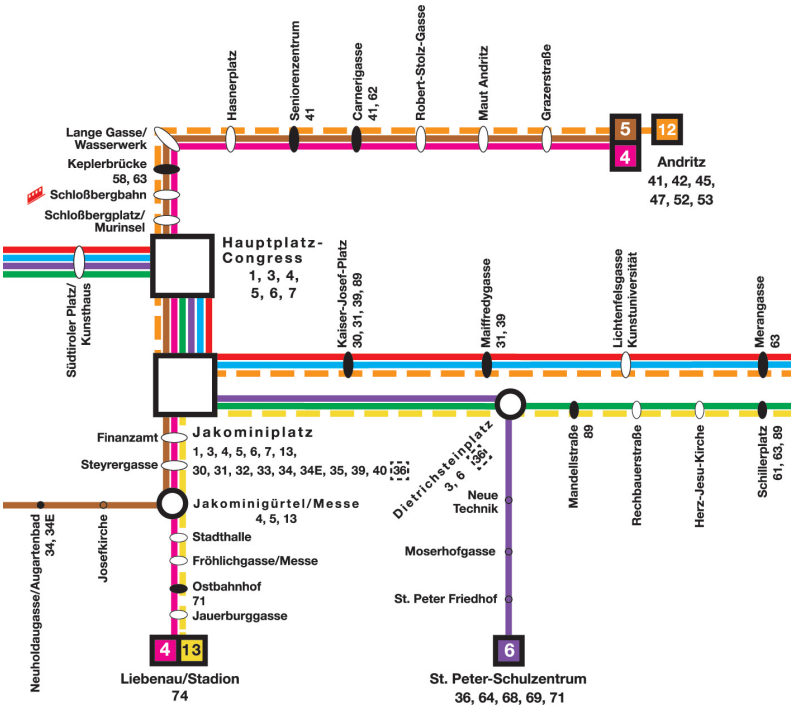


Figure 1.1: An excerpt from the Graz tram map showing the city center (source: `http://www.gvb.at`). The map reflects neither geographic properties nor distance proportions, but clearly shows the main stops and where routes cross.

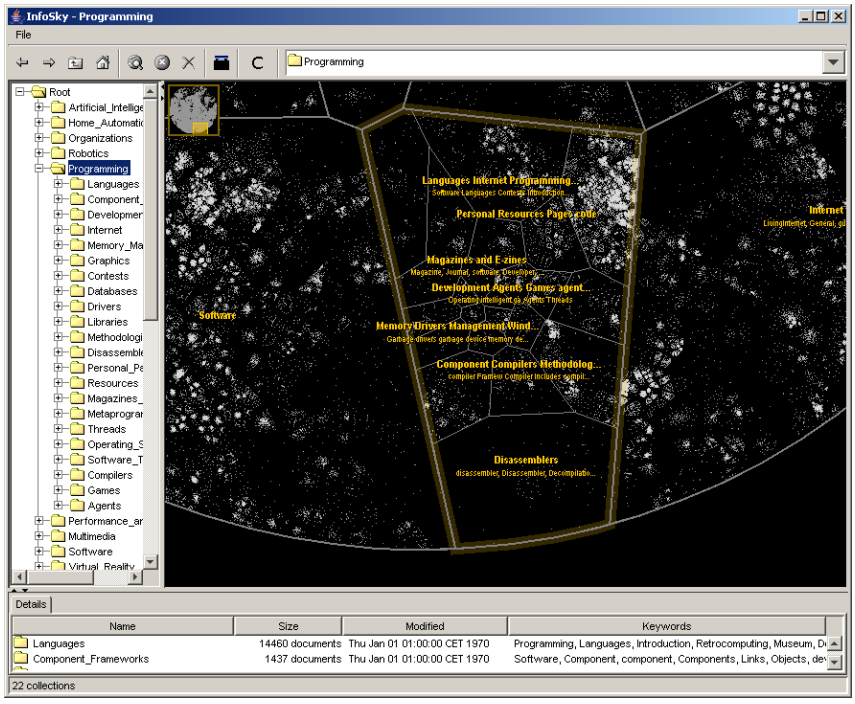


Figure 1.2: InfoSky represents documents as stars in an information galaxy (see [Andrews et al., 2002]). A demo of the system is available freely at [Know-Center, 2006]. The InfoSky demo visualizes the "Computers" branch of the `dmoz` open directory project (see `http://dmoz.org`).

> "You are the owner of some numerical data which, you feel, is hiding some fundamental relation which could be exploited to your advantage, perhaps for business or merely for pleasure. You then glance at some visual representation of that data and exclaim 'Ah ha! - *now* I understand'. This is what information visualization is about."

This insight into data is the interest of many scientific fields, which is already expressed by the word "science" itself. The word "science" comes from the Latin word "scientia", which already bears the meaning of knowledge and insight. In geophysics, for example, echo data gathered from the seabed is visualized as a three-dimensional image so that the human viewer can see how the seabed is formed (see [Ware, 2004]). Another example would be the Computational Fluid Dynamics (CFD) images that visualize the flow of air or liquids around the surfaces of objects (see [Prinz, 2005]). These two examples are more accurately associated with so-called data visualization.

Data visualization also called scientific visualization deals with the visualization of potentially large datasets, which often have some physical or geometrical background [Duke, 2001]. In contrast to data visualization, information visualization makes abstract information structures visible to the human viewer [Andrews, 2006b]. These abstract information structures can be large in scale as well. For example, a typical information visualization task is to point similarities between thousands of textual documents out to the human user.

Information visualization like graph drawing is not an invention of the computer age and computer graphics. The historically earliest examples of information visualization are cave paintings [Ware, 2004]. Cave paintings visualize the process of hunting or other daily routines. Early examples of graph drawing are bloodline drawings and family trees of the 13^{th} and 14^{th} century [Kruja et al., 2001]. Interestingly, already three-dimensional graph drawings were used in the late 18^{th} century to display the composition of crystals [Kruja et al., 2001]. A more recent example of early information visualization are Florence Nightingale's (1820–1910) diagrams on death rates in hospitals [Spence, 2001], which can be seen as early pie charts. Probably one of the most referenced artistic drawings in information visualization is the 1860 illustration of Napoleon's failed invasion of Russia from 1812 to 1813 (see Figure 1.3) by Charles Joseph Minard (1791–1870).

The development of computers and electronic data processing also produced ever growing amounts of data to be understood and analyzed by humans. The rise of computers thereby drove the development of information visualization [Chen, 2004]. During the last two decades, the field of information visualization has grown to such a size that makes it difficult to overview. This is underlined by the fact that Chen [2004] uses graph drawing itself to visualize the intellectual development of information visualization. The following taxonomy of information visualization is only a rough classification with respect to the context of this thesis and must not be seen as all-embracing. More detailed explanations of information visualization can be found in [Spence, 2001], [Ware, 2004] and [Chen, 2004] as well as [Andrews, 2002] and [Andrews, 2006b]. Further classification attempts are made in [Shneiderman and Plaisant, 2004], [Shneiderman, 1996], [Nussbaumer, 2005], and [Putz, 2005]. In the context of this thesis, information visualization is divided into the following main groups:

- Visualizations of linear structures, arrays and tables.
- Visualizations of multi-dimensional metadata.
- Visualizations of content-based vector structures.
- Visualizations of hierarchies.
- Visualizations of networks.

When visualizing arrays, tables or multi-dimensional metadata structures, the dataset to be visualized consists of a series of objects each having a certain amount of attributes. Content-based vector-spaces

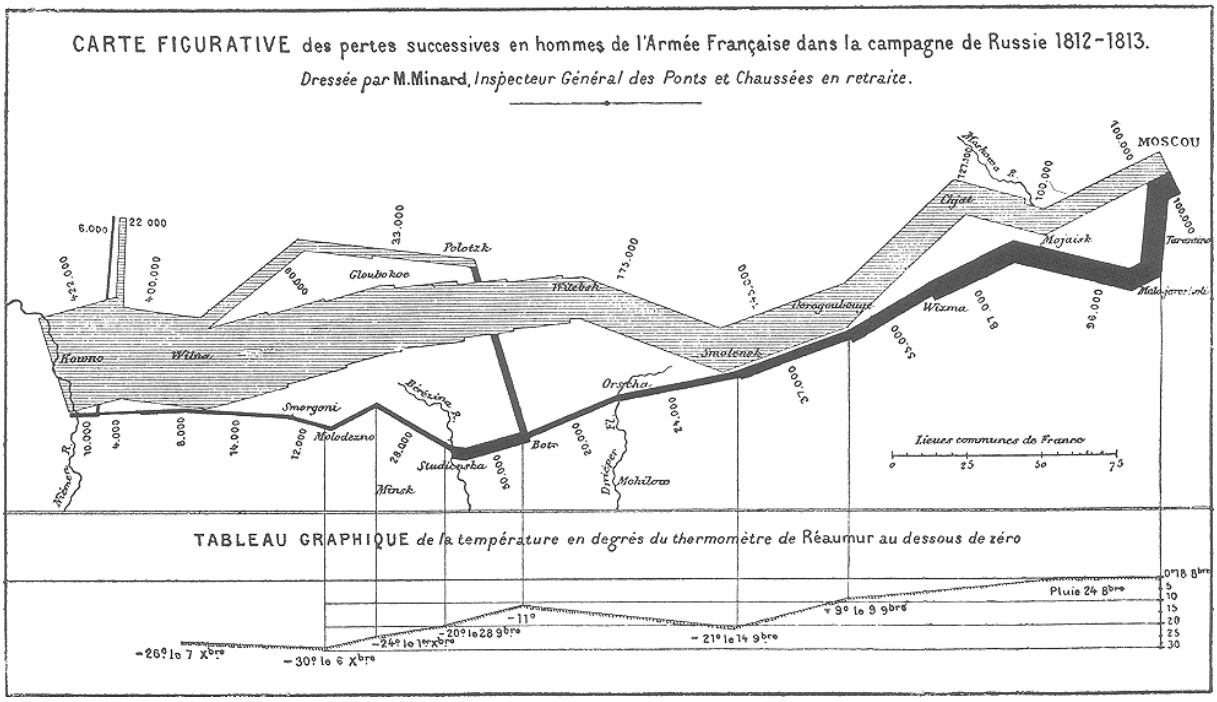


Figure 1.3: Napoleon's losses during the invasion of Russia from 1812 to 1813 illustrated by Charles Joseph Minard in 1860 (source: `http://en.wikipedia.org/wiki/Image:Minard.png`). The thickness of the upper path represents the number of soldiers in the French army during the pursuit toward Moscow. The lower path dramatically shows the fall of this number of soldiers during the retreat. Further versions of this drawing can be found in Tufte [2001], Spence [2001], and Chen [2004].

consist of a large amount of very high-dimensional vectors each characterizing the content of a certain document. The goal to be achieved is to provide the user with detailed information about every single object while at the same time providing information about all objects in general. Usually there is a struggle between these two forces when the dataset becomes larger in size. The range of solutions in this category leads from the Perspective Wall to techniques like multi-dimensional scaling (see [Mackinlay et al., 1991] and [Borg and Groenen, 2005; Cox and Cox, 2000]).

In contrast to linear structures, visualizations of hierarchies emphasize the structure that is introduced by the hierarchies. Hierarchies also define a clustering of the data. Transporting these very relationships to the user is key for hierarchical visualizations. Several ways of representing this information exist, like the traditional expand/collapse style known from file system explorers or classical planar tree drawings [Buchheim et al., 2002]. More elaborate ways of visualizing hierarchies include Tree Maps, Hyperbolic Trees and Cone Trees (see [Johnson and Shneiderman, 1991], [Lamping and Rao, 1994], and [Robertson et al., 1991]). Further information about visualizations of hierarchies can be found in [Andrews, 2002] and [Andrews, 1998].

Finally, the visualizations of networks extend the hierarchical visualizations by allowing arbitrary relationships between the entities. The important feature pointed out here is which entities are explicitly linked to which other entities. Entity-relationships are mathematically described by graphs. Visualizing graphs is the primary topic of this thesis.

1.3 Graph Drawing

The task of transforming the abstract information contained within a graph into an image appealing to and understandable by humans is called *graph drawing*. Graph drawing is heavily used in information visualization to display abstract relationships between entities. One application of graph drawing is to visualize semantic networks. Semantic networks relate entities according to their semantic meaning rather than or in addition to their physical relationships. For example, the Semantic Web is a further development of the common World Wide Web that seeks to provide semantic, machine-understandable metadata to supplement web pages (see [W3C, 2006]). This additional information reveals relationships between web pages that were not obviously related before. Graph drawing can present this information to the user for better understanding.

Another important application of graph drawing in information visualization is the hybrid combination of drawing hierarchies and graphs. The goal of this combination is to preserve the structural information provided by a hierarchy while at the same time showing the relationships between the entities. This leads to so-called layered or hierarchical graph drawing. There are many algorithms for drawing layered graphs [Sugiyama, 2002] and also an algorithm combining features of hierarchical drawings with those of unconstrained drawings [Dwyer and Koren, 2006]. Several information visualization software packages exist that use graphs and graph drawing algorithms. A brief description of these packages can be found in Section 3.10.

Drawing graphs so that they are appealing to humans is not easy to do. This comes mostly from the fact that it is difficult to define what actually appeals to humans. A graph drawing should convey as much information as possible while not overextending the human viewer's perception. Related things should be drawn close together to support the human brain's thinking in categories, while at the same time entities should still be distinguishable from each other.

Drawing of a graph by hand (such as some of the illustrations by [Tufte, 2001]) is simple as long as the graph is small and the artist can keep all details in mind while drawing. In practice, many graphs are large enough to make drawing per hand unfeasible. Hence, efforts were soon made to automate graph drawing. Especially in recent times, when large graphs are created from data collected automatically from digital sources, it is reasonable to process this data electronically as well. For example, the Doxygen source code documentation tool uses automatic graph drawing to create class diagrams from the given source code files (see Section 3.2). Drawing such graphs by hand would be far too much effort.

Of course, simply letting the computer do the graph drawing is not sufficient. It is still necessary that the images produced by automatic graph drawing are aesthetically appealing to human viewers. Graph drawing algorithms have to take these aesthetic criteria into account when drawing a graph (see Section 2.3). This demands that the aesthetic criteria are mapped to exact mathematical terms. The drawing algorithms are then optimized to meet these parameters. Unfortunately finding a mathematical formulation for some aesthetic criterion suitable for the drawing algorithm is nearly as difficult as identifying an aesthetic criterion itself.

Recent scientific research in the field of graph drawing can roughly be grouped into two directions. One direction is to improve existing algorithms with better support for aesthetic criteria (see for example [Gansner and North, 1998]) and the other is to find new algorithms for realizing the aesthetic criteria in a better way (see for example [Koren, 2005]). Nevertheless, besides satisfying the aesthetic criteria, the runtime and memory usage of an algorithm is still critical. This especially applies for huge graphs. Considering all these factors, there is currently no single technique, no single algorithm that can claim itself superior to all others. This is basically caused by the vast range of possible applications of graphs and graph drawing and by the large number of different influences on the algorithms. Hence, graph drawing is still a scientific field with many open problems to occupy the research community.

1.4 Scope of this Thesis

Graph drawing is a rather complicated process and developing graph drawing software is difficult. A fairy large amount of implementation effort has to be done before any visual results can be observed. This basically includes the loading of a graph from some storage media, representing it internally so that the algorithm can operate on it and finally displaying the computed layout of the graph to the screen. All these tasks, especially the loading and the rendering to the screen, have very little to do with graph drawing in the closer sense.

This thesis presents the Graph Visualization System (GVS), a framework for teaching, comparing and explaining graph drawing techniques and algorithms. Implementing new algorithms should be easier given the infrastructure provided by GVS. The whole GVS framework should be easy to understand and easy to learn so that student projects can be done more efficiently in future. See Section 4.1 for a more detailed description of the goals of GVS.

Furthermore, this thesis describes the fundamentals of graph drawing as needed for the implementation and understanding of GVS. A survey of current graph drawing software packages was performed to determine if there already was a package which suited the requirements of GVS (see Chapter 3). No single package matched the requirements, especially for teaching. Therefore, based on the concepts provided by the most common graph drawing packages, the GVS framework was developed. The goals of GVS as well as the causes for the various decisions made during the development are described in Chapter 4.

1.5 Structure of this Thesis

Chapter 1 introduces the topic of graph drawing generally and points out possible applications. The scope of this work is defined as well as an overview of the document is provided. The theoretic basics of graph drawing and the underlying concepts are explained in Chapter 2. Furthermore, a taxonomy of graph drawing is presented. Chapter 3 describes a selected number of common graph drawing packages. The packages are analyzed according to key features being considered as important for the later development of the GVS.

The goals of the GVS framework and its underlying concepts are documented in Chapter 4. The software design issues that arose when planning GVS are discussed as well as the important features of GVS are outlined. Chapter 5 presents selected details of the implementation that have not been described in the context of the other chapters so far. An outlook of the future development of GVS in special as well as the trends to be expected in the field of graph drawing in general is presented in Chapter 6. In conclusion, Chapter 7 summarizes the main facts of this work and states the concluding remarks and thoughts of the author.

The GVS User Guide is contained in Appendix A. The user interface and user interaction mechanism of GVS are explained in a non technical way. Technical details about GVS are included in Appendix B, the GVS Developer Guide. This guide explains how to implement a small visualization and integrate it into GVS.

Chapter 2

Graph Drawing

" Nodes and edges,
And arrowhead wedges,
That's what graphs are made of. "

[Sequel to an old nursery rhyme.]

This chapter presents the mathematical and semantic fundamentals of graph drawing. Terms commonly used within this thesis as well as within the graph drawing literature are defined and explained. Furthermore, the fundamental concepts and ideas behind graph drawing are illustrated. Since the field of graph drawing is very large, a taxonomy of graph drawing is formulated to provide better understanding.

2.1 Graph Theory

Before talking about graph drawing, the term "graph" has to be defined precisely. Fortunately, there is a common, worldwide understanding of basic graph theoretical concepts. In this section, a summary of these concepts and definitions is presented. Further details regarding graph theory can be found in [Battista et al., 1999], [Kaufmann and Wagner, 2001] and [Sugiyama, 2002].

2.1.1 Graphs

As already mentioned Chapter 1, graphs describe relationships between entities. Mathematically speaking the following sentence expresses this:

"A graph $G(V, E)$ is an abstract structure used to model a relation E over a set V of entities."

The elements of the non-empty set V are called *nodes* or *vertices*. An important feature of a graph $G(V, E)$ is that V is a set per definition. A set is a structure containing distinct and thereby distinguishable and unique elements. This uniqueness can also be stated as that no duplicate elements are allowed in a set. Of course uniqueness within a set can only be enforced if the elements are distinguishable, or in other words, if duplication can be detected. Another noticeable feature of the set V is that it imposes no explicit order of its nodes v_i, where $v_i \in V$, $1 \leq i \leq n$, $i \in \mathbb{N}$, $n \in \mathbb{N}$, although implicit order can be seen in the enumeration of the nodes v_i with $1 \leq i \leq n$.[1] This nomenclature states only an arbitrary, but unique, two-way mapping between a node v and a natural number i used to numerically identify the nodes.

[1] The enumeration of the nodes v_i may be $0 \leq i < n$, with $i \in \mathbb{N}$, $n \in \mathbb{N}$, as well without loss of generality.

The term n denotes the number of nodes in the graph $G(V, E)$, thus the size of the set V, which is $n = |V|$, where $0 < n, n \in \mathbb{N}$. The number of nodes n is often used to make statements about the size of a graph. Graphs with more than 100 nodes are considered as large. Graphs with more than 1000 nodes are denoted as huge. Nevertheless, this qualification of graphs according to the number of nodes is only an rough estimation and strongly depends on the context where it is mentioned.

The relation E describes a collection of non-empty subsets of V, each declaring these nodes as belonging together. Such subset of nodes, the elements of the collection E, are called *edges*. Edges are often enumerated in a similar manner to nodes, which is e_i, where $e_i \in E$, $1 \leq i \leq |E|$, $i \in \mathbb{N}$. The term "edge" comes from the fact that in practice, most graphs will define their relationships to be between exactly two distinct nodes. When drawing such graphs, edges will often be painted as straight lines, hence an edge is connecting two nodes. When edges connect two nodes, they are often written as $e_{i,j}$ with i and j being the numbers of the two nodes v_i and v_j this edge connects.

Note that per definition, E is modeled as a collection, which, like the set V, does not define an explicit order among the elements. In contrast to the set V, the collection E allows duplicate edges in the sense of two different edges connecting the same sets of nodes. A graph $G(V, E)$ models only a relation E over a set of nodes V, but does not specify the semantics of that relation. So there may be edges involving the same sets of nodes having different semantics. For example, in a graph which models traveling routes between cities, two neighboring cities may be connected by road as well as by train. Of course, whether this dualism is expressed by one or two edges strongly depends on the semantics and the intention of the graph drawing.

A concrete edge e, where $e \in E$, of a graph $G(V, E)$ is a non-empty subset of nodes of the set of nodes V. This edge subset of nodes has to be non-empty because a relation between no nodes makes no sense. A relation between only one single node and itself denotes a so-called *self-loop* or *self-edge*. The semantics of the graph might make self-loops necessary. For example, a graph describing postal deliveries might use self-loops to express letters returned to the sender. Self-loops could also be specified by allowing the edge subsets to be sub-collections and thereby containing duplicate elements. Nevertheless, self-loops have little influence on graph drawing (see Section 2.1.2) and are therefore mostly neglected by algorithms.

As already mentioned, most graphs will have edges with exactly two distinct nodes in practice. If a graph contains an edge e connecting more than two nodes, it is called a *hypergraph*. Accordingly, edges connecting more than two nodes are called *hyperedges*. Hypergraphs and hyperedges are very rare in graph drawing. This is confirmed by the fact that they are hardly mentioned in the two standard books on graph drawing: [Battista et al., 1999] and [Kaufmann and Wagner, 2001]. Unless otherwise mentioned, all further discussion of graphs in this thesis will assume that there are no hyperedges in the graph, thus all edges having exactly two nodes.

An important property that can be assigned to edges is the edge direction. The edge direction marks one node of the edge as the source and the other node of the edge as the destination.[2] A graph containing a directed edge is called a *directed graph*, or *digraph*. Usually, a directed graph will only have directed edges. Graphs with both directed and undirected edges can be seen as directed graphs, where each undirected edge is modeled by two directed edges pointing in the opposite direction (see edges e_3 and e_4 in Figure 2.1).

Another important property of edges is the so-called *edge weight*. An edge weight is a numerical value assigned to an edge. If all edges of a graph are weighted, the graph is called a *weighted graph*. The usually real edge weights provide additional information about the edges that algorithms can use for their computations. Often the edge weights represent the ideal distances between the nodes (see Section 5.2 and Section 5.3). For example, when the graph represents travel routes between cites, the weights of the edges representing these routes could be chosen so that they reflect the geographical distances

[2] It is assumed here that edges connect exactly two nodes. Multiple sources and multiple destinations may exist for hyperedges.

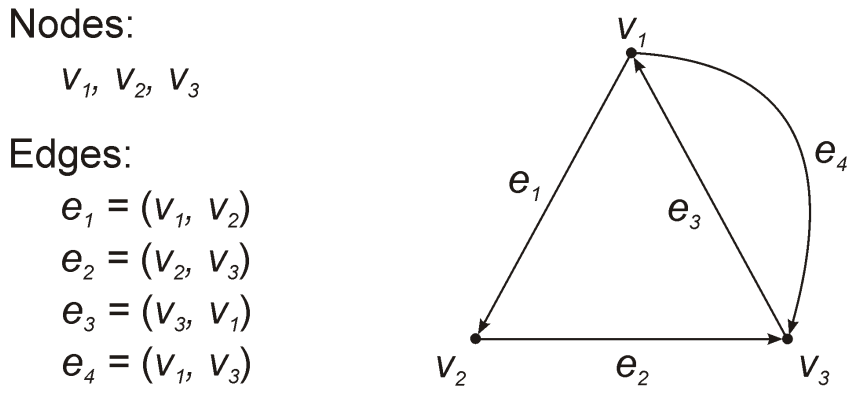


Figure 2.1: An example graph to illustrate the nomenclature used in this thesis.

between the corresponding cities. In this thesis graphs are, unless otherwise mentioned, assumed to be unweighted.

An example graph using the nomenclature described previously can be seen in Figure 2.1. Figure 2.1 shows the textual representation of a very simple graph with three nodes v_1, v_2 and v_3 and four (directed) edges e_1, e_2, e_3 and e_4. The edges e_3 and e_4 connect the same nodes, but in different directions.

2.1.2 Graph Drawing

Although textual representations of graphs convey the same information as graphical ones, they are much harder to understand and comprehend, even for simple graphs. Thus various means of drawing graphs suitable for the human user were invented. The visual representation of the graph in Figure 2.1 illustrates this. Nodes are represented by dots, edges are drawn using straight or curved lines connecting the dots of the nodes they belong to. Arrowheads represent the directions of the edges and textual labels are applied to the nodes and edges to identify them.

A *graph drawing* is a spatial representation of the abstract structure of the graph. So the graph is transformed from the abstract representation $G(V, E)$, which is in fact only a data structure, into some arrangement of geometric objects in a multi-dimensional space. This m-dimensional target space, where $1 \leq m$ and $m \in \mathbb{N}$, is called the *drawing space*. In almost all graph drawing applications, the drawing space will be two-dimensional and sometimes three-dimensional. If the drawing space is two-dimensional, it is called the *drawing area* instead. There are drawing algorithms which use m-dimensional drawings as intermediate results, see [Harel and Koren, 2002c], [Koren and Harel, 2005] and [Dwyer and Koren, 2006] for examples. Unless otherwise noted, the drawing space is assumed to be two-dimensional.

In general, the process of graph drawing can be divided into three tasks (see also Section 2.4.1):

Node Placement: Since the nodes V in a graph $G(V, E)$ represent the entities the relation E operates on, they are given a central role in most graph drawing algorithms. Thereby the placement of the nodes in the drawing area will have the most influence on the final image. Nodes are usually considered as infinitely small points.

Therefore, the drawing of nodes can be formulated as a function $f : V \rightarrow \mathbb{R}^2$ which maps each node $v \in V$, to a point $p \in \mathbb{R}^2$ on the drawing area.[3] Since the nodes are laid out in the drawing area, the function f is called the layout function and the algorithm implementing the function f

[3] For an m-dimensional drawing space this extends to $f : V \rightarrow \in \mathbb{R}^m$ correspondingly.

is called the *layout algorithm*. The appearance of the whole graph drawing is largely influenced by the node layout, so the term layout algorithm is often used to mean the whole graph drawing algorithm.

Edge Routing: To display the relation E between the nodes V of a graph $G(V, E)$, each edge $e \in E$ has to be drawn between the corresponding nodes. In many cases, edges are drawn as straight lines connecting exactly two nodes. These drawings are called straight line (edge) drawings. Nevertheless, sometimes aesthetic criteria (see Section 2.3) demand that edges be bent or curved for better appearance (see [Ware et al., 2002]). The task defining how a certain edge is to be bent, if at all, is called *edge routing*.

Edge routing can be described as a function $g : E \rightarrow k$, where $k : t \rightarrow \mathbb{R}^2$, mapping an edge $e \in E$ to a drawing function k, which in turn maps an internal running variable t, most likely $t \in [0, 1]$ and $t \in \mathbb{R}$, to the actual points $p \in \mathbb{R}^2$ on the drawing area. In practice, the drawing function k is often approximated by a polyline. A polyline is a sequence of points connected by straight lines. Thus the drawing function k simplifies to $k : u \rightarrow R^2$, where u is now a discrete variable of a finite sequence $(u_1, \ldots, u_l)$ of length l, with $2 \leq l$ and $l \in \mathbb{N}$.

Label Placement: Nodes and edges often carry semantic information with them that should be displayed in the final drawing. For example, a graph drawing showing traveling connections between cities is assumed to have the nodes labeled with the corresponding city names and could probably draw the distances between the cities on the edges. These node and edge labels consume space on the drawing area. Generally it is unwanted that these labels cover nodes or edges or are covered by nodes or edges and thereby hide information (see Section 2.3). Unfortunately label placement is even more difficult than node placement and edge routing and will not be covered here in detail. Further information about label placement can be found in [Neyer, 2001], [Harel and Koren, 2002b] and [Dwyer et al., 2005c].

To achieve an appealing graph drawing, the graph drawing algorithm has to consider all three drawing tasks at the same time. This is a nearly unmanageable thing to do. Although there are graph drawing algorithms which try to address this goal, their performance is low [Kaufmann and Wagner, 2001]. More recent approaches try to place nodes and route edges first and insert the labels later, changing the original layout as little as possible [Dwyer et al., 2005c]. Often, only the node placement and edge routing tasks are covered by drawing algorithms. Even then, edge routing is often simplified by performing only straight line routing.

2.1.3 Properties of Graphs

Graphs represent a certain structure of data. Such a structure can have interesting properties independent of any graph drawing. Two of the most important features of a graph $G(V, E)$ are the graph's connectivity and the graph's planarity.

A graph is considered *connected* if not a single node or a group of nodes can be separated from the other nodes without cutting through one or more edges. This is equivalent to saying that from every node $v \in V$ of the graph $G(V, E)$ there exists at least one path p to every other node $u \in V$, where $u \neq v$. In other words, every arbitrary pair of nodes (u, v) is connected by at least one path p. A *path* p is a sequence of nodes $(v_1, \ldots, v_k)$, where $1 \leq k$, $k \in \mathbb{N}$ and two successive nodes v_i and v_{i+1}, $1 \leq i < k$ and $i \in \mathbb{N}$, are connected by an edge $e \in E$. A directed path is a path with all edges along the path pointing to the same direction [Battista et al., 1999].

Identifying if the graph is connected or not is often a precondition for layout algorithms. If the graph is not connected, it can be split into multiple new, disjunct graphs that can be laid out independently. Connectivity testing is mostly done in cooperation with a search algorithm which traverses all nodes of the graph.

Evaluating all paths of a graph $G(V, E)$ allows the definition of the *graph theoretic distances* between nodes. The graph theoretic distance $d_{i,j}$ between the nodes v_i and v_j, with $v_i \in V$, $v_j \in V$, is equal to the length of the shortest path connecting the nodes v_i and v_j. The graph theoretic distance $d_{i,i}$ between a node v_i and itself is, unless otherwise noted, defined to be 0, regardless the node contains self-loops or not. If the graph $G(V, E)$ is unconnected, the graph theoretic distance $d_{i,j}$ between the unconnected nodes v_i and v_j is, unless otherwise noted, defined to be infinite. Most algorithms using graph theoretic distances presume that the graph is connected, so this aspect is rarely required.

Graph theoretic distances are used by many algorithms to describe proximity relations between nodes. For example, the graph theoretic distances are often used as edge weights which are interpreted by the layout algorithms as ideal, desirable distances in the layout. One of the strongest aesthetic criteria in graph drawing is the conservation of proximity information (see Section 2.3). Other proximity relations can in many cases be reformulated to graphs and graph theoretic distances. Unless otherwise noted, proximity in graphs will be stated in terms of the graph theoretic distance in this thesis.

Another very important attribute of a graph is planarity. A graph $G(V, E)$ is *planar* if it can be drawn on a plane without any edge crossings. Note that planarity only indicates that the graph can be drawn without edge crossings, not that a certain drawing method, such as, for example, straight line edge drawing (see Section 2.1.2), will produce a crossing-free drawing. Having no edge crossings significantly improves the readability of graphs (see Section 2.3). On the other hand, crossing freeness might require long and massively curved edges which may render the benefits of crossing freeness useless (see [Ware et al., 2002]).

If a graph is planar, a planar drawing without crossings will divide the drawing area into so-called *faces*. Each face is surrounded by at least one edge.[4] The infinite, unbounded face outside the graph is called external face [Battista et al., 1999]. A *dual graph* is a graph which contains the faces of another graph as nodes and the neighborhood information of these faces as edges (see [Battista et al., 1999; Kaufmann and Wagner, 2001]).

Due to the importance of planarity for crossing-free drawings, planarity testing has undergone heavy research in the past (see [Battista et al., 1999; Kaufmann and Wagner, 2001] for a list of references). Several layout algorithms, for example most of the orthogonal drawing algorithms, rely on graphs being planar. If a non-planar graph is to be used as input to such a drawing algorithm, the graph has to first be planarized by inserting a minimum number of new nodes, called *dummy nodes*, to remove the edge crossings.

Another important property of a graph $G(V, E)$ is the relation between the number of nodes $n = |V|$ and the number of edges $|E|$. A graph is called *full* or *complete*, if every node is connected to every other node by exactly one edge. This means that $|E| = O(n^2)$.[5] A graph with much lower than n^2 edges is called *sparse*. A graph with a number of edges approximately reaching or even exceeding n^2 is called *dense*. Whether a graph is sparse or dense often has significant influence on the runtime performance of layout algorithms or the storage space required to hold the graph.

2.2 Graph Representation

As already mentioned in Section 2.1.2, a textual representation of a graph is generally unsuitable for the human viewer. However, a textual or formalistic representation of a graph is very suitable for a computer. Although the storage and representation of a graph in a computer is not directly a graph drawing problem, it has direct influence on runtime performance in practice.

As there are two arrays in a graph $G(V, E)$, namely the set of nodes V and the collection of edges E,

[4] In the case of a self loop, which is when an edge starts and stops at the same node, the face will be circumvented by a single edge only.

[5] Exactly this equation is $|E| = \frac{n \cdot (n-1)}{2}$.

strategies for storing the graph exploit redundancies in these two structures. The two main approaches are the node list, also known as adjacency matrix, and the edge list.

Many algorithms require so-called adjacency information of certain nodes. A node $u \in V$ is *adjacent* to another node $v \in V$ when there exists at least one edge $e \in E$ so that u and v are in this edge e. For a directed graph, this notation extends to taking incoming and outgoing edges into account as well. An *incoming edge* e to some node u is a directed edge that has u as destination. An *outgoing edge* e to some node v is a directed edge that has v as source.

The relation E can be described by adjacency information, leading to a graph representation $A(V^2)$. For every node $v \in V$, the adjacent nodes to v are stored in a list. Instead of using lists, this adjacency information can be stored in an adjacency matrix A as well. The adjacency matrix is a table of n rows and n columns, each representing a certain node. If one node u is adjacent to another node v, this adjacency is marked in the cell at the corresponding row and column. If the graph is undirected, the adjacency matrix will be symmetric. If the graph is directed, the rows could represent the source nodes and the columns could represent the destination nodes. Nevertheless, the adjacency matrix always consumes space proportional to $|V|^2$.

The advantage of the node list or adjacency matrix approach is that neighborhood information can be retrieved very quickly, especially if the graph is very dense (which means that there are many edges). For dense graphs, the adjacency matrix is mostly filled and therefore the storage is used efficiently. A disadvantage of the adjacency matrix is that if the graph is sparse, much space will be wasted.

An alternative to the node list is the edge list. For each edge, the adjacent nodes are stored in a list. In most graphs an edge will only have two adjacent nodes and each edge list entry consists of two node references and therefore has constant size. A special optimization can be applied if the nodes are mapped to the natural number space. Then each edge in the edge list consists of only two integer numbers. The overall storage of such an edge list would then be $2 \cdot |E|$ integers.

For the edge list, the storage is obviously proportional to the number of edges $|E|$ in the graph.[6] This makes the edge list superior to the node list when storing sparse graphs. The major disadvantage of the edge list method is that the adjacent nodes of a certain node cannot be easily retrieved. The whole edge list has to be traversed in order to assemble the information resulting in long running times.

In practice, many graphs are sparse having less edges than nodes ($|E| < |V|^2$). Therefore most graph file formats such as dot or GraphML use edge lists to store the graphs (see Section 3.2 and [Brandes et al., 2006]). At runtime it is crucial to have both information about adjacency as well as explicit edges at hand, because most algorithms deal with both. Thereby a hybrid combination of the two techniques is key for fast running times. Section 4.4.2 discusses the hybrid data model used in GVS.

2.3 Aesthetics and Constraints

In this thesis in particular, but generally in the whole field of graph drawing, layout algorithms will not only be compared by their runtime behavior. More value will be placed on their ability to produce pleasing, appealing, and intuitive layouts. Of course, there is no strict definition of what is to be considered as pleasing, appealing, and intuitive. Beauty in graph drawing, it seems, lies within the eye of the beholder as well.

Nevertheless, it is the goal of every graph drawing application to produce aesthetically appealing layouts. All graph layouts try to transport information from the abstract graph to the human viewer and humans perceive information much faster if it is presented in a pleasing, appealing, and intuitive way. Therefore, aesthetic criteria or rules have to be found to help the algorithms to produce aesthetically beautiful drawings. Aesthetic criteria hinder the algorithm from placing the elements in a way that the

[6] Assuming that each edge has a constantly bonded number of adjacent nodes c, where $c \in \mathbb{N}$ and $2 \leq c$ and c is much lower than $|E|$.

human user cannot easily follow. Mathematically speaking, they are constraints that reduce the degree of freedom a layout algorithm has to draw a graph. For example, one important constraint is to keep the area consumed by the drawing as small as possible because the human field of view is limited.

Battista et al. [1999] define "aesthetics as general rules [...] that refer to the entire graph" whereas "constraints refer to specific subgraphs and sub drawings". In [Sugiyama, 2002] an even more abstract, more theoretical approach to describe aesthetics and constraints is provided. In this thesis, constraints are understood as all restrictions that may be applied to a graph drawing algorithm. This includes both general aesthetic criteria as well as application domain-specific constraints. The following list summarizes the most important aesthetic criteria and constraints:

Conserve proximity: Every graph describes relationships between entities and these relationships indirectly group entities together. For example, let a node u be connected with node v and node v be connected with node w so that the only path between node u and node w contains node v. It is intuitive to say that node u is closer to node v than to node w. Or, more formally, the graph theoretic distance $d_{u,v}$ between the nodes u and v is lower than the graph theoretic distance $d_{u,w}$ between the nodes u and w.

Probably the most fundamental aesthetic criteria is to conserve this proximity information of the graph in the graph drawing. Thereby nodes that are close to each other by graph theoretic distance should be placed together closely in the final layout as well (see Section 2.1.3). Often edge weights are used to pass the proximity information to the layout algorithms. Not only the graph theoretic distances could be used as proximity metric, but nearly any proximity metric could be defined dependent on the application domain of the graph drawing. Several layout algorithms try to conserve proximity by various techniques (see [Eades, 1984], [Fruchterman and Reingold, 1991], [Kamada and Kawai, 1989], [Koren, 2005] and [Dwyer and Koren, 2006]).

Area minimization: Large and wide drawings are difficult to perceive. The more the human eye has to wander around the drawing to overview it, the more difficult it is to understand. Therefore, minimizing the area of the drawing is key. In most cases the term area will be understood as the bounding rectangle surrounding the drawing, but also the convex hull could be used as a metric [Battista et al., 1999; Fruchterman and Reingold, 1991].

Another important issue considering the drawing area is the aspect ratio of the drawing rectangle [Battista et al., 1999]. This should usually match the aspect ratio of the viewing device the graph is to be displayed on, so that the available display area can be utilized most efficiently.

Crossing minimization: Drawings with many edge crossings are difficult to follow. If the graph is not planar, drawings having minimal edge crossings are desirable. Crossings between edges and nodes or their labels also hinder the readability of the drawing. Further references can be found in [Battista et al., 1999].

Bend minimization: The straighter edges are, the easier they are traced by the human visual perception system. Bends in edges, or even too sharp curves make them hard to follow. So it is good practice to keep the total number of edge bends as well as the local number of bends on single edges low (see [Battista et al., 1999; Kaufmann and Wagner, 2001]).

Edge angle maximization: If edges leaving or entering a node are too close together, they are likely to become indistinguishable, especially when the graph is dense. To avoid this, edges on a certain node should have maximum angles between each other. An example for this is 4-way-orthogonal layout [Battista et al., 1999]. Each node in a 4-way-orthogonal layout has exactly four adjacent edges, each spread out 90 degrees from each neighbor. Figure 2.2 (c) shows a layout similar to being 4-way-orthogonal, with the limitation that not all nodes have exactly 4 neighbors.

Symmetry and shape: Highly symmetrical things are easier to perceive than unsymmetrical ones. Furthermore the human brain is very good at thinking in metaphors. If the graph or parts of the graph are laid out so that they have a certain shape supporting the graph's meaning, it will be much more informative.

Clustering: When graphs are very dense they quickly become unmanageable by the human brain. Often the viewer is not even interested in all the details of every single node, but only in a general overview of the whole graph. Clustering places things together which belong together. This reduces graph density and allows an overall, hierarchical view of the graph.

Note that it is usually not possible to achieve all of these constraints at the same time. For example, the constraint to keep the drawing area minimal usually opposes the constraint to keep the number of edge bends minimal. Bending edges usually allows graph nodes to be placed closer together, but bent edges are much harder to follow by the human eye. Normally, a graph drawing application has to trade off the effects of different constraints in accordance with the results it wants to achieve. Figure 2.2 shows an example of how opposing constraints produce completely different layouts of the same graph.

Since not all aesthetic criteria can be achieved simultaneously, the question arises which criteria is the most important. Empirical studies were performed to analyze the concrete influence of the aesthetic criteria on the human perception of graph drawings [Ware et al., 2002; Purchase et al., 2002; Purchase, 1997]. In summary, the different results produced by these studies showed that the ranking of the aesthetic criteria is strongly dependent on the application domain. For example, for finding the shortest path between two nodes in a graph drawing, the continuity of that path is key, whereas identifying disconnecting edges in a graph drawing is heavily dependent on the number of edge crossings [Ware et al., 2002]. More information on validation of graph drawing aesthetics can be found in [Purchase, 2004].

2.4 A Taxonomy of Graph Drawing

A great amount of research has been done in the field of graph drawing and graph visualization and the field of graph drawing has become difficult to overview. There are many different actors involved in graph drawing. For example the end user, the viewer of a graph, is mostly interested in the graph's meaning. On the other hand, a software engineer has to consider runtime complexity and implementability and will generally be more focused on graph algorithms than on the actual semantics of the graph itself. Putting all these concepts into one single taxonomy is probably an unaccomplishable task.

The following taxonomy is divided into three general views a person may have on graph visualization:

Appearance: what the image looks like.

Algorithmic: how the image is drawn.

Domain: What the image represents.

Note that these views are not disjoint. They overlap in many areas and topics in one view may include topics in another view. Consider the views as windows into the field of graph drawing. Each window allows to see some section of graph drawing, where the same section may be seen from different windows. Figure 2.3 illustrates this concept.

2.4.1 The Appearance View

Someone not involved in graph drawing and mathematics will most likely characterize images of graphs according to their appearance. Although general properties of images such as color depth or resolution

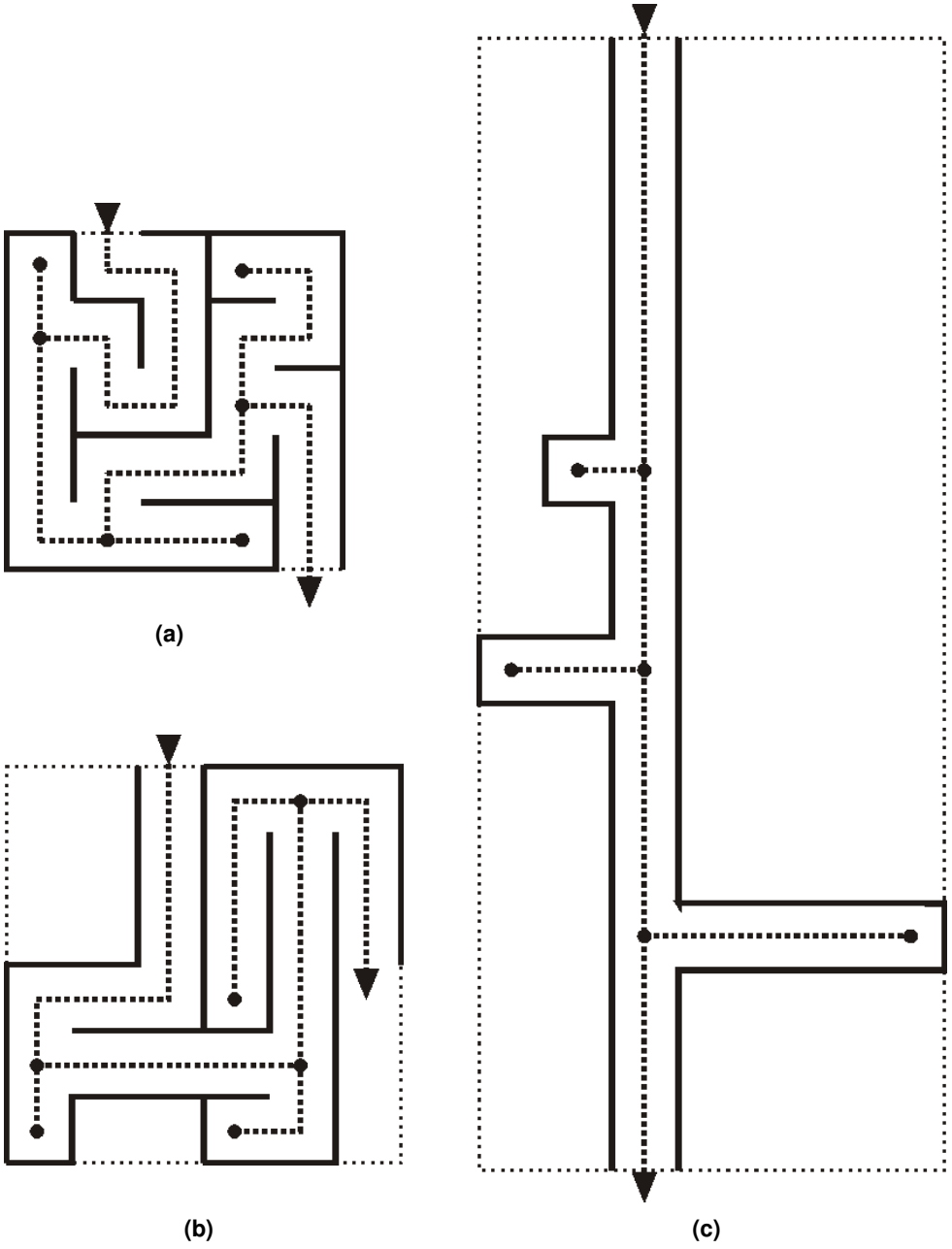


Figure 2.2: A labyrinth can be seen as an orthogonal drawing of a graph using minimal area and a maximum number of bends to confuse the viewer. Figure (a) shows the area minimal drawing of the labyrinth. Note that the main path is difficult to discover. Figure (b) shows the same graph on larger area with less bends. Finally, Figure (c) shows the same graph, but with a minimal number of bends. Now it is very easy to find the path through the labyrinth.

could be used to characterize the appearance of graph drawings as well, these features are not explicitly discussed here, because they are not directly related to graph drawing although they have influence on the appearance.

The appearance of the graph will be defined by the way nodes are placed and edges are routed. Furthermore the way additional information, for example node or edge labels, is displayed in the graph strongly influences appearance. Together this divides the appearance of a graph drawing into the following three parts (see also Section 2.1.2):

Node Placement: Nodes can be placed in two different ways on the drawing area: freely or on a grid. Additionally, borders may be applied to keep the nodes within some defined area [Battista et al., 1999]. Free placement means that the nodes are placed continuously on the drawing area, restricted only by the aesthetic criteria.

Placing nodes on a grid enhances the understandability and symmetry of a graph at the cost of flexibility. Grids can, among others, be square, rectangular, circular, radial or a hybrid combination of different grids depending on the desired layout (see [Sugiyama, 2002]). Furthermore, restrictions may be applied only on one axis or in one certain direction. This allows the placement of nodes so that hierarchies or directional dependencies become clearly visible (see [Sugiyama, 2002] and [Dwyer and Koren, 2006]).

Edge Routing: Most algorithms place nodes first and then route the edges between them afterwards. The most simple way to route edges is to draw straight lines between the corresponding nodes. This fails if the graph is dense and edges would therefore cross nodes or cross each other too often, which produces unpleasing results. In this case, bends can be inserted into edges. Bends can either be sharp so that the edge is drawn as a polyline, or they can be smoothed out so that the edge appears to be a curve.

Another issue when considering edge drawing is how to draw a directed edge. In practically all cases a directed edge is drawn with an arrow pointing from the source node to the destination. Though, for dense graphs, other techniques such as one-directional layering or color coding may be used to express edge directions [Dwyer and Koren, 2006].

Hyperedges may be drawn using some kind of special crossing on the edge or by drawing multiple edges between all connected nodes. However, several problems arise with hyperedges. Crossings or branches must be clearly distinguishable from ordinary crossings in the drawing and reduce readability, whereas drawing multiple edges results in denser layouts. Fortunately hyperedges are so rare in most graph applications that they can be seen more as a theoretical concept than an important part of graph visualization. This is supported by the fact that hyperedges are not considered by most of the common graph drawing algorithms at all.

Item Labeling: In most applications graphs are used to visualize the relations between objects. It is important that the viewer can identify which relations are displayed between which objects. A graph drawing consisting of points and lines only will usually not be very helpful. Therefore, in almost all cases of graphs being used in information visualization, the graph elements are assigned some kind of label. These labels can be textural or graphical or a combination of both. Labels can be applied to both nodes and edges, and sometimes even to faces (see Section 2.1.3).

Although labeling is probably the most important feature for a graph drawing, it is neglected by most of the layout algorithms. Taking variable sized labels into account drastically increases an algorithm's complexity not only in terms of runtime, but also in terms of simplicity, understandability, and implementability [Neyer, 2001]. Recent approaches draw a graph with traditional algorithms first, place the labels and then modify the drawing to reduce node and edge overlaps while preserving the graph's layout [Dwyer et al., 2005c; Harel and Koren, 2002b].

There are several terms used to describe special combinations of the features described above [Battista et al., 1999; Kaufmann and Wagner, 2001]. When nodes are placed on a normal grid and edges are routed on this grid as well, or at least parallel to the grid lines and having only orthogonal bends, the resulting layout is called *orthogonal*. Trees are commonly drawn placing nodes with the same depth (the distance from the root node) on layers. Each layer is parallel to the next one. Therefore, such layouts are called *layered*. Directed graphs with one distinct direction may also be drawn in layers [Battista et al., 1999; Sugiyama, 2002].

2.4.2 The Algorithmic View

A software or algorithm designer will use several techniques to solve the problems introduced by graph drawing and the aesthetic criteria. Figure 2.4 shows three different layouts of the same graph drawn by three different graph drawing algorithms. The following strategies are frequently used to implement layout algorithms (see [Battista et al., 1999; Mutzel and Eades, 2002]):

Planarization: Orthogonal layouts use planarization techniques to make a non-planar graph planar and then draw the planar graph using graph theoretic algorithms [Battista et al., 1999; Kaufmann and Wagner, 2001].

Divide and Conquer: Graphs are broken down into smaller components for which smaller layouts are calculated. The smaller layouts are assembled to form the overall layout [Battista et al., 1999]. This technique is often used when drawing hierarchies.

Axis Separation: Drawing by axis separation is a technique which orders nodes along each axis using different criteria. For example, the Sugiyama drawing method for directed graphs uses one axis of the two-dimensional drawing space for layering the nodes and the other axis for ordering nodes on the layers in such a way to minimize edge crossings [Sugiyama, 2002].

Force-Directed Placement: One of the most flexible and therefore most commonly used methods is the spring-based or force-directed approach. The underlying concept of the graph layout is a physical energy model. The edges are modeled by physical springs and the nodes may be charged so that forces arise which repel or attract nodes.

Force-directed placement algorithms produce natural looking layouts with the benefit that related nodes in the graph are placed closely to each other. A shortcoming of the force-directed approach is that calculating the layout is an iterative process and is quadratic in runtime. This is especially crucial for large graphs with more than several hundred nodes. There is vast literature about spring-based drawing and about how to improve it [Eades, 1984; Fruchterman and Reingold, 1991; Kamada and Kawai, 1989]. Further information about force-directed placement can be found in Section 5.1.

Multi-dimensional Embedding: Creating a layout of a graph in low-dimensional space is far more difficult than creating a layout in higher-dimensional space [Harel and Koren, 2002c; Koren and Harel, 2005]. Furthermore, aesthetic criteria can be formulated and enforced much more easily in multi-dimensional space. Hence, an appealing layout can be "embedded" in multi-dimensional space and then transformed into a lower- (two-) dimensional space by some, hopefully fast, multi-dimensional scaling technique which preserves the layout as far as possible (see [Harel and Koren, 2002c]).

Spectral Layouts: Large graphs with thousands or even tens of thousands nodes challenge common layout algorithms. Spectral algorithms utilize eigenvectors of matrices defined by the graph to compute the layout. One major benefit of spectral methods are that they can produce optimal layouts (in accordance with some aesthetic criteria) in reasonable time (see [Koren, 2005] and [Koren, 2003]).

Several further, finer distinctions could be made here regarding numerical solver algorithms or other mathematical techniques, but the enumeration presented above should be seen as an overall categorization rather than an exact one. Different classifications can be found in [Sugiyama, 2002, page 17], [Kaufmann and Wagner, 2001] and [Battista et al., 1999].

2.4.3 The Domain View

The topic of graph drawing can also be looked at from the customer's or user's perspective. The question that arises here is what the visualization should represent at all. Of course, graphs are predestined to represent relations between entities, but each application domain will have its own imagination of how this is best done. For example, UML diagrams are usually drawn in orthogonal style, whereas hierarchical directory structures are typical candidates for layered drawings.

Since there are so many applications for graphs in information visualization, the following list does not claim to be complete. It rather gives a rough and general overview of different layouts which arise from different applications.

Maps: One of the first applications of graph drawing were maps (see [Tufte, 2001]). Many of today's graph drawing concepts have been shaped by map drawing throughout history. Usually, maps not only show abstract relationships but also convey distance and geographical information. Geographical accuracy is usually inversely proportional to the level of abstraction of the map. For example, a road map will usually have not much in common with the graphs presented in this thesis, whereas a subway or train map, with a higher degree of abstraction, is recognizable as graph.

Entity-relationships: In computer science, but also in many other fields like chemistry and genetics, entity-relationship diagrams are very common. They may represent dependencies in UML diagrams, relationships in databases, or links in hyperspace. Many drawing algorithms are intended for use in these application domains.

Flow diagrams: Flow diagrams represent successive tasks with directed dependencies among each other, and are used in describing business transfers and workflows. Layered graph drawing algorithms are well-suited to flow diagrams.

Trees and hierarchies: Trees are a fundamental data structure in computer science. Furthermore, trees intuitively represent hierarchies that are common not only in computer science, but in many aspects of everyday life as well.

The classical layout for a tree is the layered, rooted, top down view. Several algorithms exist to visualize trees in this way (see [Buchheim et al., 2002]). Alternative display forms for trees include radial [Battista et al., 1999; Kaufmann and Wagner, 2001] and hyperbolic (see Section 3.9.2) layouts. See Section 4.2.1 for more information on tree and hierarchy visualization.

A good graph drawing application will take all three of these views into account to produce the drawings. Each decision for a certain plan of action in one view will influence the decisions in the other view. Unfortunately, there is no universal formula to balance these factors. For every graph drawing application, even for every graph drawing itself, these choices have to be made again. This is perhaps one reason why graph drawing is considered a very complicated topic.

2.4.4 Dynamic Graph Drawing

All graph drawing techniques described so far are considered to be static, because they assume that a graph's structure does not change after the layout is computed. In contrast, dynamic graph drawing deals with the drawing of graphs which frequently change.

For example, consider an application where users need to add or remove nodes and edges to and from the graph frequently. The graph layout could, of course, be computed again after every user interaction using static graph drawing methods, but this would most likely not produce user-friendly results. Such a redrawing could completely rearrange the graph drawing, confusing the user. A dynamic graph drawing algorithm must therefore preserve as much of the original layout as possible after an interaction. This is only one example for the various new criteria that arise when doing dynamic graph drawing.

Because static graph drawing is already a difficult task, dynamic graph drawing is even more difficult. Only very few dynamic graph drawing algorithms exist that are flexible enough to be used in general. Dynamic graph drawing will not be covered in this thesis directly. More information on dynamic graph drawing can be found in [Tatzmann, 2004] as well as in [Kaufmann and Wagner, 2001].

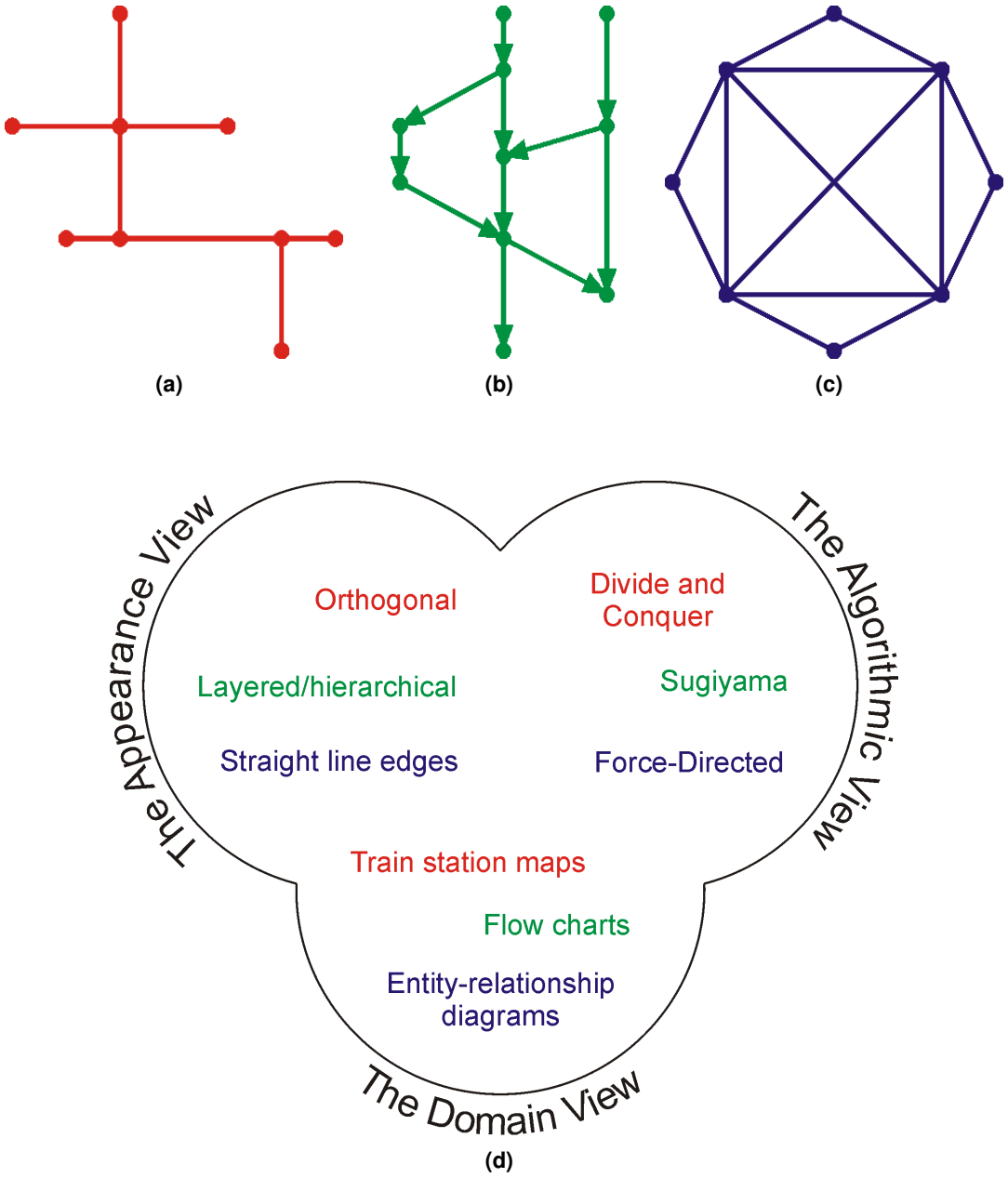
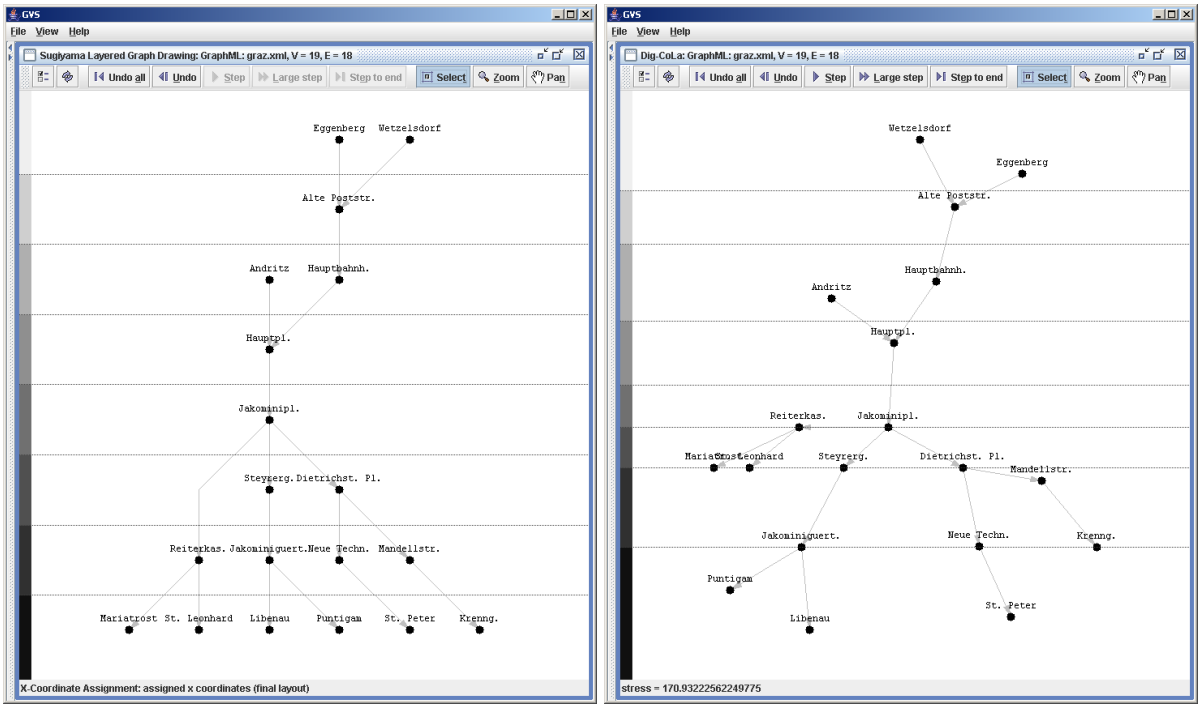
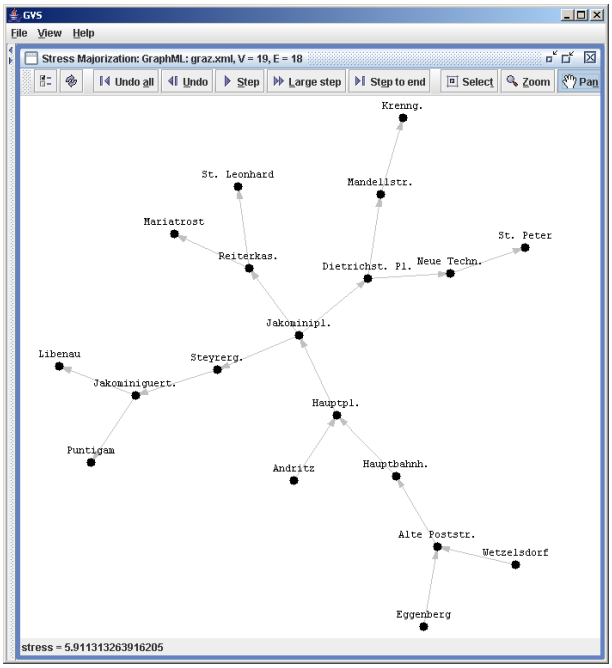


Figure 2.3: Three views on the field of graph drawing: by appearance, algorithm and domain. Illustration of the three views as described in Section 2.4 on three exemplary graph drawing applications. The red example in Figure (a) symbolizes an orthogonal map calculated by some divide and conquer algorithm, which could be used to draw train station maps (see [Battista et al., 1999]). The green example in Figure (b) shows a typical layered flow chart as could have been created by the Sugiyama algorithm (see [Battista et al., 1999; Kaufmann and Wagner, 2001; Sugiyama, 2002]). The blue example in Figure (c) could characterize an entity-relationship diagram, which is often drawn with straight line edges mostly by force-directed algorithms (see Section 5.1). Figure (d) shows how these three examples could be classified by the three views of the taxonomy.



(a) Sugiyama-style layered drawing.

(b) Dig-CoLa constrained layout.



(c) StressMajorization layout.

Figure 2.4: Three screenshots of GVS visualizations of the `graz.xml` graph, which is a reduced version of the Graz Tram Map shown in Figure 1.1. Figure (a) shows the layout produced by the GVS implementation of the Sugiyama algorithm (see [Battista et al., 1999; Sugiyama, 2002]). Figure (b) presents the layout calculated by the Dig-CoLa algorithm (see Section 5.3). Figure (c) was produced using the StressMajorization technique (see Section 5.2).

Chapter 3

Graph Drawing Packages

" If I have seen further it is by standing on ye shoulders of Giants. "

[Isaac Newton]

Since graph drawing is heavily used in computer science and software engineering, it is no wonder that many software packages for drawing graphs are available. This chapter discusses a selection of graph drawing packages and compares their advantages and disadvantages. A general overview of each package is given as well as a summary of the underlying software architecture.

The packages presented here were selected for their relevance to the succeeding chapters and their usefulness for the design of GVS. Of course, there are various other software packages as well and this list does not claim completeness. Most are special purpose tools dedicated to a particular application. See [Jünger and Mutzel, 2003], [Healy and Nikolov, 2005], [graphdrawing.org, 2006] as well as [Google, 2006] for more packages.

3.1 Important Issues of Graph Drawing Packages

Discussing all the graph drawing packages fully is not possible in this thesis. Only the key features of each package are characterized. Nevertheless, some aspects of importance for the development of GVS are described in more detail. The following sections contain graph drawing packages that strongly influenced the design of GVS. The description of these packages is structured as follows:

General overview: The package and its history is briefly introduced. The main features and attributes of the graph drawing package are presented.

Availability: Information about the availability of the software is provided. Although many packages are open source and their source code is available for non-commercial use, some packages are commercial and do not provide their source code.

Implemented algorithms: Often the capabilities of a package for drawing graphs can be more accurately described by the algorithms the package uses. This allows a distinction to be made between real graph drawing packages supporting a variety of different algorithms and special purpose software which only uses a few algorithms.

Software architecture: Of particular interest to the design of GVS is the way other graph drawing software packages are designed. For every package, a short overview of the software design principles used in the package is presented.

Discussion: The discussion about the advantages and disadvantages of each package is assembled from the facts gathered about the package with respect to the design of the GVS.

3.2 Graphviz

Graphviz is one of the oldest still active graph visualization packages [Graphviz, 2006b]. The package was first implemented in the early 1990s at AT&T Research Labs [Ellson et al., 2003] and has developed into one of the most widely used and accepted graph drawing packages. The AT&T developer group maintaining Graphviz actively participates in graph theory and graph drawing research. One of the latest results of their efforts is the Dig-CoLa layout algorithm (see Section 5.3).

Instead of being a single all-in-one package, Graphviz consists of several separate command line tools. These tools are intended to be concatenated by Unix pipelines and hence implement the Pipes and Filters design pattern [Buschmann et al., 2004]. This allows not only automated graph drawing, but also automating the whole graph drawing application. Given some command line parameters, graph input files are transformed into image output files without further user interaction. The input file format for all Graphviz layout engines is the common "dot" format [Graphviz, 2006a]. Several further command line tools exist for transforming other graph file formats into dot. Concerning output formats, postscript, Scalable Vector Graphics (SVG), and several raster formats such as JPEG, GIF, and PNG are supported directly by the Graphviz tools.

The separate building blocks Graphviz consists of are called layout engines. These actually perform the task of drawing the graph. The central layout engine in the Graphviz package is the dot layout engine [Gansner et al., 2002]. Other filters are available for different layout styles and algorithms (see [North, 2002]). Figure 3.1 shows the layouts produced by different layout engines.

Although Graphviz provides a minimal graphical user interface, it is basically a collection of command line tools. The user interface is intended only for specifying the parameters and files graphically, but does neither display the graphs nor show the produced layouts. Other packages for viewing images are required to display the output.

Availability: Graphviz is an open source project under a common license which allows commercial use with restrictions. The source code is freely available. Graphviz is part of various Unix/Linux distributions most commonly known as the "dot" package[1]. There are Windows versions of Graphviz as well.

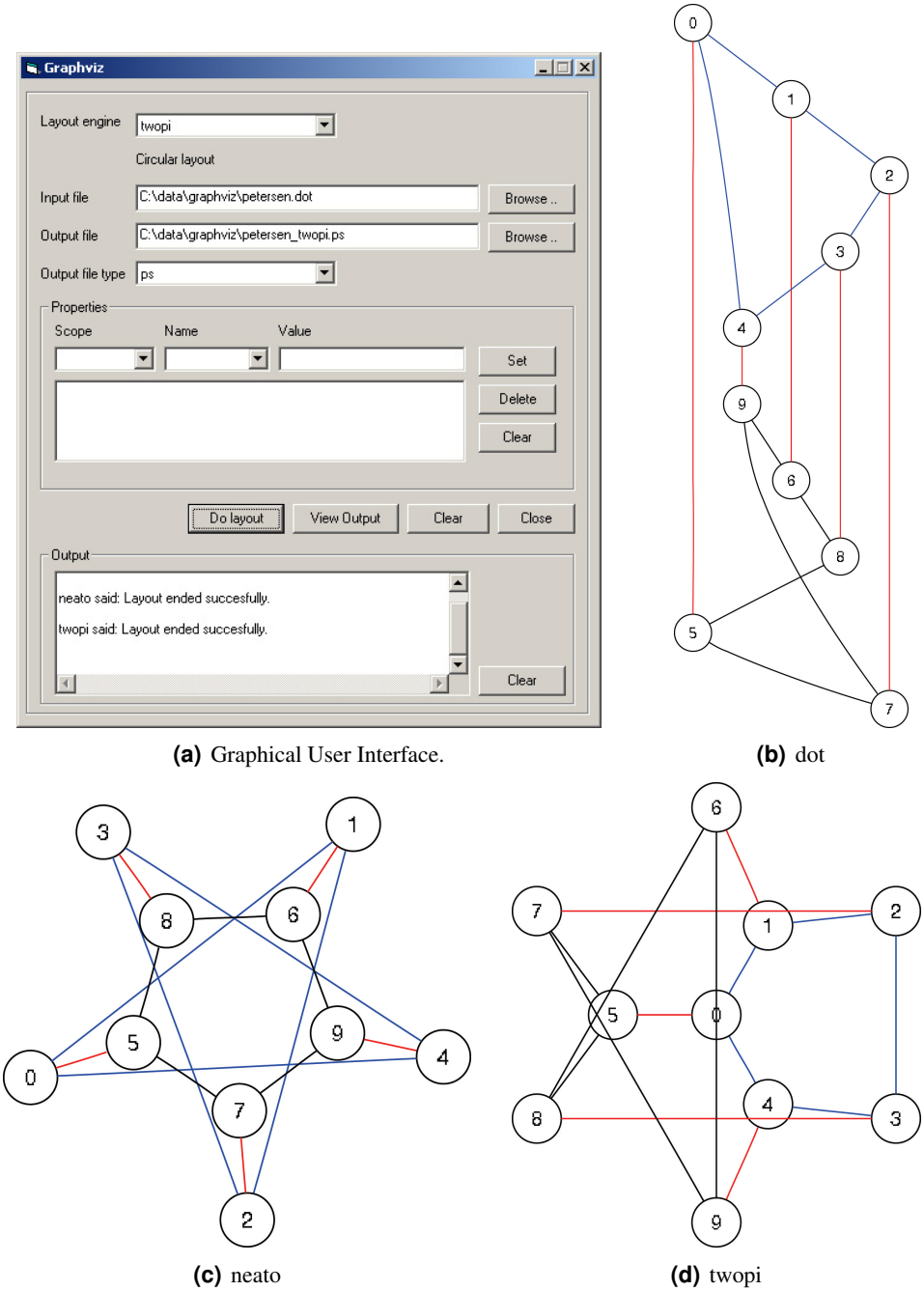
Implemented algorithms: Graphviz implements many different layout algorithms, including layered layouts, the palette of spring-based layouts, radial layouts, as well as an algorithm for avoiding and removing node (label) overlaps (see [Ellson et al., 2003]). Of course, the algorithms invented or developed by the AT&T research group, namely StressMajorization (see Section 5.2) and the Dig-CoLa algorithm (see Section 5.3) are available in Graphviz too. See also [Gansner, 2004].

Software architecture: Graphviz was originally written in plain C. Later, the design was changed to object-oriented C++, but still many features in the Graphviz software architecture remain in C. Originally, Unix was chosen as the target platform, but because of Graphviz's popularity amongst developers, many other platforms are supported yet as well.

The package's facade facilitates the Pipes and Filters design pattern [Buschmann et al., 2004]. From that point of view, the software design of the filters is very simple. First, the input streams are parsed. Second, the parsed graph data is transformed by the layout engine. Third, the computed layout is transformed into the desired output image.

Internally, Graphviz uses the Layers design pattern [Buschmann et al., 2004] to organize the layout engines. This makes it possible to develop new layout algorithms in a fairy high level of abstraction without having to worry about low-level details.

[1] Dot is the name of the first Graphviz layout engine. See [Gansner et al., 2002].



(a) Graphical User Interface. **(b)** dot

(c) neato **(d)** twopi

Figure 3.1: The Graphviz package. Figure (a) shows the Graphical User Interface (GUI) to Graphviz's main programs. Figures (b) to (d) are different layouts of the same graph computed by different layout engines. Figure (b) was created using the dot engine [Gansner et al., 2002]. Figure (c) shows the same graph laid out by the neato engine that uses a Kamada and Kawai [1989] force-directed approach [North, 2002]. Figure (d) displays the graph rendered using the twopi radial layout engine [Ellson et al., 2003].

Graphviz also can be used as a library within other applications [Gansner, 2004]. An Application Programming Interface (API) is provided to access the Graphviz functionality. The Graphviz API is supposed to be used by C or C++ programs. Several language bindings exist to integrate Graphviz into different languages or systems (see [Graphviz, 2006c]). Unfortunately there are currently no Java bindings.

Discussion: Graphviz is a slim and very fast package without unnecessary overhead. This comes mainly from the fact that Graphviz does not, like other packages, try to intermingle graph drawing and editing. Developing a (good) viewer or editor is not an easy task. The house-made viewer seems to be the bottleneck in nearly all other graph drawing packages. Note that the dotty graph editor [Koutsofios and North, 1996], which could be seen as Graphviz's graphical front end, is built upon dot, not intermingled with it.

Combined with Graphviz's powerful command line features, the Pipes and Filters architecture of Graphviz is probably the cause why "dot" is so popular in the developer community. For example, Doxygen, the C++ documentation generation system, uses Graphviz to lay out class diagrams derived directly from source code files [Doxygen, 2006]. For a list of other projects using Graphviz see [Graphviz, 2006c].

A negative aspect of Graphviz, from GVS' point of view, is that it is not only not written in Java, but in fact written partially in old style procedural C. This makes it difficult to directly use the Graphviz code, although executing the whole layout engine like an ordinary shell program from within the Java Virtual Machine would be possible.

Another negative point is Graphviz's dependence on the dot graph file format. When Graphviz was developed in the early 1990s, there was no standard graph file format, so Graphviz invented its own, rather complicated one. With the development of XML and the free availability of high-performance XML parsers, GraphML has become the new quasi standard graph file format and has rendered dot obsolete (see [Graphviz, 2006c] for a list of conversion tools).

Despite this, Graphviz is probably the most common and most accepted graph drawing package currently available. Since it is still under active development by the AT&T Research Labs group, Graphviz will definitely continue to be one of the global players in the graph drawing community.

3.3 Algorithms for Graph Drawing (AGD)

The Algorithms for Graph Drawing (AGD) package is a collection of several algorithms for drawing graphs in two-dimensional space [AGD, 2006]. AGD emerged from a cooperative project of Austrian and German universities in 1996 [Jünger et al., 2003]. The main goal of AGD is to provide the means for the design, analysis, implementation, and evaluation of graph drawing algorithms. In AGD, algorithms are embedded in a surrounding framework that handles all other, non-algorithmic issues, such as file input and output, low-level drawing etc. The focus lies on the developers' view rather than the end users' view.

Availability: The AGD Demo application is freely available for academic, non-commercial use, although at the time of writing, an online registration procedure is required to gain access. Figure 3.2 shows the AGD Demo application. Windows and Linux versions of the package are available. The developer files contain all headers and libraries necessary to use the package and create new algorithms with it. However, the core source code is not provided.

Implemented algorithms: Algorithms are provided for several aspects of graph drawing. There are algorithms for planarization, orthogonal drawing, algorithms for trees and hierarchies as well as layered

and spring-based methods (see [Jünger et al., 2003] and compare [Battista et al., 1999; Mutzel and Eades, 2002]). A more comprehensive listing of the algorithms in AGD can be found in [Mutzel et al., 2003].

Software Architecture: The AGD package is written in object-oriented C++. AGD uses the LEDA algorithmic package for basic data structure handling and computation [LEDA, 2006].

The AGD software architecture is based on the object-oriented concept of derivation and extension [Mutzel et al., 2003; Gutwenger et al., 2002]. New algorithms are implemented by deriving from existing ones or implementing special interfaces. The framework handles all preconditions and non-algorithmic work like actual drawing or input and output. This allows the algorithm designer to concentrate on the algorithm only and saves the algorithm classes from unnecessary overhead. Another important feature of AGD is its strict separation of graph structure and graph representation.

Discussion: AGD is a very large, but still modular and flexible framework. Object-oriented derivation as well as C++ templates allow new algorithms to be added without unnecessary overhead. The framework automatically integrates a new algorithm according to its preconditions. For example, the framework ensures that an algorithm that declares it can only operate on planar graphs will be prevented from being executed on non-planar graphs.

On the other hand, although AGD is a very comprehensible framework, it is more designed for the developer and not so much for the end user. Even though there are concepts that ease integration of new algorithms, actually implementing such new features in a framework of that large size might require more effort than, for example, implementing the same layout algorithm in a simple Java package. Furthermore AGD depends on the LEDA library, which is a commercial product.

3.4 Java Universal Network/Graph Framework (JUNG)

The Java Universal Network/Graph Framework (JUNG) is an open source Java 1.4 framework developed at the University of California, Irvine [JUNG, 2006a]. JUNG facilitates developing new graph drawing techniques similar to the AGD framework presented in the previous section. Furthermore, JUNG provides the Java developer with tools and classes to integrate the JUNG graph drawing features into other applications. According to O'Madadhain and Fisher [2005], the later was the primary cause for starting the JUNG project in 2003. Compared to the other graph drawing packages presented here, JUNG is very young and comparatively unknown.

Since it is written in Java, JUNG is able to cooperate with Java technologies like Java applets, Java Database Connectivity (JDBC), Remote Method Invocation (RMI) and further Java technologies. By using Java Swing and thereby Java2D, JUNG visualizations are platform-independent and can be used in any Java or web application. Figure 3.3 shows an example of JUNG being used as a Java applet. A list of Java applications already using JUNG can be found in [JUNG, 2006b].

Availability: The JUNG framework binaries and the JUNG Java source code are freely available under the open source Berkeley Software Distribution (BSD) license. All JUNG documentation is available online as well.

Implemented algorithms: JUNG provides algorithms not only for graph drawing, but also for analyzing networks [O'Madadhain et al., 2006]. The later heavily use matrix operations to perform their tasks so matrix calculation and evaluation is a large part of JUNG's library as well. For more complex matrix operations, the Colt library is used [Colt, 2006]. Additionally, JUNG uses a flexible extension mechanism to ease the implementation of new matrix-based algorithms. Further algorithms are available for computing statistics and graph analysis.

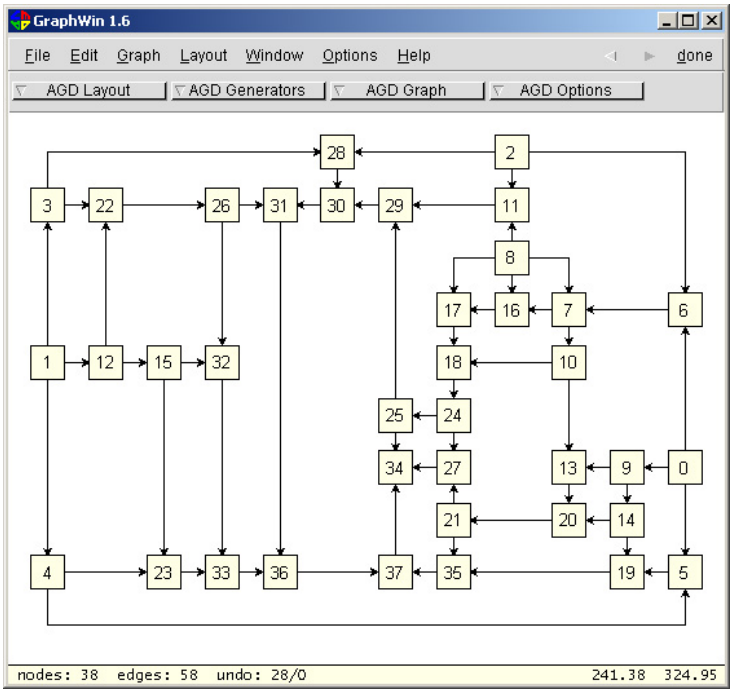


Figure 3.2: The AGD Demo application provided with the AGD package. A random, planar graph is drawn using the AGD PureOrthogonal layout algorithm [Mutzel et al., 2003]. The user interface is based on the on `graph_win` class of the LEDA library [Jünger et al., 2003].

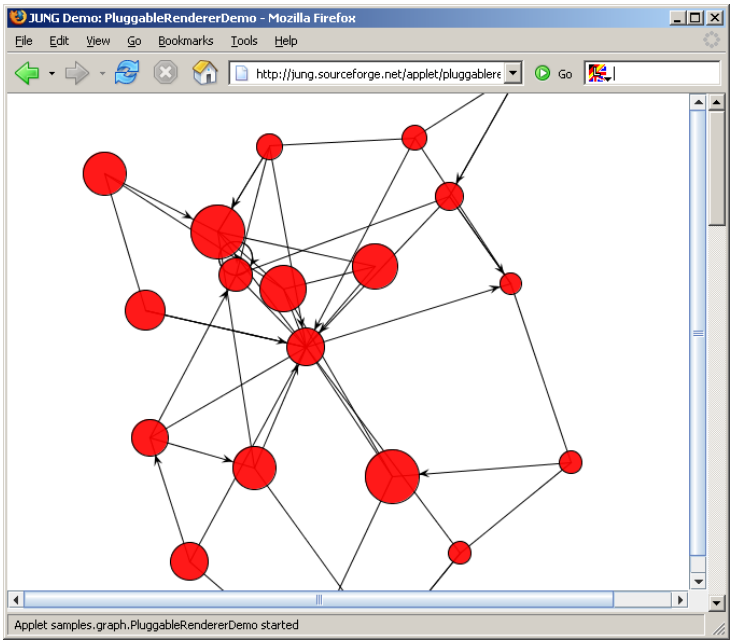


Figure 3.3: The JUNG PluggableRenderer Demo applet. See `http://jung.sourdeforge.net/applet/pluggablerendererdemo.html`. JUNG is designed for cooperating with web technologies such as Java applets, so this demo runs in a simple web browser window.

Unfortunately the set of actual graph drawing algorithms is rather limited. According to the JUNG API documentation [JUNG, 2005], the basic framework only contains spring-based algorithms and an implementation of Fruchterman and Reingold [1991] as well as an algorithm for laying out self-organizing-maps. Contributions from the JUNG online community include several more algorithms such as Kamada and Kawai [1989].

Software architecture: Being a Java framework, JUNG is strongly based on object-oriented design principles. Graphs, nodes, and edges are represented by objects linked to each other. Thereby graphs can easily be assembled using standard Java method invocation. Of course reading graphs from files is supported as well.

JUNG separates graph representation from generating the layout and rendering [O'Madadhain and Fisher, 2005]. Each of these stages is performed by separated components. This ensures that each component is exchangeable and new implementations can be added easily.

Another key concept of JUNG is that additional information on nodes and edges may be provided using standard Java data types. This can be done by either extending the corresponding graph element classes or by JUNG's annotation mechanism [O'Madadhain and Fisher, 2005]. Using the annotation mechanism, arbitrary data can be added to the graph elements. Note that this mechanism is very similar to the metadata concept used in GVS (see Section 4.4.2).

Discussion: JUNG has several advantages over the other graph drawing packages. In particular, being a full-featured Java framework using Java Swing makes it suitable for many applications while still being platform-independent. Its documentation and availability are positive aspects of JUNG.

In many concepts JUNG is already very close to those required for the GVS (see Chapter 4). The decision for not basing GVS on JUNG was for the following reasons:

- JUNG provides only very little means of animation. Although animation as needed by GVS could have been implemented, the effort would have been quite high.
- According to [O'Madadhain and Fisher, 2005], JUNG is neither a finished tool, nor intends to be one. It is permanently under development.
- The most important disadvantage, which actually settled the decision, is that JUNG is a Java 1.4 framework. GVS was intended to be a full featured Java 1.5 package from the start (see Section 3.11).

Nevertheless, JUNG is one of the most powerful Java packages presented here, and the fact that JUNG is open source and freely available, while still being actively developed, makes it an interesting option.

3.5 Pajek

Pajek is a semi-commercial package developed at the University of Ljubljana for analyzing large networks [Pajek, 2006; Batagelj and Mrvar, 2003]. The word Pajek itself is the Slovene word for spider. Analyzing large networks requires a fairy large set of capable tools and algorithms. Pajek combines many of these functions into one single package. Amongst them there are, of course, methods for drawing large graphs that represent such networks. Furthermore, there are algorithms for clustering and partitioning large graphs (networks) to make them more manageable. With these analytical features, Pajek is heavily used in social network analysis [de Nooy et al., 2005]. See [Batagelj and Mrvar, 2006] for a more comprehensive discussion of all of Pajek's features. Figure 3.4 shows Pajek's user interface.

In contrast to other graph drawing packages, Pajek emphasizes the interaction with the computation and analysis of the data itself. Pajek supports several different data types that may be used simultaneously

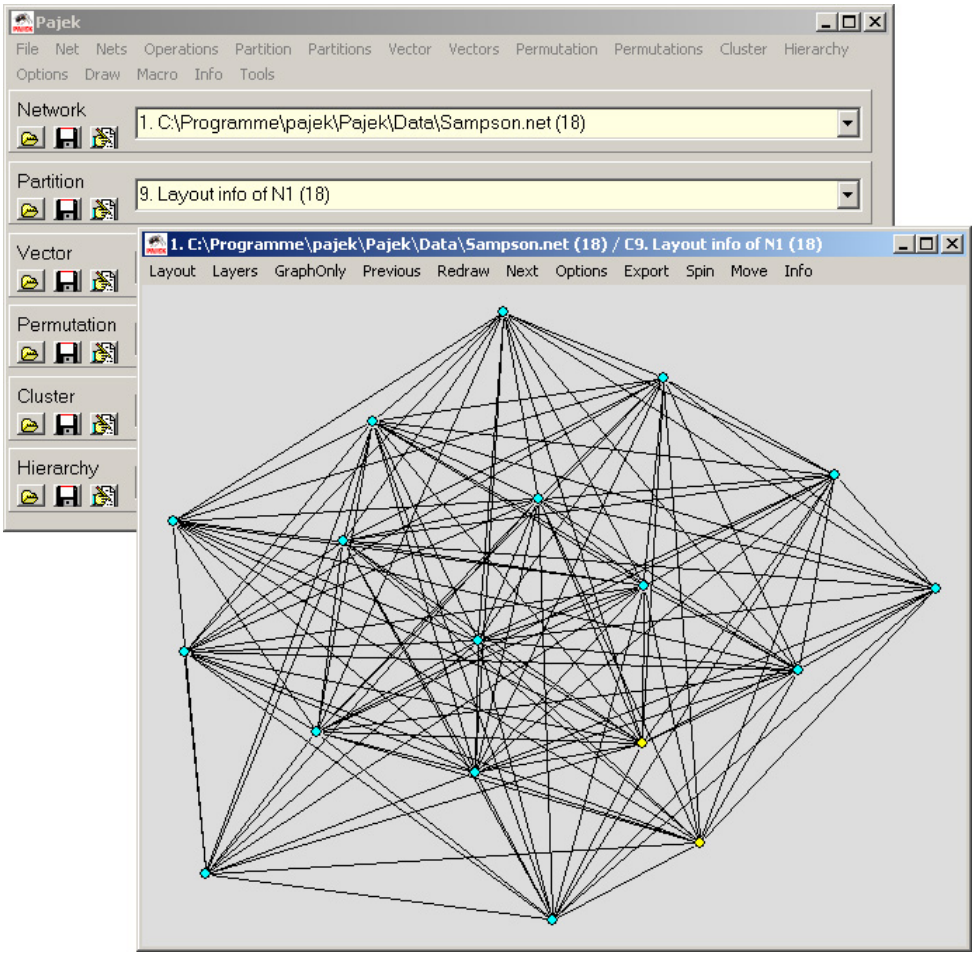


Figure 3.4: Pajek. The foreground window shows how Pajek lays out a graph using the Kamada and Kawai [1989] algorithm with random starting positions. The background window, which is Pajek's main window, specifies the data needed for the visualization. The light circles mark the two closest nodes in the layout.

to describe the data [Batagelj and Mrvar, 2003, 2006]. Transformations allow conversion from one data type to another. Analytical operations, such as clustering, flow analysis, and neighborhood finders, can be applied to the data. Pajek equips the user with a large toolbox of operations to create visualizations, rather than a limited set of predefined visualizations.

In this sense, Pajek can be compared to a calculator [Batagelj and Mrvar, 2003]. Data is loaded into the accumulators of the system. Sequential operations are performed on these accumulators. The user repeatedly transforms the data type and modifies the data until a satisfying result is achieved. Then this result can be printed out to several different formats such as postscript, VRML, or SVG [Batagelj and Mrvar, 2003]. A macro mechanism can be used automate repeatable tasks and thereby save redundant work [Batagelj and Mrvar, 2006].

Availability: Although binaries are available for non-commercial use, the source code is not available. Currently, there exists only a Windows version of Pajek, but a portable version is planned [Batagelj and Mrvar, 2003].

Implemented algorithms: Since Pajek is more focused on analysis rather than solely drawing large networks, there are many algorithms and functions not directly related to graph drawing. Nevertheless, the graph drawing algorithms include improved implementations of spring or energy based models [Kamada and Kawai, 1989; Fruchterman and Reingold, 1991] and an eigenvector-based technique is used for faster drawing. See also [Batagelj and Mrvar, 2003] and [Batagelj and Mrvar, 2006].

Software architecture: Pajek is written in Delphi and contains heavy support for various different input and output formats like GraphML and SVG. Since the source code of Pajek is not available, no judgment of the software architecture can be made at this point.

The most outstanding feature of the Pajek software is that it uses six internal data types [Batagelj and Mrvar, 2006]. Transitions between all these data types are provided so that different algorithms with different preconditions can operate on the data. This is in strong contrast to nearly all other graph drawing packages that try to define exactly one general and unique internal data model. One of the benefits of having multiple internal data models is that algorithms are simpler and faster if they are supported by the underlying data model. On the other hand, different data models require transformations between them, which in effect costs performance. The large number of different algorithms in Pajek make a unified data model infeasible.

Discussion: Pajek contains many features and much functionality all combined in one single package, everything under a single roof. This rich feature set makes Pajek suitable for many graph analysis and visualization tasks. Furthermore, Pajek allows visualizations to be assembled from scratch with minimal effort.

Unfortunately, this also causes the user interface to be rather crowded (though still manageable). Pajek seems to be focused on the developers' view rather than the users' view. Detailed knowledge of the various algorithms is needed to effectively use them. The programming language, Delphi, is not so common in the software development community. The fact that Pajek's source code is not available does not allow third parties to add new algorithms.

3.6 WilmaScope

WilmaScope is a package dedicated to 3D graph visualization using force-directed techniques originally developed by Tim Dwyer [WilmaScope, 2006]. Amongst the display of 3D layouts (see Figure 3.5), several techniques for clustering are supported [Dwyer and Eckersley, 2003; Dwyer, 2004]. Though WilmaScope is a stand-alone system, it supports interaction with other programs in order to perform visualization tasks.

Availability: WilmaScope is written in Java 1.3 and would therefore be fairy platform-independent if WilmaScope did not use Java3D library, which is often the cause for problems on many platforms (see Section 4.3.2). There is a fully standalone Windows package that includes the Java Runtime environment so that WilmaScope can be used with little installation effort. WilmaScope is an open source package under the GNU lesser public license and the Java source code is fully available.

Implemented algorithms: WilmaScope uses force-directed algorithms and multi-dimensional scaling [Dwyer and Eckersley, 2003]. These algorithms are extended to be suitable for three-dimensional graph drawing. One special application of this is the three-dimensional visualization of UML data using WilmaScope as described in [Dwyer, 2001]. WilmaScope's plugin mechanism allows the inclusion of Graphviz's dot layout engine (see Section 3.2) into the application, which enables layered graph drawing within WilmaScope.

Software architecture: The WilmaScope package is written in Java. The Java3D API is used for drawing the graphs. Graphs can be loaded into WilmaScope by generators [Dwyer, 2004]. Generators exist for creating new graphs with random properties or for loading graphs from files. Through the latter, WilmaScope supports its own XML-based format as well as a reduced form of GraphML. A generator in WilmaScope is a Java class implementing a certain interface with routines to carry out the transformation of external data into WilmaScope's internal format. This is very similar to GVS's own Generator concept (see Section 4.4.1).

The main design pattern in WilmaScope is a strictly separated Model View Controller (MVC) scheme [Buschmann et al., 2004]. The Model component consists of classes describing the graphs as well as the layout algorithms. Graphs and their layouts are displayed by the View component, which separates the visualization from the actual layout algorithm. User input and management are handled by the Controller component.

WilmaScope can be extended using standard object-oriented derivation or by WilmaScope's plugin and generator mechanism. Plugins support the addition of new functionality at runtime. One example is the already mentioned dot plugin to lay out graphs using the Graphviz's dot layout engine (see Section 3.2). A CORBA API was implemented in WilmaScope in order to allow other applications to access WilmaScope functionality.

Discussion: WilmaScope's software architecture, especially the plugin and generator mechanism, makes the package flexible and modular. Both have inspired the development of GVS (see Section 4.4.1 and Section 4.4.8). Another advantage of WilmaScope is that it achieves platform independence by being written in Java.

Nevertheless, for GVS, WilmaScope's concentration on three-dimensional graph drawing is too restrictive. GVS is aimed at two-dimensional drawing and WilmaScope does not emphasize this. Furthermore the dependence on the Java3D API reduces its applicability for GVS (see Section 4.3.2).

3.7 GEOmetry for Maximum Insight (GEOMI)

The GEOmetry for Maximum Insight (GEOMI) is intended for analyzing large networks using several techniques and was developed by Tim Dwyer and Michael Forster [GEOMI, 2006]. The main focus lies on insight into geometry of graphs and networks, as the title already suggests. The human user should perceive the structure of a large network by visually exploring and viewing its features in three dimensions (see Figure 3.6). Graph drawing and user interaction techniques are heavily used to support the user in this task.

Availability: GEOMI is based on WilmaScope (see previous section) and is therefore a Java application. Furthermore, GEOMI inherits WilmaScope's dependency on the Java3D library. Although WilmaScope is an open source project, the source code of GEOMI is not yet available.

Implemented algorithms: The GEOMI package contains various algorithms for analyzing, comparing, and grouping large graphs [Ahmed et al., 2005]. Since GEOMI is targeted on three-dimensional graph drawing, the algorithms used by GEOMI are improved three-dimensional versions of their two-dimensional counterparts known from other packages. There are algorithms for drawing hierarchies, trees, clusters, and groups in three dimensions [Ahmed et al., 2005].

In order to provide more insight into the structure of networks, special emphasis is placed on dynamic algorithms. The graphs may be changed dynamically by adding new nodes and edges, probably coming from an external data source. This behavior is, for example, needed for analyzing a network of web pages. Newly found pages are added to the graph at runtime. The graph drawing algorithm dynamically

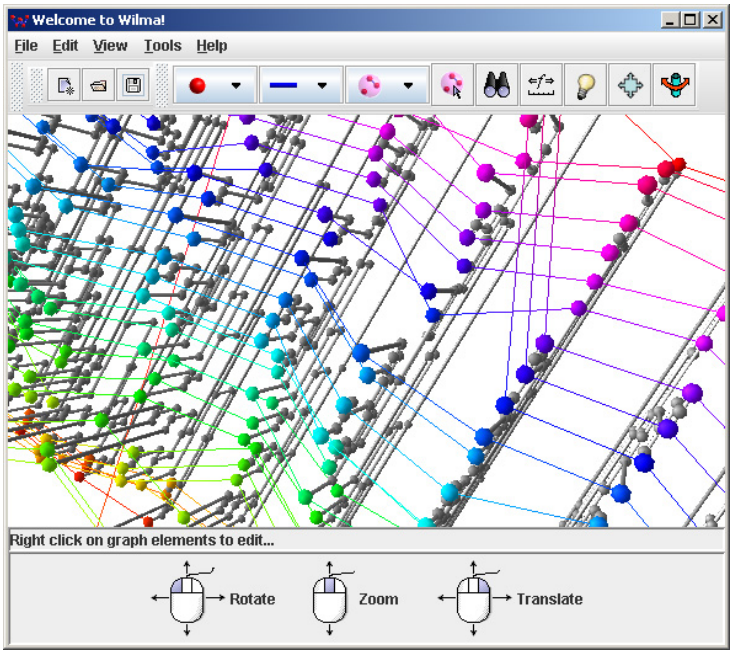


Figure 3.5: The WilmaScope graphical user interface showing multiple, three-dimensional layouts of trees. The gray bars show the structures of the trees whereas the colored lines link similar leaves together.

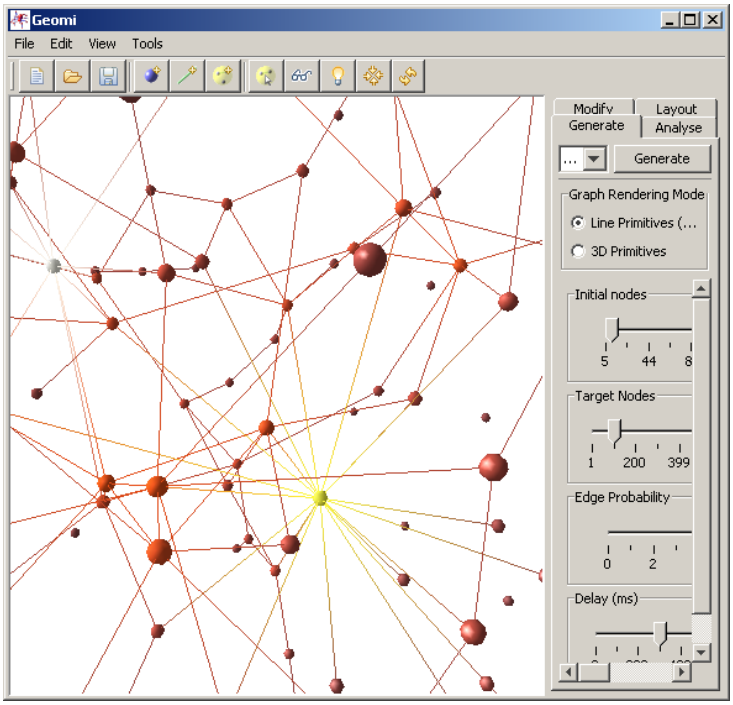


Figure 3.6: The GEOMI graphical user interface showing the `humdata.xwg` graph. The edges are rendered as lines instead of tubes to give more insight into the graph.

adapts the graph according to the new information. Thus, the user can monitor the change of the graph's structure over time. This can be seen as a sophisticated animation technique (see Section 4.4.7).

Software architecture: Unfortunately, the source code of GEOMI is not yet freely available. The software architecture information presented here is deduced from [Ahmed et al., 2005] and from [GEOMI, 2005].

The GEOMI package can be categorized into two parts: the framework and the plugins. The framework extends WilmaScope and creates an environment for the GEOMI plugins. The plugins actually carry the functionality, for example, for drawing a graph or interacting with the user. Each plugin is, with certain restrictions, an independent component. GEOMI's and WilmaScope's plugin concept is similar to the one of the InfoVis Cyberinfrastructure (see Section 3.10.3) and has inspired GVS' plugin management, which is presented in Section 4.4.8.

Discussion: Three-dimensional drawing of graphs helps to visualize large networks more intuitively, because shadows and lighting support the three-dimensional perception of the graph. Force-directed methods are used to extend two-dimensional graph drawing concepts to three dimensions. Interestingly, the three-dimensional graph drawing suffers from the same drawback as its two-dimensional counterpart, namely occlusion. For very dense graphs, the various edges and nodes are so close together that they occlude each other. This problem appears to be even more severe in three dimensions, because all nodes and edges are represented by three-dimensional objects.

Nevertheless, the main "maximum insight" seems to come from the user interaction with the three-dimensional objects rather than the layout algorithms themselves. The user may rotate the graph around and hence view sides that were not visible before. This effect is not easily achievable in two dimensions.

3.8 Tulip

Tulip is a graph drawing package dedicated to visualization of huge graphs developed by Auber David at the University of Bordeaux [Tulip, 2006; Auber, 2003]. It provides a modular, portable framework for developing new visualization applications. There is also a graphical user interface and editor for analyzing and displaying graphs.

Availability: Tulip is an open source package, freely available for Windows and Linux/Unix platforms. The source code is available under the GNU general public license.

Implemented algorithms: The Tulip framework contains various algorithms such as Reingold and Tilford [1981] and bubble tree [Grivet et al., 2004] (see Figure 3.7). Further algorithms are available for circular and radial layouts, for laying out graphs on grids and for drawing trees. For more information about the specific algorithms see [Auber, 2002].

Software architecture: Tulip is written in C++ heavily using the Standard Template Library. To render the graph drawings, which may be two- or three-dimensional, OpenGL is used [Auber, 2003]. The user interface components of the package are assembled from the Trolltech Qt widget library [Trolltech, 2006], which allows Tulip to be platform-independent.

Like other packages, Tulip uses object-oriented software design. The very flexible plugin mechanism allows the addition of new layout algorithms and input/output functionality to the core framework (see [Auber, 2002]). A main part of Tulip's performance is the result of using OpenGL for fast rendering. This also allows the addition of features like textures or glyphs easily. Glyphs are simplified meshes that

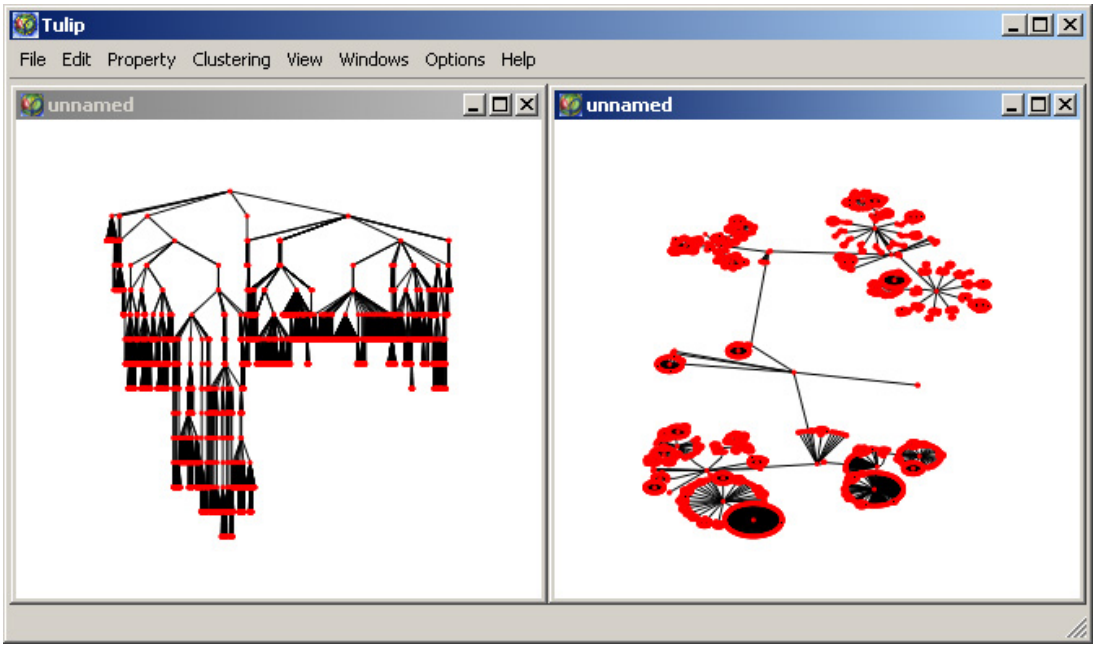


Figure 3.7: The Tulip graphical user interface. The left image was created using Tulip's hierarchical orthogonal tree algorithm Reingold and Tilford [1981]. The right image shows the same graph in a bubble tree representation [Grivet et al., 2004].

represent nodes. For example, a graph showing a food chain could have three-dimensional models of animals instead of nodes to make it more appealing.

Discussion: The Tulip framework is vast compared to other packages and provides much functionality, which is probably not required by most users of the framework. The flexible plugin system as well as the extensible kernel are the backbone of this package. OpenGL rendering is very fast compared to Java3D, which is used by several other packages.

Nevertheless, such a large framework as Tulip is not easy to understand. Tulip especially suffers from a lack of user and developer documentation. The graphical user interface of Tulip is not easy to handle. Furthermore the source code is difficult to understand, which is another drawback of the size of the framework.

3.9 Further Packages

The following section contains further graph drawing packages of minor importance for GVS, either because they are commercial or because they address different targets. A brief description of the packages is given here as well as links to further literature.

3.9.1 GLuskap

GLuskap is a three-dimensional graph drawing package written completely in Python [GLuskap, 2006; Python, 2006]. The word "gluskap" itself is the Algonquin name for the creator force [Dyck et al., 2004b]. Its source code is available from the University of Lethbridge, Canada under the GNU general public license.

The package uses standard libraries for rendering like OpenGL and SDL [Dyck et al., 2004b]. Although GLuskap can export its drawings to images or other graph description file formats, the main purpose is to export to the Persistence of Vision (POV) ray tracing format. A POV ray tracer can then be used to create high quality, photorealistic 3D images. Another important feature of GLuskap is that it can be used for stereoscopic viewing. It is thereby possible to interactively view a graph in a real three-dimensional environment [Dyck et al., 2004a]. Of course, corresponding hardware such as shutter glasses is necessary to use this technology.

Unfortunately the graph drawing algorithmic environment of GLuskap is fairy limited compared to other graph drawing packages. The emphasis distinctly lies on the three-dimensional visualization rather than the graph drawing algorithms (see Figure 3.8). Since GLuskap is written in Python, it is platform-independent and can be executed wherever the Python interpreter and the required graphics libraries are available.

3.9.2 Walrus

The Cooperative Association for Internet Data Analysis designed Walrus to draw huge graphs in three-dimensional hyperbolic space in order to visualize large network data [Walrus, 2006]. It is based on the research with the H3 hyperbolic browser (see [Munzner, 2000]). Hyperbolic coordinates and hyperbolic drawing allow a more compact display than traditional Cartesian drawing [Lamping and Rao, 1994]. This is especially crucial for drawing very huge graphs with millions of nodes.

Walrus is an open source package written in Java and its source code is freely available under the GNU general public license. The three-dimensional rendering is performed by Java3D. Walrus contains a graphical user interface (see Figure 3.9) to browse and zoom the large graphs being analyzed. Visual enhancements like color coding, transparency, and selection are implemented to increase the expression of Walrus graphs.

The only layout algorithm supported is hyperbolic layout. The hyperbolic layout of arbitrary graphs strongly depends on the spanning tree, which has to be provided by the user. Carelessly chosen spanning trees can result in poor aesthetics.

3.9.3 yFiles

yFiles is the main product of the yWorks company [yWorks, 2006]. yWorks is the commercial successor of the GraVis project at the University of Tübingen that was lead by Michael Kaufmann. The goal of GraVis was to design a flexible and generally usable framework for graph drawing.

yFiles continues the effort of making graph drawing easy for software developers. It is fully written in Java, but because the package is commercial, only the documentation is freely available. The yFiles package contains several different layout algorithms [Wiese et al., 2003]. It is designed for being used by other applications in end user products, rather than for special graph algorithmic purposes.

One interesting aspect of yFiles' software architecture is the way yFiles handles metadata associated with the graph elements (nodes and edges). Such data can be retrieved or set using special accessor interfaces [yWorks, 2005]. This allows the separation of the abstract graph structure and the metadata, while at the same time keeping the metadata close to the element it is assigned to.

3.9.4 JGraph

The JGraph library was originally created by Gaudenz Alder at the Swiss Federal Institute of Technology in Zurich in 2002 [JGraph, 2006]. Although there is still an open source version, JGraph has become mostly commercial. This applies especially to the support and the documentation. Not even the user manual is available freely.

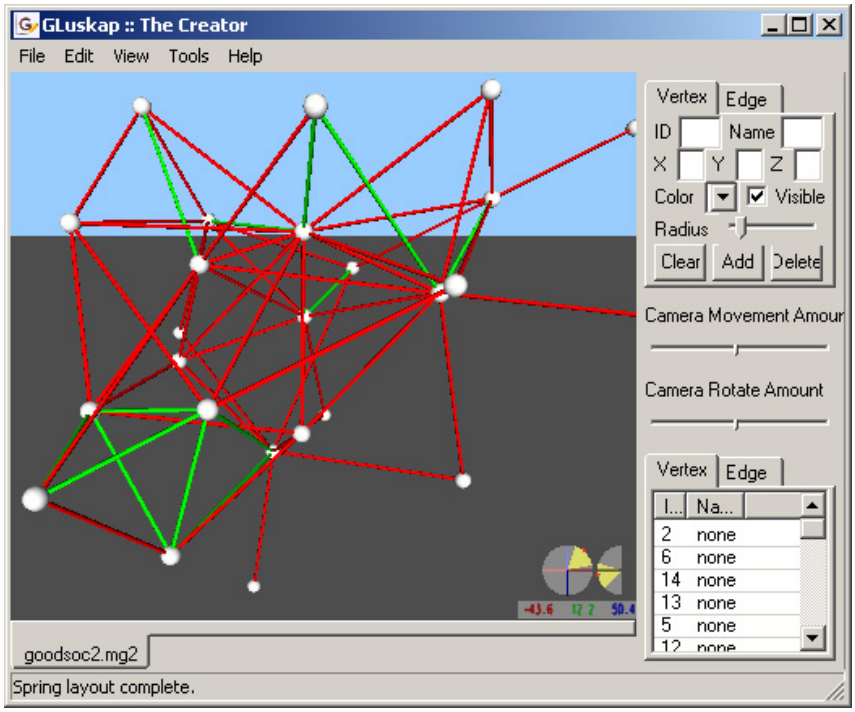


Figure 3.8: The GLuskap graphical user interface. The image shows the spring-based layout of a graph with a boolean attribute assigned to its edges. The color of the edges provides information about this boolean value.

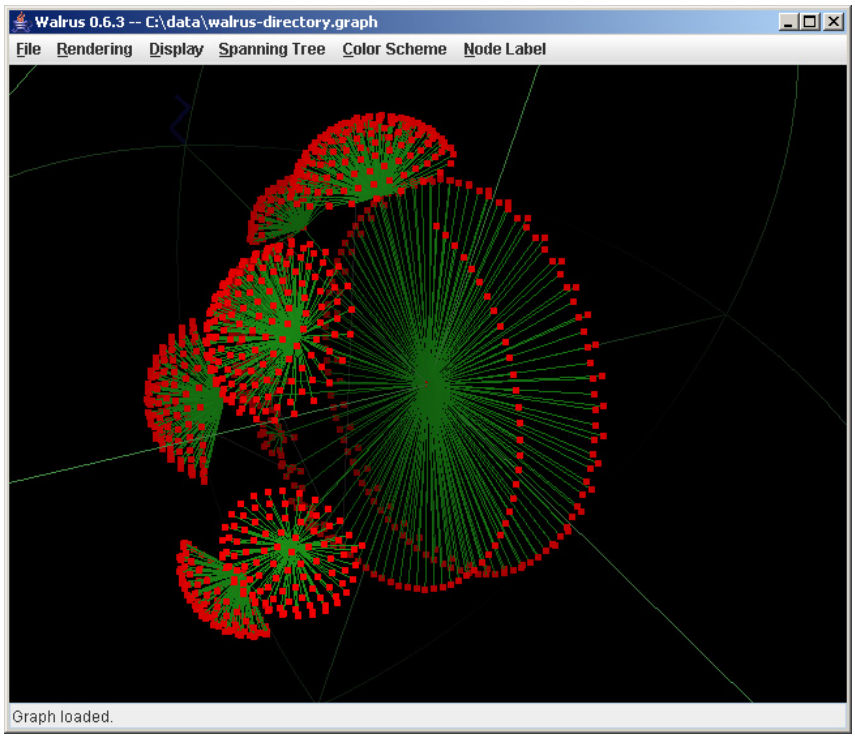


Figure 3.9: The Walrus graphical user interface showing a three-dimensional layout of the Walrus directory tree.

JGraph integrates graph drawing into the Java Swing framework. This allows other Java applications to easily integrate graphs into their user interfaces. This frees the surrounding application from having to deal with graph drawing details. Hence even developers not familiar with graph drawing details may use graphs in their applications.

JGraph may be compared to yFiles because they both are commercial Java products that try to make graph drawing available for everyday applications. Nevertheless, yFiles contains much more functionality and many more layout algorithms than JGraph. On the other hand, JGraph is easier to embed in applications.

3.9.5 aiSee

The aiSee package is the commercial successor of the Visualization of Compiler Graphs project at Saarland University in 1991 [aiSee, 2006]. These visualizations turned out to be helpful in many other fields as well. aiSee was improved to be more flexible, to support more file formats generated by other applications, and to support huge graphs.

Compared to other end user commercial graph drawing packages like yFiles and JGraph, aiSee is a stand-alone application. aiSee postprocesses abstract graph files provided by other applications and turns them into appealing drawings. An extensive user interface supports interactively influencing the graph drawing and thereby optimizing the visual output.

3.9.6 Tom Sawyer

The Tom Sawyer corporation is dedicated to the analysis of relational data of any kind [TomSawyer, 2006], including visualizing potentially large graphs. Tom Sawyer provides several product families for data analysis, information visualization, and graph drawing. An interesting feature of the Tom Sawyer software is that it is available not only for many platforms, but also for many programming languages. The most notable languages are C++ and Java.

3.10 Information Visualization Packages

The packages presented so far are all devoted to graph drawing and often directly emerged from the graph drawing community. In contrast, the packages presented in this section are more general information visualization packages (see Section 1.2). Their main purpose is to implement and show new techniques in the field of information visualization rather than solely graph drawing. Nevertheless, graph drawing is an important topic in information visualization as well, so many information visualization packages also contain graph drawing features, which will be discussed here.

3.10.1 The InfoVis Toolkit

The InfoVis Toolkit is a Java 1.4 information visualization framework created and maintained by Jean-Daniel Fekete [Fekete, 2006, 2004]. It is open source and its source code is freely available.

The underlying data structure of this framework is a table of unified components [Fekete, 2004]. This makes it suitable for content-base vector-spaces and multi-dimensional scaling applications (see [Andrews, 2006b]), but also restricts the toolkit's generality. A fast, OpenGL based 2D rendering library is used to accelerate the drawing of the visualizations. With its component-based software architecture, new visualizations can be easily integrated into the existing framework.

The algorithms implemented in the InfoVis Toolkit are more information visualization than graph drawing related. The InfoVis Toolkit is intended for supporting visualizations like parallel coordinates, scatter plots, or tree maps (see [Fekete, 2003]). Although graph drawing could be implemented by putting

the adjacency matrix into the tabular InfoVis Toolkit data structure and implementing the graph drawing and rendering from scratch, doing so would not be very feasible, compared to implementing the same visualization in a dedicated graph drawing package.

3.10.2 Prefuse

Prefuse is an open source Java 1.4 toolkit for interactive information visualization that originated from the University of California at Berkeley and the Palo Alto Research Center [Prefuse, 2006; Heer et al., 2005]. Prefuse strongly facilitates the user interaction component of information visualization. Similar to JUNG (see Section 3.4), Prefuse uses Java Swing and Java2D to build the user and drawing interface and is thereby platform-independent.

In contrast to the InfoVis Toolkit (see Section 3.10.1), Prefuse actually uses a graph as its basic data structure. The graph represents the abstract data that should be visualized. This abstract data is filtered into a collection of items that define how they are rendered to the screen. For example, a node with textual data may be filtered so that it is finally rendered to the screen using a circle and a text label.

The main components of Prefuse are its so-called actions. Actions actually decide how the abstract data is filtered and how the items interact with each other and with the user. A Prefuse visualization is basically defined by the order and the kind of the actions that are applied to the graph. Prefuse already provides several actions for creating tree maps, radial, and grid-based layouts, but in terms of graph drawing, the main layout mechanism provided is force-directed layout [Heer et al., 2005; Heer, 2004]. Although it is possible, implementing graph drawing strategies not based on force-directed methods would be difficult, or at least would not profit much from Prefuse's dynamic interaction concepts.

Note that at the time of writing Prefuse has undergone several design changes. The most notable one is that new data types have been added to support tables and trees directly. In this sense, Prefuse has become more similar to the InfoVis Toolkit (see Section 3.10.1). Further changes have been applied to the action concept and the way the final image is rendered to the screen. The most current information about the recent changes in Prefuse can be found in [Prefuse, 2006].

3.10.3 InfoVis Cyberinfrastructure

The InfoVis Cyberinfrastructure is the successor to the InfoVis Repository, which was released by Katy Börner and Yuezheng Zhou in 2001 [IVC, 2006; Börner and Zhou, 2001]. In contrast to the packages presented so far, the InfoVis Cyberinfrastructure integrates algorithms and packages from various different sources into one single framework. Using the InfoVis Cyberinfrastructure should allow the developer to seamlessly access and combine the benefits of these different components of the framework to create and design new information visualization applications.

In addition to the integration of the various different components into the framework, documentation about the algorithms and resources are provided online. So the InfoVis Cyberinfrastructure can also be seen as a collection of reference and information material about the various integrated algorithms rather than solely a software framework.

The framework itself is a Java 1.4 application. The three major components of the framework are: the data models, the GUI, and the plugins. For a more detailed description of the other components see [Penumarthy et al., 2004]. The data models provide data access mechanisms to the various forms of data that may exist in an application, such as matrices, trees, graphs, etc. Using the data models provides a unique and algorithm-independent way to access and load data into the framework. While the data models can be seen as the framework's data input component, the GUI is responsible for presenting the drawing results to the user. Furthermore, the GUI provides a way for the user to interact with the application and to start the plugins.

The InfoVis Cyberinfrastructure plugins perform the task of executing a certain algorithm from another toolkit or package. For example, to integrate a certain visualization into the InfoVis Cyberinfrastructure framework, a plugin exists that handles the data input and output conversion between the InfoVis Cyberinfrastructure and that external toolkit. Of course it is also possible to implement algorithms in the InfoVis Cyberinfrastructure directly [Penumarthy et al., 2004]. Since the InfoVis Cyberinfrastructure is very sparing in the requirements integrable toolkits must fulfill [Penumarthy et al., 2004], already numerous algorithms and packages have been integrated into the InfoVis Cyberinfrastructure.

3.11 Considerations on the Graph Drawing Packages

The large number of available packages for graph drawing already points out the difficulty of the topic itself. Even though there are so many of them, no two packages are the same. Every package addresses a different segment of graph drawing, each using different techniques, each emphasizing a different focus. Unfortunately there is no single all-in-one package that GVS could have been based on.

While Graphviz is an experienced and very common package, it suffers from the fact that large parts of it are written in plain C. On the other hand, large graph drawing frameworks like AGD and Tulip still suffer from the difficulty of operating in terms of such a large and comprehensive framework. Of course, implementing a small visualization would be more or less easy in every framework, but a large framework does not necessarily guarantee that large visualizations can still be implemented that easily. Furthermore, the larger a framework is, the more restrictive it becomes, because it forces the developer to use the framework's mechanisms only. Without knowing the framework's capabilities exactly, it is very risky to use the framework as basis for a project such as GVS.

A very suitable candidate for GVS would have been the JUNG framework. JUNG is a smaller framework already written in Java. Unfortunately the actual Java version JUNG is based on is 1.4, so the new features of Java 1.5, mainly generics and enumerations, could not be used then.[2] Starting the new GVS project based on the old Java 1.4 technology is not feasible. Additionally it is very likely that JUNG will be ported to Java 1.5 for this very reason. This would cause all dependent projects to be ported as well. If it is not ported, it is probable that JUNG will lose out to the new Java technologies and versions beyond Java 1.5.

The three-dimensional packages, like WilmaScope, GEOMI, GLuskap and Walrus, are all based on the scene graph Java3D API [Java3D, 2006b]. Java3D is a complicated API that adds another level of difficulty to implementing new algorithms to one of these frameworks. Even though there are some advantages of three-dimensional graph drawing, two dimensions seem sufficient for most of the current layout algorithms. Technologies like Java2D [Sun, 2006b] or the Java bindings for OpenGL (JOGL) [JOGL, 2006b] are more feasible for two-dimensional rendering. In Section 4.3.2 these issues are discussed in detail.

In summary, the survey of graph drawing packages showed that there existed no single package suitable for GVS. This led to the decision to implement GVS from scratch, of course taking into account the most promising concepts of the various packages.

[2] Of course Java 1.5 features could be used in a Java 1.5 program that uses classes from a Java 1.4 library, but the type-safety and convenience of Java 1.5 would not apply to the Java 1.4 code parts.

Chapter 4

The Graph Visualization System (GVS)

" In the middle of difficulty lies opportunity. "

[Albert Einstein]

The previous chapter described several graph drawing packages and their main concepts. These concepts and ideas were taken into account when developing the Graph Visualization System (GVS). This chapter describes the goals and aims of GVS and explains some of the decisions made during the development process. Furthermore, underlying concepts, ideas and features of GVS software design are explained. Technically detailed information about the implementation is provided in Appendix B.

4.1 Goals

According to project management literature [Drucker, 1993; Malik, 2001], the goals of every project must be stated clearly at the beginning. The following two primary goals proved key for GVS:

1. GVS is intended for use in teaching to explain how graph drawing is done.
2. GVS has to be flexible and extensible so student groups can work with it.

Point 1, GVS being a teaching framework, indicates that the GVS will be used in information visualization or graph theory courses to explain graphs and graph drawing concepts. GVS should be a showcase and toolbox to help students understand these concepts. This drives the need for a simple, easy to follow presentation of what is going on behind the scenes of graph drawing. Common graph drawing packages, as presented in Chapter 3, do usually not provide such features. They are intended for actually carrying out the graph drawing tasks as fast as possible with a minimum of details being revealed to the user. Furthermore, the source code of nearly all those packages, if available at all, is hardly documented and therefore difficult to follow.

This applies to Point 2 too. It should also be possible for students to implement their own graph drawing algorithms. Unfortunately, most graph drawing packages are very large software frameworks that are not easily understandable. There is often more effort involved in understanding a large framework and integrating a new method into it than implementing a (small) application from scratch. GVS should allow students or student groups to easily implement small applications within the context of the framework with minimal effort. Such an implementation should also integrate itself into the rest of the framework so that overall cohesion is preserved. This already suggests that student groups will mainly implement new layout algorithms. The framework should therefore support them in doing so as much as possible (see Appendix B).

Derived from the points above, the following design goals were stated for GVS. Of course these goals are not unique to GVS and to accomplish them is desirable for nearly any piece of software. They are written down here to explicitly emphasize their importance for the development of GVS:

Simplicity: The user interface, but also the whole framework, has to be as simple and understandable as possible. The user interface must speak a simple and explanatory language, unlike the user interfaces of the graph drawing packages presented in Chapter 3. Those user interfaces provide complete control over the underlying software designed to accomplish many complicated and different tasks. The main GVS use case in contrast is very simple: load a graph and visualize it. The GVS user interface must reflect this simplicity.

Module Separation: Although the main use case of GVS is simple, the task of graph drawing itself is not. Graph drawing ("visualizing") cannot be accomplished in one module. It has to be split into several, separate components. Each component must be as independent of the other components as possible. In particular, drawing the layout to the screen has to be separated from the layout algorithm, because the drawing itself involves platform-specific code which must not be intermingled with the abstract graph drawing algorithms.

Reuse: The separation of the framework into several modules also supports code reuse. Someone implementing a new layout algorithm will only have to write the code for the algorithm and reuse the components provided by the framework to do the drawing and the user interaction.

The person implementing the algorithm should not even have to worry about how the rest of the framework works and, of course, similar tasks should not be re-implemented over and over again. This was a problem with previous student implementations of graph drawing algorithms (see Section 4.2). Most of the implementation effort was spent in areas like graph representation or rendering, which had little to do with layout algorithms and graph drawing at all.

Taking these goals into account, it became obvious that the common graph drawing packages, as presented in Chapter 3, were not suitable for use as a basis for GVS. This comes from the fact that they all have different purposes and intends than GVS. For example, Graphviz (see Section 3.2) is meant for converting graph representations into nice looking images rather than showing how this task is carried out. AGD on the other hand (see Section 3.3) is flexible, modular and intended for use to design and analyze new algorithms, but is also rather complicated.

4.2 Groundwork

Two in-house information visualization software packages exist at the Institute for Information Systems and Computer Media (IICM) at Graz, University of Technology. The first is the Hierarchical Visualization System (HVS), which is focused on developing new techniques for displaying hierarchically structured data. The second system, the Java Modular Framework for Graph Drawing (JMFGraph), is focused on dynamic and layered graph drawing. HVS and JMFGraph might be seen as the intellectual parents of GVS. Therefore the following section is devoted to these two packages.

4.2.1 The Hierarchical Visualization System (HVS)

The Hierarchical Visualization System (HVS) is, as the name already suggests, a software package devoted to displaying hierarchies and tree-structured datasets [Putz, 2005]. Several different visualization techniques are implemented and can be used simultaneously on the same dataset. This allows users to gain insight from different perspectives, which could not be easily achieved by a single visualization on its own. The different visualizations are synchronized with each other. For example, the user may

select a sub-tree in one visualization and the selection becomes instantly visible in every other visualization. Figure 4.1 illustrates this. Furthermore, HVS supports integrating existing hierarchy visualization solutions into the framework.

HVS is a Java 1.4 and Java Swing application and uses Java design patterns. Most notably, the basic pattern used is the Model View Controller (MVC) design pattern [Buschmann et al., 2004]. The model component of HVS is the data structure, the hierarchy loaded into the system. The view components are visualizations representing the abstract data structure to the user. The controller component is responsible for translating the various user inputs to HVS specific events which manipulate the model and thereby the view. Beside these basic components, HVS contains additional modules for searching and filtering the data.

Concerning the model component, HVS facilitates a strict separation between structure and data [Putz, 2005]. To be strictly accurate, the term "data structure" does not apply to the HVS model component because it actually consists of two parts: the data representation and the structural representation of the hierarchy. This strict separation between data and structural representation allows both entities to change independently. For example, data elements could be rearranged within the hierarchy without being changed or they could be reused multiple times.

To assemble the data model and to assign data to the nodes and leaves of the hierarchy, a factory based approach [Gamma et al., 1997] is used. Using multiple factories, the data model can be created from one of many different data sources transparent to the visualization developer and of course transparent to the user [Putz, 2005]. Furthermore, this approach introduces a standardization of the data elements, allowing them to be searched and filtered. Filters allow parts of the data model to be temporarily hidden.

The controller component of HVS is responsible for capturing user input. When user interaction is performed, such as, for example, the selection of items, or the expansion or hiding of a sub-tree of the hierarchy, or simple navigation within the hierarchy, the user's actions are translated into corresponding events. These events are then broadcast to the currently open visualizations so that they can update their views accordingly. This approach ensures that all views remain synchronized.

Finally, viewer components display the underlying data model and make the data visible to the user. In terms of the observer design pattern [Gamma et al., 1997] they are observers which listen to the various events coming from the controller. On receiving special events, they refresh their display to reflect the changes. An interesting feature to be mentioned here are the VisualizationProperties. VisualizationProperties control the way the node data and attributes are mapped to color. GVS uses a similar approach to unify the appearance of its graphs while still providing rendering flexibility (see Section 4.4.5).

Another aspect of HVS is its plugin-oriented design. Plugins encapsulate parts that can be separated from the rest of the framework and provide a mechanism to load these parts at runtime. Typical examples of such separable parts are the visualizations themselves. GVS plugins (see Section 4.4.8) can be seen as a simple enumeration of items available for execution, rather than a mechanism to load and unload code at runtime.

HVS is already a suitable and well proven framework for visualizing hierarchies. Furthermore HVS has already been used in several courses at the Institute for Information Systems and Computer Media (IICM) and student groups have already implemented new visualizations. From that point of view, HVS has similar goals to GVS (see Section 4.1). This suggests that the concepts implemented in HVS have already proven themselves useful. Therefore, many of them find corresponding counterparts in GVS.

The reason for not basing GVS on HVS was the difference in underlying data models: a tree (HVS) is a special form of a graph (GVS). Thus HVS could be built upon GVS, but not be the other way round. Extending HVS to handle graphs would have been possible with much effort, but would also have destroyed the simplicity of both systems.

4.2.2 The Java Modular Framework for Graph Drawing (JMFGraph)

The Java Modular Framework for Graph Drawing (JMFGraph) is a Java 1.3 framework intended to explore theoretic graph drawing concepts. Although JMFGraph would support general visualization techniques, currently only layered graph drawing algorithms in Sugiyama-style are available [Stedile, 2001]. Like HVS, JMFGraph uses Java Swing to build the user interface and perform the actual drawing.

Graphs are loaded into the JMFGraph system by input modules. Each input module serves as a factory, transforming a certain input format into JMFGraph's proprietary data format. Several modules exist for generating input from different data sources. This is similar to the way GVS uses Generators to gather graph input (see Section 4.4.1). Technically, the input modules transform a Java input stream into a JMFGraph "graph stream" from which then the actual data model of the graph is derived. This way a large part of the graph data assembly can be outsourced from the input module to the framework. Of course this plan of action also complicates the process of graph loading, because not all graph file formats are suitable for streaming.

In JMFGraph, graphs are represented by multi-linked edge maps. Such an edge map, as described in [Stedile, 2001], is an adjacency matrix (see Section 2.2) that does not store empty fields. In other words, for every node, the list of adjacent nodes is stored, which is simply the list of edges to and from that node.

Although the specific design pattern is not explicitly mentioned in [Stedile, 2001], intuitively JMFGraph seems to implement the Model View Controller (MVC) design pattern as well [Buschmann et al., 2004]. The user interface, the layout computation and the actual rendering of the image are all contained in separated modules. Nevertheless, the layout component seems to violate the concept of separation slightly. In JMFGraph, layout algorithms declare what they are capable of by implementing some of the following interfaces:

Local View: The local view layout is computed when the user decides to specially focus on a certain region of the graph. For example, when the user selects and thereby emphasizes a certain node, the layout will be adjusted to center this node and display its link neighborhood (see [Andrews, 2002]).

Global View: In contrast to the local view, the global view calculates the whole layout. This can be seen as the default behavior expected from a graph drawing application.

Step Mode: The step mode computes the layout step by step. The user selects which step of the layout should actually be drawn. Using step mode, the user can trace how the layout algorithm assembles the layout. See Figure 4.2 for an illustration.

Orientation: Through orientation mode, the layout can be told to recompute the layout for different orientations.

The distinction between local and global views on the layout algorithmic level allows to develop algorithms that adapt the layout to the user dynamically, but the orientation of the drawing is definitely nothing the layout algorithm should have to worry about. A layout algorithm may align several parts differently according to the orientation of the drawing area, but the large majority of layout algorithms do not take the orientation of the drawing area into account.

This focus on orientation seems to be mainly introduced by JMFGraph's concentration on Sugiyama-style layouts, which in fact may take the orientation of the drawing into account. This lack of generality, as well as the age of the JMFGraph code make JMFGraph unsuitable to form the basis of GVS. This is the main reason why GVS was implemented as a new Java 1.5 application trying to incorporate all the lessons learned from its predecessors and international siblings.

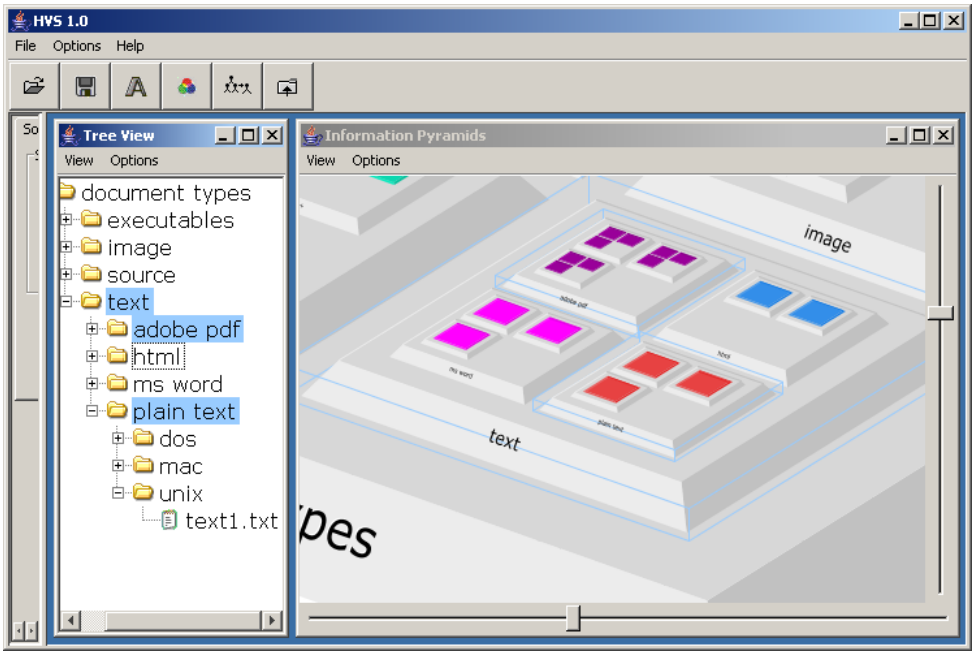


Figure 4.1: The HVS user interface showing the standard tree view as well as the information pyramids visualization of the same hierarchy (see [Putz, 2005] and [Wolte, 1998]). The search panel on the left has been minimized to give more room to the visualizations. The two visualizations are synchronized, as can be seen by the blue selection mark enclosing the same items in both the tree view as well as in the information pyramids.

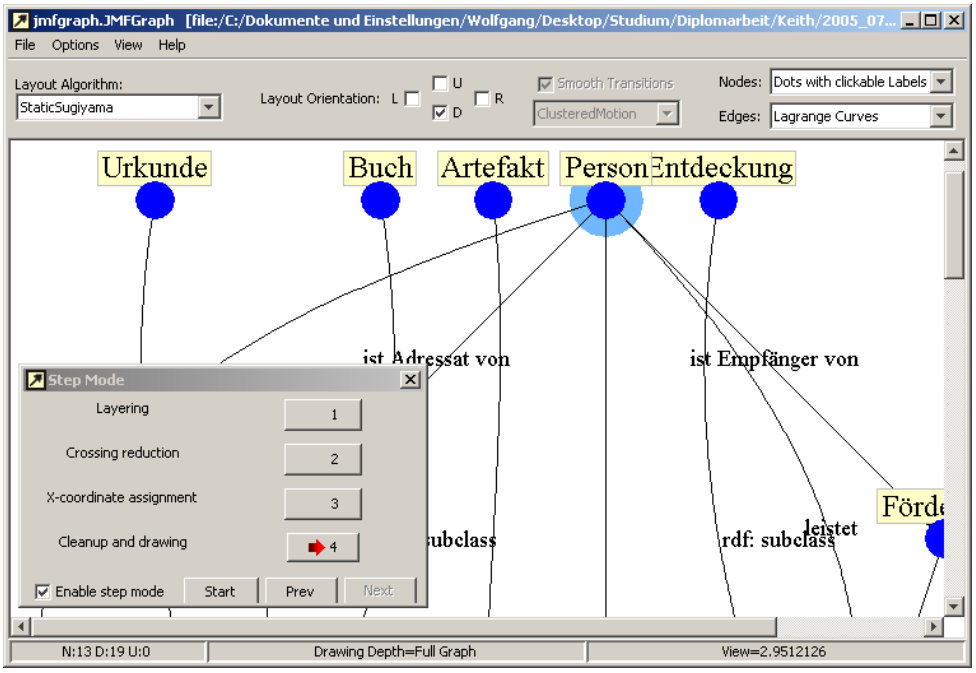


Figure 4.2: The JMFGraph user interface showing a static Sugiyama layered drawing [Battista et al., 1999]. The "Step Mode" control panel in the front allows the user to step through the layout algorithm.

4.3 Key Issues For Designing GVS

Derived from the graph drawing packages presented in Chapter 3 as well as the more closely related ones from the previous sections, this section provides a list of key issues and features of GVS. Proceeding from these key features, the actual software design of GVS, which will be presented in the next section, is straightforward. The following key issues formed the basis of the GVS software design:

Separation of graph loading and graph storage. Various proprietary file formats exist and there are several ways to store graphs (see Section 2.2). A strict separation between the loading of graphs and the representation of graphs within the system is key.

Encapsulation of graph storage. The internal data model of the graph greatly influences the performance of the whole application. It is obviously not a good idea to base the whole system on one, too specialized concept of data access. Therefore, a layer of abstraction [Buschmann et al., 2004] was introduced between the access to the data and its actual storage and representation.

Separation of graph structure and metadata (on graph elements). Graphs define a relationship over entities (see Section 2.1), but they are most often associated with some semantics. There must be ways to store this semantic information, called metadata, about the graph and about graph elements, while still keeping it out of the graph drawing and layout process.

Separation of layout (algorithm) and rendering. Laying out a graph and actually rendering it are two different things. A layout algorithm must not be burdened with technical image generation details.

Encapsulation of layout algorithms. Since the layout algorithm is the real element of interest in GVS, special emphasis was placed on it. It must be easy to implement new layout algorithms and to integrate them into the framework.

Encapsulation of rendering. While layout algorithms define the overall layout of a graph drawing, the creation of the digital image displayed to the user involves completely different techniques. Fortunately, these techniques will be mainly the same for all graph drawings performed in GVS. Therefore, the rendering must be encapsulated into a separate, reusable component.

Algorithm stepping. A graph layout is usually not computed as a whole, but rather in small pieces, step by step. Each step improves the layout a little. By displaying the steps explicitly the user can perceive the progress of the layout and therefore gain insight into the layout algorithm.

Automatic animation between steps. In order to support stepping, animation between the layout steps must be used. Animation allows the user to more easily follow the progress of the layout algorithm.

Simultaneous comparison of different visualizations on the same graph. The most intuitive way to compare different layout algorithms is to simultaneously compute two layouts from the same graph using two different layout algorithms and observe the results. By using stepping, the user can perceive at a glance how the two layout algorithms compute the layout differently.

Simultaneous comparison of the same visualization on different graphs. To better understand a certain layout technique, it is helpful to observe how this technique works on different graphs. Therefore it must be possible to lay out several different graphs simultaneously using the same layout algorithm.

Easy development of new visualizations. To minimize the effort of implementing new visualizations and layout algorithms for the GVS framework, component reuse and modularization must be encouraged.

Before the actual design of GVS could start, two more decisions had to be settled. The first was which programming language, or more precisely, which programming paradigm should be used. The second was which rendering API should be used to perform the rendering of the images on the display.

4.3.1 Choosing the Programming Language

Most layout algorithms described in the main graph drawing literature [Battista et al., 1999] and [Kaufmann and Wagner, 2001] as well as those presented in the papers of the graph drawing community, for example [Eades, 1984], [Kamada and Kawai, 1989] or [Dwyer and Koren, 2006], use a linear, procedural programming model. This would imply using a procedural programming language like C for the implementation of GVS. Older graph drawing packages like Graphviz (see Section 3.2) incorporate this idea.

Of course from the framework's point of view, there are many different modules (objects), interacting to form the application.[1] Hence, an object-oriented programming paradigm seems favorable. This decision is proved by the fact that almost all graph drawing packages presented in Chapter 3 are programmed in object-oriented languages. Although there are several other object-oriented programming languages, C++ and Java are the most common.

While C++ would have had the benefit of performance, Java offers platform-independence and advanced user interface features. The latter and the fact that HVS and JMFGraph (see Section 4.2) were Java Swing programs too settled the decision to implement GVS in Java. In order to keep pace with recent Java developments, the current Java version 1.5 was chosen.[2]

4.3.2 Choosing the Rendering API

Obviously, a graph drawing package has to perform large amount of drawing using a rendering API. The following standardized rendering APIs for Java were considered:

Java3D : Java3D is a 3D scene graph library for Java [Java3D, 2006a]. It provides multiple concepts for rendering abstraction and interaction similar to VRML or X3D [Web3D, 2006]. Although Java3D is partially hardware accelerated through the underlying OpenGL layer, it is definitely one of the slowest alternatives. Since GVS requires fast and simple primitive drawing, it would not be able to utilize the many advanced features provided by Java3D. See [Java3D, 2006b] for more information about the Java3D API.

JOGL: The Java bindings for OpenGL (JOGL) are a lightweight Java wrapper around the underlying OpenGL driver for the corresponding platform [JOGL, 2006a]. JOGL provides the fastest, hardware accelerated high performance rendering in Java. More information about JOGL can be found in [JOGL, 2006b].

Java2D: While JOGL provides the fastest way to perform rendering in Java, Java2D provides the most convenient one [Java2D, 2006]. Since Java 1.3, the Java2D API is fully integrated into the Java 2 Platform and thereby would be available to GVS without additional effort. Java2D provides an API for drawing two-dimensional objects, images, text and so forth. For further information about Java2D see [Sun, 2006b].

If rendering performance were a western movie, JOGL would be "the good", Java3D would be "the bad" and Java2D would be "the ugly".[3] Java3D appears to be a dead-end technology, because rumors

[1] Actually this would lead to parallel programming languages like Ada or Occam. However, these programming languages are hardly known and therefore do not satisfy Point 2 presented in Section 4.1.

[2] The full name is Java 2 Standard Edition runtime version 5.0, SDK version 1.5.

[3] See the Sergio Leone western movie "The Good, the Bad and the Ugly" (1966).

exist that it will not be further developed by Sun Microsystems directly. These rumors are supported by the fact that Java3D has been outsourced from the core platform to the Java developer community (compare [Java3D, 2006b]). At the beginning of the GVS project, it was believed that the full OpenGL hardware acceleration was needed to perform the rendering, but in later phases of development it turned out that actually the layout algorithms were the bottleneck rather than the image rendering. Hence, GVS currently supports both renderers: JOGL and Java2D. By default, the Java native Java2D renderer is activated so that the core GVS package does not depend on the JOGL library (see Section 4.4.6 and Appendix A).

4.4 Software Architecture

The following section describes the GVS software architecture. Besides the actual architecture details, background information is given explaining the reasons why certain parts were developed as described. The GVS software architecture is organized into the following separate modules:

Generators are responsible for generating graphs from various input sources.

Graphs represent the actual data structure of the graphs in the system.

Visualizations are sub-modules of the framework representing particular graph drawing applications.

Layout Algorithms contain the code for laying out the graph step by step.

Drawers decide how the calculated layout is displayed to the user.

Renderers are responsible for rendering this display to the screen.

Animation Engines and Animation Algorithms calculate the animation between the layout steps.

Generators and Visualizations are plugin components. Plugins are special annotated components that are independent of each other and can be removed from and "plugged in" to the framework at will. Generators and Visualizations are the parts of GVS that will most likely be reimplemented in order to add new functionality to the framework. See Section 4.4.8.

4.4.1 Generators

The first step in graph drawing is of course to acquire the graph to be drawn. After the graph has been acquired and all necessary information is available to the layout algorithm, the actual layout can be done. Usually, the graph will be loaded from a file on the local file system, but several other sources are possible as well. For example, the graph could be streamed from some web server or it could be generated randomly from scratch.

Nevertheless, the most common scenario will be loading a graph from a file. Unfortunately there are many different file formats describing graphs. The most popular are the Graphviz dot format (see Section 3.2) and the more recent, XML-based GraphML (see [Brandes et al., 2006]). Several other formats exist that are mostly designed for special purposes describing special graphs [Stedile, 2001].

Since many formats exist, it is infeasible to implement them all at once. Rather, a flexible mechanism is provided which allows Generators to be implemented on a per format basis (compare Section 3.3 and Section 3.4). A Generator in GVS is responsible for generating a graph the system can work with, the so-called AbstractGraph (see Section 4.4.2). In other words, a Generator transforms the proprietary description of a graph, for example contained in a file, into GVS' own data model. All further access to the graph will be made through the AbstractGraph. The Generators and AbstractGraphs shield the rest of the framework from low-level representation details.

Note that the origin of the content generated by the Generator is hidden from the framework as well. Nearly every form of data could be the source to a Generator. The only precondition is that the data is transformable into an AbstractGraph. The specific Generator implementation decides how to perform the transformation.

4.4.2 Graphs

The foundation of any graph drawing software is the data model used to describe graphs. In terms of GVS, the graph data model will simply be called the "graph". Chapter 3 and Section 2.2 already imply the key features a graph data model must have:

- Every graph data model must support a graph consisting of nodes connected by edges (see Section 2.1).
- Both edge traversals and node traversals must be supported.
- Furthermore, graph theoretic properties must be retrievable from the graph independently of the graph's actual representation in memory. This applies especially to adjacency information (the adjacent edges and the adjacent nodes of a node).
- There must be a mechanism for associating semantic metadata with a graph and its nodes and edges.

GVS provides the following three interfaces of graphs:

- BasicGraph
- AbstractGraph
- VirtualGraph.

BasicGraph

The BasicGraph interface is the base interface of all graph interfaces and thus the base interface of the AbstractGraph and the VirtualGraph. The BasicGraph defines the overall methods of data retrieval from a graph, the means for node and edge traversal, as well as methods for gathering graph theoretic information such as the adjacencies of graph entities. An abstract implementation of the BasicGraph interface exists providing several convenience methods to support the developer. All graph theoretic algorithms use the BasicGraph interface in their computations so that they can be reused for the broadest range of applications.

AbstractGraph

A Generator transforms a graph into an AbstractGraph. An AbstractGraph is a static representation of the information conveyed by the graph as well as the metadata assigned to the graph's elements, such as node labels or edge weights. The AbstractGraph interface is designed for static access only. Once the AbstractGraph has been created, its structure is fixed and must not be changed.

One important feature of GVS is to launch multiple visualizations of the same graph to show their differences (see Section 4.3). It would be infeasible to load the graph over and over again, or to duplicate the graph whenever a new visualization is launched.[4] An AbstractGraph can be seen as a static template, from which possibly multiple VirtualGraphs are created. The AbstractGraph works as a bridge of communication between the various VirtualGraphs that were created from it.

[4] Actually the graph structure is duplicated by the VirtualGraph to create a local working copy for the visualization, but the metadata is shared. See [Prinz, 2006] for details.

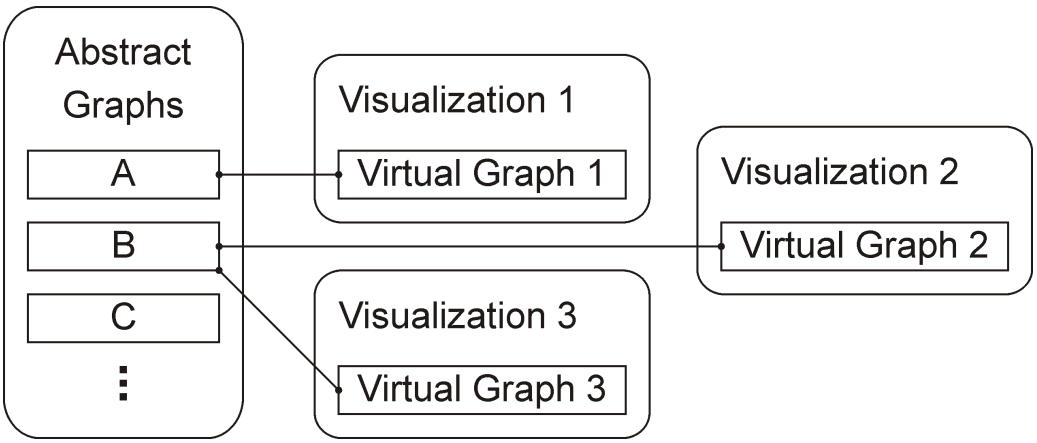


Figure 4.3: Multiple Visualizations, actually GenericVisualizations, in a typical GVS session. Note that Visualization 2 and 3 were launched from the same AbstractGraph B. Hence, they have the same underlying global AbstractGraph B, while they have own, local VirtualGraphs 2 and 3.

VirtualGraph

AbstractGraphs are static and must not be changed after their creation, but many layout algorithms modify the graph they are working on. Trivially, all layout algorithms will change the positions of the nodes and edges of the graphs, because that is what they are supposed to do. Some layout algorithms may even introduce new nodes or edges to the graph, so-called dummy nodes and edges, or others may remove such from the graph while operating. Nevertheless, because potentially many different LayoutAlgorithms may work on the same underlying AbstractGraph, it is crucial that they are not allowed to influence each other.

In order to ensure this separation, a local copy, the VirtualGraph, is made of the global AbstractGraph. Each Visualization has its own VirtualGraph, where each VirtualGraph references back to their original, underlying AbstractGraph. Figure 4.3 illustrates this concept. The VirtualGraph is a more flexible data structure though. It allows nodes and edges to be added and removed from the VirtualGraph dynamically, as the Algorithm requires.

Another addition in the VirtualGraph interface is that a VirtualGraph stores the positions of the nodes and the routings of the edges directly. Although these properties could also be stored as MetadataObjects on the corresponding graph elements, it is because of their ubiquitous importance in every layout algorithm that they have been integrated directly into the VirtualGraph interface. Furthermore, storing these very important properties directly has performance advantages.

Although dynamic graph drawing is not directly supported by GVS (see Section 2.4.4), it is possible to implement dynamic graphs using VirtualGraphs. Visualizations are free in choosing how to perform their layout tasks (see Section 4.4.3), so a dynamic graph drawing application could, for instance, bypass the static AbstractGraph by invoking a dynamic graph loading Generator directly coupling it internally to a VirtualGraph. Another way to realize dynamic graph drawing without bypassing the AbstractGraph would be to load a minimum, static skeleton, possibly not consisting of any nodes or edges, as an AbstractGraph and then let the Visualization or the LayoutAlgorithm decide how to assemble the graph dynamically. For example, when a graph is dynamically streamed from an online data source, the corresponding Generator could create an empty Abstract graph containing the stream information as metadata. A dynamic Visualization could then use this metadata to stream and process graph elements step by step.

Metadata

In GVS, graph entities like nodes and edges are mapped to singleton, unique objects. This allows metadata to be attached to these objects in a singleton, unique way. Metadata itself is represented by MetadataObjects which identify themselves via textual keys. The key is used to distinguish between several MetadataObjects attached to the same graph entity.

For example, a graph node could have the properties color and label. To annotate a node with these properties, two different MetadataObjects will be attached to the node, one holding the color and the other containing the label. Both MetadataObjects must have different, unique keys, for example, "COLOR" and "LABEL". The corresponding MetadataObjects can be retrieved from the node using these keys.

4.4.3 Visualizations

Loading the graph into the system is the first step in graph drawing. Drawing the graph is the second step. Before the graph can be drawn, it has to be laid out on the drawing area by the LayoutAlgorithm. Whereas it is the intend of most graph drawing systems to calculate this layout in the background as fast as possible and transparent to the user, GVS uses a completely different approach. This comes from the fact that GVS is supposed to visualize how the layout is assembled (see Section 4.1 and Section 4.3). Stepping and animation are used in GVS Visualizations to display the process of layout creation rather than just showing the final image.

In GVS, layout and rendering are done within a so-called Visualization[5]. A Visualization can be seen as the (graph-) data sink, when a Generator is a data source. The Visualization is responsible for displaying the user interface, to input necessary parameters for the layout, for performing the layout in a demonstrative way, and for displaying it to the screen.

This makes the Visualization itself a very large component. Since most visualizations will only differ in the LayoutAlgorithms, it would be infeasible to reimplement all Visualization parts over and over again. Therefore, GVS provides a GenericVisualization implementation that can be assembled from various other components on demand. For example, a visualization for explaining a certain, new layout algorithm may equip the GenericVisualization with the standard Drawers, Renderer and user interface provided by the framework. So the GenericVisualization acts as a mediator in the sense of the mediator design pattern [Gamma et al., 1997].

A GenericVisualization in general has to process user input and to produce graphical output. Invisible to the user, the LayoutAlgorithm and the VirtualGraph form the underlying data model. The GenericVisualization's architecture follows the Model View Controller (MVC) design pattern [Buschmann et al., 2004]. According to this, the components of a GenericVisualization implementation can be categorized into the following three groups (see Figure 4.4):

Model: Together, a VirtualGraph and a LayoutAlgorithm form the GenericVisualization's internal data model. The Model contains all functions that change the Visualization's internal state, which is the graph.

View: The View component is responsible for transporting the information provided by the graph and the graph's layout to the user. In terms of graph drawing, the View transforms the abstract layout of a graph into some image displayed on the screen. The Drawers and the Renderer in cooperation with the AnimationEngine and its AnimationAlgorithm define the appearance of the graph on the screen.

Controller: Via the graphical user interface, the Controller allows the user to steer the GenericVisualization and to set parameters and options. Unlike the Model and the View, the Controller does not

[5] In this thesis, the term "Visualization" (with a capitalized first letter) denotes the corresponding GVS Visualization interface (see [Prinz, 2006]), whereas the word "visualization" (all lower case) means a visualization in its general meaning.

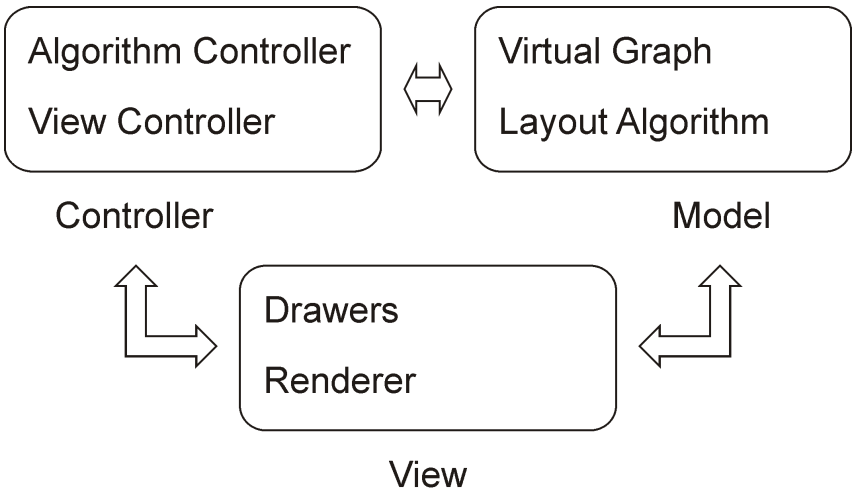


Figure 4.4: Illustration of the Model View Controller (MVC) Design Pattern used to organize the components of a GVS GenericVisualization. Note that the controller component is actually assembled from the controllers of the View and the Model components.

have much of its own functionality. The Controller is just a lightweight input device that manages properties of the Model and the View. Actually it is assembled from the user interfaces provided by various other components of Model and View.

It should be mentioned here that the GenericVisualization does not strictly follow the MVC pattern as described in [Buschmann et al., 2004]. This is mostly because of the weak Controller component that is divided amongst Model and View. Strict separation of the Controller from Model and View would have been too much effort with too little benefit. Furthermore, the split Controller allows the user interface to be built close to where it belongs. For example, a LayoutAlgorithm knows best which parameter it needs so it is reasonable that the LayoutAlgorithm provides the user interface to itself. Introducing a separate component for the user interface here would have caused a gap between the parameter set of the LayoutAlgorithm and the parameters present in the user interface. A similar architectural design pattern to MVC is the Presentation Abstraction Controller (PAC) [Buschmann et al., 2004]. In some aspects, the GenericVisualization in combination with its usage in the rest of the framework would also fit the PAC pattern.

4.4.4 LayoutAlgorithms

The LayoutAlgorithm receives a VirtualGraph as input and contains all code necessary to lay that graph out on the drawing area. In order to do that, the LayoutAlgorithm has to perform several layout steps. Each layout step should modify the layout slightly so that the modification is observable by a human user. Typically the user will trigger the steps using the user interface (see Section A.4.1).

After every layout step, the display is updated to represent the new layout. This way the user can track the progress of the layout generation directly. The force-directed methods for calculating graph layouts are good examples for stepping, because they already incorporate a number of iterations in their very algorithms.

Note that a LayoutAlgorithm is completely unrestricted in placing nodes and edges. How the nodes are spread out, how edges are routed between them, and how much space is left for labels (see Section 2.1.2) is decided solely by the LayoutAlgorithm. However, a LayoutAlgorithm should never contain presentation- or rendering-dependent code. If a LayoutAlgorithm suggests a certain way of presentation,

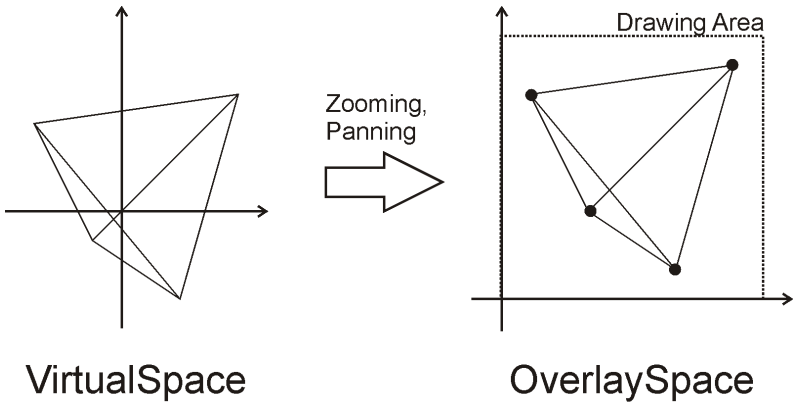


Figure 4.5: Illustrations of the VirtualSpace and OverlaySpace. The graph is laid out in VirtualSpace. Zooming and panning transform the graph into OverlaySpace, which is mapped onto the drawing area.

it should pass this information to the corresponding Drawers via the metadata mechanism. For example, a LayoutAlgorithm may leave out rectangular areas around the nodes so that node labels can be placed there. The dimensions of that areas should be annotated by metadata objects so that the corresponding Drawer can paint the node labels correctly.

4.4.5 Drawers

While the LayoutAlgorithm simply assigns positions to the graph elements of the VirtualGraph, the Drawers decide the look of the layout. For example, the LayoutAlgorithm may place a node to a certain position, but the actual symbol that is drawn, its color, and any textual label is defined by the corresponding Drawer. Since the drawing of nodes and edges differ significantly, and drawing into the background is different from drawing an overlay foreground, special Drawers exist for these special purposes. Nevertheless, the discussion here applies to all sorts of Drawers.

Node locations and edge routings are set by the LayoutAlgorithm. In order to hide the details of screen resolution, window size and so forth from the LayoutAlgorithm, all coordinates set by the LayoutAlgorithms are transformed before being drawn to screen. The LayoutAlgorithm places the nodes and edges in an infinite, two-dimensional space, called the VirtualSpace. This VirtualSpace is mapped to the so-called OverlaySpace. The OverlaySpace is a coordinate space which is pixel aligned to the window rectangle.

The mapping from VirtualSpace to OverlaySpace allows the implementation of panning and zooming completely independently of the LayoutAlgorithm and the Drawers. For instance, zooming only influences the scale factor which transforms VirtualSpace into OverlaySpace coordinates. Panning is realized by a simple, two-dimensional translation in OverlaySpace. Figure 4.5 illustrates this. All the transformations between the VirtualSpace and the OverlaySpace are maintained by the GraphScene. The GraphScene contains the mathematical operations that need to be performed.[6]

After translating the node and edge coordinates from VirtualSpace to OverlaySpace, the associated Drawers start building the final image of the layout. By taking the various properties and options the user may specify into account, the Drawers create the drawing primitives, such as rectangles or polygons, which represent the graph entities on the screen. For example, a node will usually be drawn by a small

[6] In terms of OpenGL, the GraphScene could be seen as the GVS counterpart to the model-view matrix [Shreiner et al., 2003].

circle filled with some color. These primitives are passed to the corresponding Renderer for rendering to the screen.

4.4.6 Renderers

While the Drawers decide how the layout is displayed to the user, the Renderers actually carry out these operations. Renderers are wrappers in terms of the Adapter design pattern [Gamma et al., 1997] around the underlying rendering APIs, such as Java2D or JOGL (see Section 4.3.2). A Drawer shields the LayoutAlgorithm from representation details. A Renderer shields the Drawer from underlying technical details of the rendering API. Using these two levels of abstraction [Buschmann et al., 2004], the rendering API may be exchanged independently without influencing the representation, while at the same time the representation may be modified independently.

Every Renderer must provide its own implementation of the DrawingInterface. The DrawingInterface is a collection of drawing commands independent of any particular rendering API which can be executed by Drawers. For example, such commands include drawing of a rectangle, drawing lines, drawing text and so forth. All drawing operations in the DrawingInterface work in the coordinate system of the OverlaySpace, which eliminates the various problems that arise between the proprietary coordinate systems of different rendering APIs.

4.4.7 AnimationEngine and AnimationAlgorithm

In every step, LayoutAlgorithms assign new positions to the nodes. Depending on the step size and the specific LayoutAlgorithm used, these new positions might differ from the old ones dramatically. When the difference between old and new positions is too large, the user will not be able to follow the change. To avoid this loss of the mental map, animation is introduced to smoothly transfer the old positions into the new ones. Users can track the nodes' motion to their new positions.

Animation in terms of GVS requires the computation of a series of intermediate layouts which blend from the old positions to the new ones. The algorithms to calculate these interpolated layouts are called AnimationAlgorithms. Although several different ways of interpolating these layouts exist (see [Tatzmann, 2004]), simple linear interpolation between old and new node positions is sufficient for the first version of GVS.

An AnimationEngine is responsible for carrying out the animation. Since neither the LayoutAlgorithm nor the Drawers must be aware of the animation happening, the AnimationEngine works as an adapter between these two components. In summary, the AnimationEngine transforms a transition from one layout to another computed by the LayoutAlgorithm into a sequence of transitions according to the AnimationAlgorithm. This sequence is then displayed by the Drawers.

4.4.8 Plugin Management

Generators and Visualizations are plugins, a piece of program code not directly part of the main framework. Typical applications for plugins would be new Generators or new Visualizations. Plugins are used in many applications. In web browsers, such as Microsoft Internet Explorer or Mozilla Firefox, plugins enhance the web browser with new functionality from third parties without modifying the original program. In terms of graph drawing, the new functionality to be added to GVS will most likely be a new LayoutAlgorithm and of course some appropriate Visualization, although the following discussion about GVS plugins applies to Generators as well.

GVS plugins are tightly coupled to the GVS framework. This means that they are executed within the same Java Virtual Machine as the rest of the framework. In contrast, loosely coupled plugins would run completely separate from the system, communicating with the framework using well-defined protocols.

Examples of loosely coupled plugins are audio or video player plugins for web browsers. The main advantage of loosely coupling a plugin to the framework is that the looser the connection is, the less the parts influence each other. The drawback of this independence is that the extensive use of the framework and its components is limited as well. Since GVS Generators and Visualizations should be as lightweight as possible, tight coupling is favored.

In GVS, plugins are implemented as factories [Gamma et al., 1997] which produce their certain type of application. For example, the FruchtermanReingoldVisualizationPlugin creates a GenericVisualization that uses the FruchtermanReingoldLayoutAlgorithm (see Appendix B). Considering this particular implementation, the plugin part itself only consists of a few lines of code. Most implementation effort lies within the LayoutAlgorithm.

Chapter 5

Selected Details of the Implementation

" ...within 100 years, computers will be twice as powerful, 10,000 times larger, and so expensive that only the five richest kings of Europe will own them. "

[Professor Frink, The Simpsons episode 3F20.]

This section contains the documentation of selected details of the implementation that are of particular interest. Basically these details are the mathematical background behind the layout algorithms implemented in GVS. A brief history of the development of force-directed methods is described. StressMajorization [Gansner et al., 2005] is one mathematical technique to numerically solve the problems introduced by force-directed placement. Using StressMajorization and special equation solvers which take hierarchical constraints into account, the Dig-CoLa layout algorithm was developed [Dwyer and Koren, 2006].

5.1 Force-Directed Methods

One of the strongest aesthetic constraints in graph drawing is to place related entities close together and to move unrelated ones further apart (see Section 2.3). This already suggests describing these attractive and repulsive interactions by physical force models. In nature, several real-world models exist incorporating such behavior. For example, electrically charged spools align themselves in an electromagnetic field. Masses connected by springs rearrange themselves until the whole system reaches a state of minimal kinetic energy. All these physical models are very similar in their mathematical foundations.

One of the first methods that applied such a physical model for graph drawing was Eades' spring model [Eades, 1984]. This algorithm treats nodes as physical masses, initially randomly located on the drawing area. If nodes are adjacent, their corresponding masses are connected by logarithmic springs. The masses of non-adjacent nodes repel each other by forces inversely proportional to their distances. The springs produce forces that draw adjacent nodes together, while the repulsive forces prevent the whole layout from clumping together. In a first step, the forces on each mass are calculated and then, in a second step, all masses are moved proportionally to these forces. These steps are iteratively repeated until the layout reaches a sufficiently stable state.

Although this algorithm has several disadvantages (see [Eades, 1984] and [Gansner and North, 1998]) it main advantage is its simplicity. The algorithm is easy to understand, very easy to implement, and produces aesthetically appealing results. The spring-based or force-directed methods became very popular in graph drawing. The most popular improvements of Eades are Fruchterman and Reingold [1991] and Kamada and Kawai [1989]. Further improvements on the runtime performance of spring-based models have been done by Chalmers [1996], Morrison et al. [2003], and Morrison and Chalmers [2004].

Figure 5.1: The Eades [1984] and Fruchterman and Reingold [1991] layout algorithms. Figure (a) shows the layout produced by the GVS Eades visualization. Figure (b) shows the layout composed by the Fruchterman and Reingold visualization for the same graph. Note that the nodes are more expanded in the Fruchterman and Reingold layout.

Instead of using logarithmic springs and thereby physical laws, Fruchterman and Reingold use their own proprietary laws to compute the forces. As in Eades, attractive forces are only calculated between adjacent nodes, whereas repulsive forces exist between all node pairs, which includes adjacent ones. The special feature of Fruchterman and Reingold is that the forces are applied so that they distribute the nodes evenly across the predefined drawing area. This gives Fruchterman and Reingold layouts a more expanded look, whereas Eades layouts seem to be clumped together. See Figure 5.1 for an illustration.

In contrast to Eades and Fruchterman and Reingold, Kamada and Kawai use graph theoretic distances as a measure during the node placement process instead of the adjacency information. While the adjacency information can be easily retrieved from graphs (see Section 2.2), the graph theoretic distances have to be calculated explicitly. The benefit of graph theoretic distances is that they introduce a measure between any pair of nodes which directly reflects the distance between these nodes.

Using the graph theoretic distance $d_{i,j}$, where $1 \leq i \leq n$, $1 \leq j \leq n$, $i \in \mathbb{N}$ and $j \in \mathbb{N}$, between two arbitrary nodes $v_i \in V$ and $v_j \in V$ of the Graph $G(V, E)$, the stress $stress(X)$ of an $n \times d$ layout X, where n is the number of nodes and $d \in \mathbb{N}$ is the dimensionality of the layout, is formulated by:

$$stress(X) = \sum_{i<j} w_{i,j} \left(||X_i - X_j|| - d_{i,j} \right)^2$$

The $n \times d$ layout X is just a compact matrix representation of the d-dimensional coordinates of the nodes on the drawing area. The dimensionality d is typically 2, sometimes 3 or higher. The ith row in X is denoted by X_i and contains the coordinates of the node v_i. Thereby the norm $||X_i - X_j||$ just measures the distance between the positions of the nodes v_i and v_j on the drawing area. The stress function $stress(X)$ of a layout X sums up the derivation of this distance of every node pair in the layout compared to the ideal, graph theoretic distance $d_{i,j}$ of the pair. The normalization constant w_{i_j} is given as $d_{i,j}^{-\alpha}$ with $\alpha = 2$ and simply corrects the square sum.[1]

The $stress(X)$ is the sound mathematical formulation of the aesthetic criteria to place related things close together. Hence, the goal of every force-directed graph drawing algorithm is to minimize this

[1] For further suitable values of α see [Kamada and Kawai, 1989] as well as [Gansner et al., 2005] and [Cohen, 1997].

$stress(X)$ to obtain an appealing layout X. The Kamada and Kawai [1989] layout algorithm iteratively computes partial derivatives of $stress(X)$ to perform a gradient descent and adjusts the nodes so that the final layout is a local minimum of $stress(X)$. The disadvantage of this layout algorithm is that it computes only a local minimum, where usually a global minimum is desirable.

The problem of minimizing the stress function $stress(X)$ in reasonable time is also found in multi-dimensional scaling. Chalmers [1996] uses a spring-based stress model to compute a two-dimensional layout of a high-dimensional dataset. In terms of graph drawing, this problem is equivalent to producing a two-dimensional drawing of a full graph using a proprietary distance metric. Because datasets are usually very large in multi-dimensional scaling applications, Chalmers [1996] needed to improve the square runtime per iteration of common force-directed algorithms, such as, for example, Eades.

A stochastic approach is used by Chalmers [1996] to achieve linear runtime per iteration. Instead of taking all forces between all nodes into account in each iteration, only the forces between each node and a constant number of wisely chosen other nodes are calculated thus producing linear runtime. These key-nodes are chosen iteratively so that a majority of the unchosen nodes is close to them. Additionally, randomness ensures the correct choice of the key-nodes. Further improvements of this algorithm are presented in [Morrison and Chalmers, 2004] and [Morrison et al., 2003].

5.2 StressMajorization

Minimizing the $stress(X)$ is not an easy task, mainly because the $stress(X)$ function has many local minima. The Kamada and Kawai method can only guarantee to find one such local minimum, which may be fairy large. The StressMajorization[2] method tries to solve this problem by redefining the $stress(X)$ function so that the new function has only one, and thus a global, minimum. Of course several other approaches exist that, in a wider sense, try to overcome these shortcomings, such as [Harel and Koren, 2002a], [Harel and Koren, 2001], [Davidson and Harel, 1996], [Harel and Koren, 2002c], [Koren and Harel, 2005] as well as [Koren, 2003] and [Koren, 2005].

The Laplacian matrix L^w of the normalization constants $w_{i,j}$ is defined as [Gansner et al., 2005]:

$$L^w = L_{i,j} = \begin{cases} -w_{i,j} & i \neq j \\ \sum_{k \neq i} w_{i,k} & i = j \end{cases}$$

For a given $n \times d$ layout X, the matrix L^X is defined as [Gansner et al., 2005]:[3]

$$L^X = L^X_{i,j} = \begin{cases} -\delta_{i,j} \operatorname{inv}(||X_i - X_j||) & i \neq j \\ -\sum_{j \neq i} L^X_{i,j} & i = j \end{cases}$$

where $\delta_{i,j} = w_{i,j} d_{i,j}$ and

$$\operatorname{inv}(x) = \begin{cases} \frac{1}{x} & x \neq 0 \\ 0 & x = 0 \end{cases}$$

Using these matrix definitions, the stress function $stress(X)$ can be rewritten in matrix notation. An analysis of the square terms of the sum in $stress(X)$ shows that the stress function can be bounded from above (see [Gansner et al., 2005]). This boundary allows an iterative approach to calculate a series of layouts $X(t)$, where $0 < t$ and $t \in \mathbb{N}$, that sequentially lowers the stress. Or, in other words, $stress(X(t+1)) \leq stress(X(t))$. The new layout $X(t+1)$ can be calculated using the old layout $X(t)$ by:

$$L^w X(t+1) = L^{X(t)} X(t)$$

[2] As stated in [Gansner et al., 2005], the technique is called "stress majorization". In order not to confuse this technique "stress majorization" with the stress function $stress(X)$, "stress majorization" is called StressMajorization in this document.

[3] In literature, the matrix L^X is also denoted as L^Z for a given $n \times d$ layout Z.

StressMajorization computes the final layout by solving this equation in every iteration step using a standard conjugate gradient solver for each axis separately. Several other solvers are described in [Gansner et al., 2005] as well. The iteration is performed until the change in stress between subsequent layouts decreases below a certain tolerance bound.

5.2.1 Conjugate Gradient

The conjugate gradient method used by StressMajorization is described in [Weisstein, 2002a] and [Luenberger, 2003]. The conjugate gradient solver is also used by the Dig-CoLa algorithm to compute the optimal arrangement of the y-axis (see Section 5.3.1). A detailed listing of the conjugate gradient algorithm used by the GVS implementation can be found in [Carmel et al., 2004].

The simple gradient descent method tries to reach the nearest local minimum by going down the path of the steepest gradient [Weisstein, 2002b]. This has the drawback that many iterations are required to reach the minimum if the gradient is very flat. Instead of following the gradient directly, conjugate gradient uses the residuals of the function to refine the direction of every step [Weisstein, 2002a; Luenberger, 2003]. The following notation is used:

$$\underbrace{L^w}_{A}\underbrace{X}_{x}(t+1) = \underbrace{L^{X(t)}X(t)}_{b}$$

With the direction $d(t)$ and the residuals $r(t)$ initialized to $d(0) = r(0) = b - Ax(0)$, the following steps are iteratively performed until the change between successive iteration steps is sufficiently small.

1. The step size α is computed taking the residuals $r(t)$ and the direction $d(t)$ into account (see [Weisstein, 2002a]):
$$\alpha = \frac{r(t)^T r(t)}{d(t)^T A d(t)}$$

2. The solution $x(t+1)$ of the iteration step is achieved by making a step of size α with given direction d:
$$x(t+1) = x(t) + \alpha d(t)$$

3. The new residuals $r(t+1)$ are calculated similarly according to the step size α:
$$r(t+1) = r(t) - \alpha A d(t)$$

4. In order to compute the new direction $d(t+1)$ for the next step, the orthogonalization factor β is needed. β ensures that the new direction $d(t+1)$ and the new residuals $r(t+1)$ are orthogonal to all previous directions and residuals [Weisstein, 2002a].
$$\beta = \frac{r(t+1)^T r(t+1)}{r(t)^T r(t)}$$

5. Finally, the new direction $d(t+1)$ is assembled using the orthogonalization factor β:
$$d(t+1) = r(t+1) + \beta d(t)$$

As suggested in [Carmel et al., 2004], the iteration is stopped when the (Euclidean) norm of the residuals $r(t+1)$ falls below a certain level ϵ ($|r(t+1)| \leq \epsilon$). In the GVS implementation this level ϵ was chosen to be $\epsilon = 0.001$.

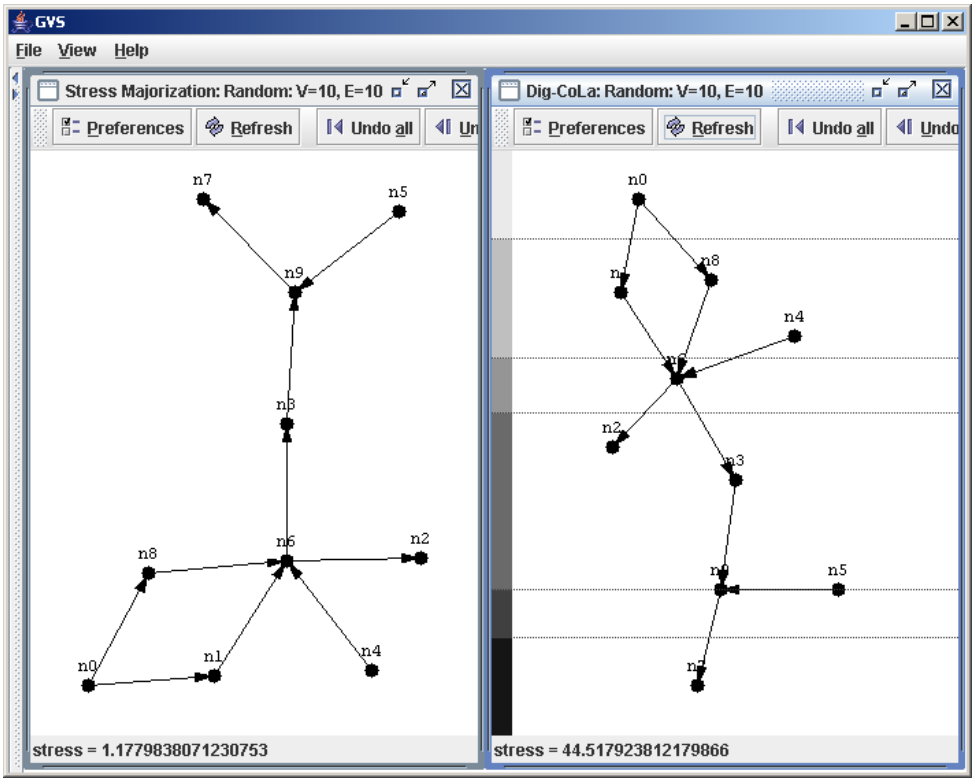


Figure 5.2: Comparison of the StressMajorization (left) and the Dig-CoLa (right) drawings of the same graph as implemented in GVS. The StressMajorization technique produces an unconstrained layout similar to those shown in Figure 5.1, whereas the Dig-CoLa algorithm constrains the placement of the nodes according to directional flow given by the edge directions.

5.3 Dig-CoLa

StressMajorization has the benefit that each axis of the layout can be computed separately while the monotonic decrease of the stress function $stress(X)$ is still guaranteed. This allows additional constraints to be applied to the placement of nodes along each axis. The Directed graph (drawing) with Constrained Layout (Dig-CoLa) algorithm exploits this by introducing the hierarchical information implied by edge directions of the graph to one axis. Dig-CoLa thereby produces layouts similar to the layered layouts [Sugiyama, 2002], but still with the speed and flexibility of force-directed placement. See Figure 5.2 for a comparison of StressMajorization and Dig-CoLa.

5.3.1 Y-Axis Optimal Arrangement

The first step of Dig-CoLa is to partition the nodes into layers along one axis, which is usually the y-axis. This is done by minimizing the hierarchical energy $E_H(y)$ of the n-dimensional vector y (see [Carmel et al., 2004], [Carmel et al., 2002] and [Koren and Harel, 2004]). The hierarchical energy $E_H(y)$ is defined similar to the stress function $stress(X)$ used by StressMajorization [Dwyer and Koren, 2006]:

$$E_H(y) = \sum_{e \in E, v_i \in e, v_j \in e} (y_i - y_j - \delta_e)^2$$

where δ_e gives the relative hierarchy between the nodes v_i and v_j that form the edge e [Dwyer and Koren, 2006].

Iteratively minimizing the hierarchical energy $E_H(y)$ leads to the optimal arrangement for the y-axis. This means that the nodes are placed along the y-axis so that the edges between them show a distinct top-to-bottom flow. The optimal arrangement can be calculated by the conjugate gradient algorithm described in Section 5.2.1. Several other solvers could be used as well [Carmel et al., 2004]. While the optimal arrangement could already be used as the layout of the y-axis with the other axis computed by StressMajorization, this would produce very restrictive layouts.

It can be observed that nodes in the optimal arrangement tend to cluster together [Dwyer and Koren, 2006]. This behavior is exploited by Dig-CoLa to group these nodes together into layers. New layers are introduced whenever the gap between succeeding nodes along the y-axis exceeds a certain threshold. The straightforward algorithm to partition the nodes into layers according to the optimal arrangement can be found in [Dwyer and Koren, 2005] and [Dwyer and Koren, 2006]. This form of layering is also superior to the one used in the Sugiyama algorithm, because the optimal arrangement approach does not require the removal of cycles in the graph (see [Sugiyama, 2002; Battista et al., 1999] and [Dwyer and Koren, 2005]).

5.3.2 Quadratic Programming with Orthogonal Constraints (QPOC)

Partitioning the nodes into layers allows these layers to be constraints in the StressMajorization process of the y-axis. The x-axis is still calculated as described in Section 5.2. The layout $y(t+1)$ of the y-axis is obtained by minimizing the following expression [Dwyer et al., 2005a]:

$$y(t+1)^T L^w y(t+1) - 2y(t+1)^T L^{X(t)} y(t)$$

subject to the constraints introduced by the levels. These constraints enforce that a node in a lower level is not placed higher than a node in a higher level. This constrained quadratic minimization problem can be solved using quadratic programming with back projection. Further, but less performant solutions to this problem are mentioned in [Dwyer et al., 2005a]. Actually, Dwyer and Koren [2006] present an improved version of Dig-CoLa, which allows the specification of minimal gaps between the levels in the layout. Nevertheless, the algorithm implemented in GVS and presented in the following is described in [Dwyer et al., 2005a] and does not take account of minimal gaps.

The Quadratic Programming with Orthogonal Constraints (QPOC) solver calculates the y-axis layout using the following steps repeatedly until the change of the layout per step is sufficiently small.[4] For simplicity, the notation $A = L^w$ and $b = 2L^{X(t)}y(t)$ is used.

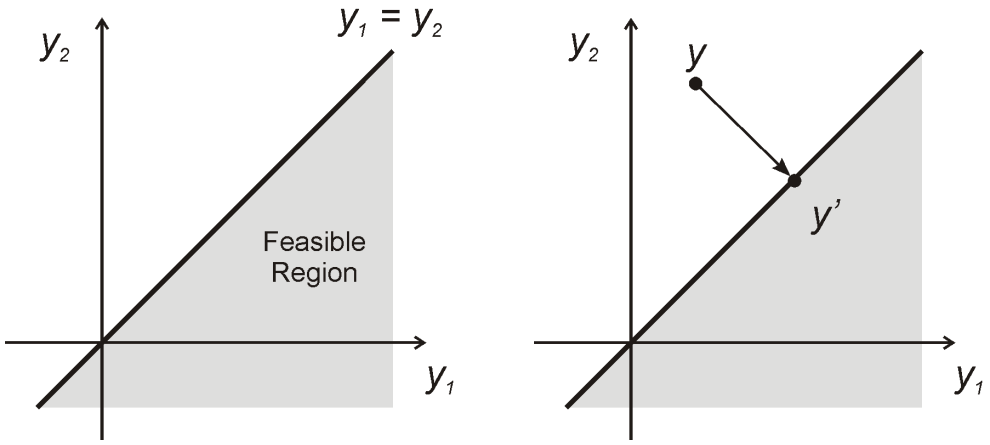
1. A new layout y is computed by performing a step of size s from the old layout $y(t)$ along the gradient g:

$$\begin{aligned} g &= 2Ay(t) + b \\ s &= \frac{g^T g}{g^T A g} \\ y &= y(t) - sg \end{aligned}$$

2. Because this new layout may violate the constraints, the new layout y is projected back to the closest layout y' that does not violate the constraints (see Section 5.3.3):

$$y' = \text{project}(y)$$

[4] It may be especially noted here that the *change per step* $|y(t+1) - y(t)|$ must be sufficiently small. This means that $|y(t) - y(t-1)| - |y(t+1) - y(t)| < \epsilon$. The GVS implementation of QPOC uses $\epsilon = 0.01$.



(a) The Feasible Region **(b)** Projection of y to y'

Figure 5.3: Illustration of the projection mechanism applied to a two-dimensional layout space $y_1 \times y_2$. Figure (a) shows the feasible region for the layout constraint $y_1 \geq y_2$. Figure (b) demonstrates the normal projection of an unfeasible layout y to the closest feasible layout y' on the feasible region.

3. The actual descent step d of step size α is then performed toward this projected layout y' to get the new layout $y(t+1)$.

$$\begin{aligned} d &= y' - y(t) \\ \alpha &= \max\left(\frac{g^T d}{d^T A d}, 1\right) \\ y(t+1) &= y(t) + \alpha d \end{aligned}$$

The back projection of the layout ensures that the final solution fulfills the layering constraints, while it is still close to the stress minimized optimal solution as would have been calculated by StressMajorization.

In summary, the Dig-CoLa layout algorithm combines the benefits of the StressMajorization technique with the ability to convey hierarchical information. In this manner Dig-CoLa supersedes the much older Sugiyama layered drawing method. Furthermore Dig-CoLa does not suffer from the major drawback of the Sugiyama layouts, namely that the graphs have to be cycle-free.

5.3.3 Projection

A simple gradient descent approach alone would sometimes compute layouts violating the layering constraints. Therefore, such a solution must be modified so that it does not violate the constraints. A specific layout for n nodes can be seen as an n-dimensional vector in an n-dimensional vector-space. The constraints on the layout thereby partition this n-dimensional vector-space into feasible and unfeasible regions. If a layout vector turns out to lie within an unfeasible region, it must be transformed into a vector that lies within a feasible region. This transformation can be seen as a projection of the vector onto the feasible region [Dwyer et al., 2005a]. The problem with projection is that it must modify the original vector, which was computed to be an optimal solution to the equation system presented in Section 5.3.2. Therefore the projected vector should be as close to the original as possible (see Figure 5.3).

For example, consider a very small graph $G(V, E)$ consisting of exactly two nodes v_1 and v_2 and one single edge e directed from v_1 to v_2. The optimal arrangement and the succeeding partitioning to layers will place the nodes into two different layers. Node v_1 will be placed into the first layer, above

the second layer, which will contain node v_2. In this example, a complete y-axis layout can be seen as a two-dimensional vector y in a two-dimensional vector-space $y_1 \times y_2$, called the layout space. Now, the ordering constraint introduced by the layering forces the location y_1 of node v_1 to be higher than the location y_2 of node v_2 ($y_1 \geq y_2$). Thereby the feasible region of the two-dimensional layout space, the $y_1 \times y_2$ plane, is simply the lower triangle of that plane. Figure 5.3(a) illustrates this. A layout y that lies in the upper triangle must be projected to a new position y' in the lower triangle.

In order to preserve as many features of the original, but unfeasible layout y, the new layout y' must be as close to it as possible. In other words, the (Euclidean) distance $|y-y'|$ must be minimal. The layout y' in the feasible region closest to the original layout y is given by the normal projection of y onto the line $y_1 = y_2$. See Figure 5.3(b) for an example of this projection. Of course, this projection mechanism becomes immediately more complex when more levels and more nodes are involved.

The implementation of the full projection algorithm is rather complicated and presented in detail in [Dwyer et al., 2005a]. In summary, this projection implementation ensures that for each level, all nodes in and above that level in the hierarchy are positioned above the nodes below that level. The projection is performed by iteratively moving nodes that violate this rule upward in the layout for each level.

5.3.4 Discussion of Dig-CoLa

The Dig-CoLa layout algorithm can be seen as a competitor to the Sugiyama layout algorithm for layered graph drawing [Dwyer and Koren, 2006]. A drawback of the Sugiyama algorithm is that it requires the graph to be cycle-free. This means that cycles, or more precisely, edges pointing against the flow, have to be removed from the graph explicitly in a preprocessing step before the algorithm can be executed (see [Battista et al., 1999]). Dig-CoLa has the advantage that it allows edges against the flow and thereby the graph structure is not modified by the algorithm.

Nevertheless, proclaiming that Dig-CoLa does not require preprocessing is not fully accurate. Dig-CoLa, like all force-directed methods, depends on a distance measurement between the nodes of the graph. Ideally, graph theoretic distances are used for this purpose. Unfortunately computing graph theoretic distances for larger graphs, the so-called "all-pairs-shortest-paths" problem, is a time-intensive operation. In GVS, an implementation of Floyd's algorithm is used, which has cubic running time. Thereby, the computation of the graph theoretic distances will take significantly more time than the actual Dig-CoLa layout algorithm itself. Note that this computation depends only on the structure of the graph and is thereby the same and constant for all graphs with the same structure. Thus, the results could be saved to a file and loaded faster the next time the graph is drawn using Dig-CoLa.

Another interesting fact of the implementation of the Dig-CoLa and the Sugiyama algorithms in GVS is that the GVS implementation of Dig-CoLa uses many matrix operations, which operate on two-dimensional arrays of practically constant size, whereas the GVS Sugiyama implementation basically processes lists, which grow and shrink dynamically. Dig-CoLa requires many numerical vector calculations whereas Sugiyama requires reordering of elements in lists. This gives the Sugiyama implementation a slight performance advantage over Dig-CoLa for small graphs, because list processing is well-supported by Java whereas matrix operations are not supported at all. Nevertheless, for large graphs, Dig-CoLa produces appealing results after a small number of steps, whereas Sugiyama may not produce appealing results at all. Furthermore, Sugiyama will more likely run into OutOfMemory exceptions, because of the dynamic nature of the lists it uses.

While the Sugiyama layout explicitly tries to minimize edge crossings, Dig-CoLa does not include this feature. An experimental study on the performance of Dig-CoLa showed that this does not necessarily lead to more edge crossings in Dig-CoLa, especially for large graphs [Dwyer and Koren, 2006]. Furthermore, Ware et al. [2002] showed that crossing reduction has been overestimated in the past. Taking this into account, Dig-CoLa seems to be a very attractive alternative for drawing large graphs. Sugiyama-style layouts can become massive in size and thereby incomprehensible for large graphs. In

contrast, Dig-CoLa manages to preserve appealing layouts for large graphs, because of its force-directed pedigree.

5.4 Visualizing Longest Path Layering

The layered graph drawing algorithms described in [Battista et al., 1999] and [Kaufmann and Wagner, 2001] produce layered layouts of directed graphs which show the flow of directed edges as well as their implied hierarchy. Several algorithms for producing these layouts exist, but the main idea was first invented by Sugiyama, Tawaga and Toda in 1981 (see [Battista et al., 1999]), which is why such layouts are informally called Sugiyama-style layouts. These algorithms generally consist of the following four steps:

1. Cycle removal.[5]
2. Layer assignment.
3. Crossing reduction.
4. Horizontal or x-coordinate assignment.[6]

In GVS, the SugiyamaLayoutAlgorithm performs each of these steps in sequence. Layer assignment is done using the Longest Path Layering algorithm as presented in [Battista et al., 1999]. This algorithm places all sinks into the bottom-most layer. Sinks are nodes which have no outgoing edges. For all other nodes the longest path to any of these sinks is calculated. A node is placed into the layer being the longest path above the bottom.

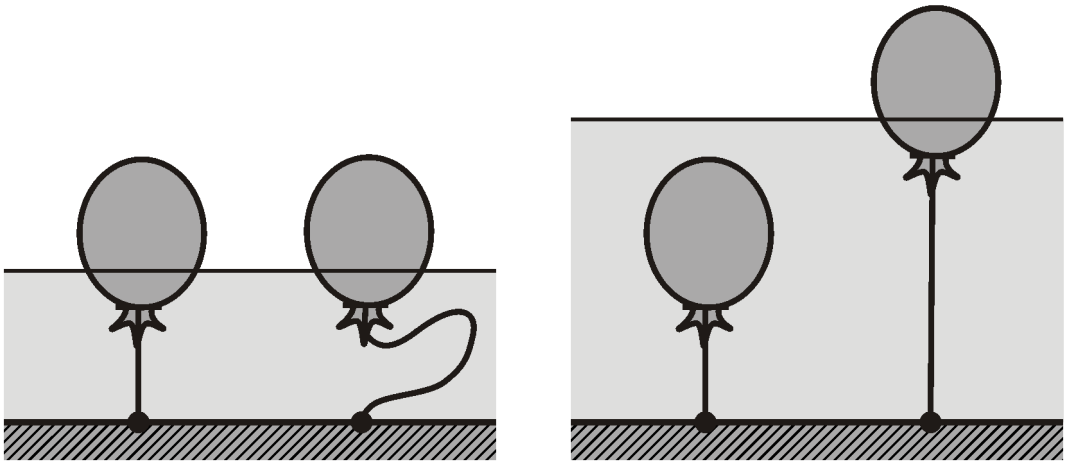
The following metaphor is used to visualize this layering in GVS (see Figure 5.4). All nodes are balloons filled with air floating on the water surface of a basin. Each balloon is tied to the ground of the basin by a chain that has exactly the length of the longest path of the corresponding node. In every layout algorithm step, a certain amount of water is put into the basin and raises the water level. Due to the restraining force of the chains that tie the balloons to the ground, more and more balloons will be forced under water and will remain at the height of their corresponding layer. The algorithm is finished when all balloons are underwater.

This metaphor basically describes what the algorithm does. Every step a new layer is introduced above all existing ones. The sinks of the graph are removed and put into that new, topmost layer. The removal of these sinks will produce new sinks[7] in the graph, which will find their way into the next higher layer in the next step. This is repeated until all nodes have been placed into layers.

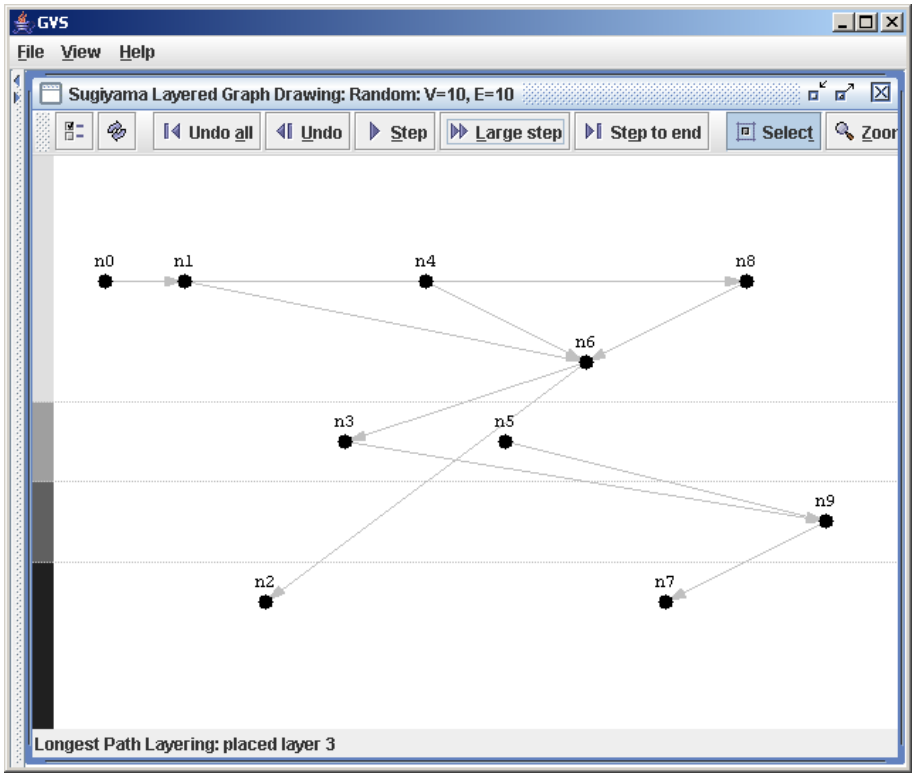
[5] Cycle removal is often considered a preprocessing step, which makes the graph cycle-free, rather than a step of the layout algorithm (compare [Battista et al., 1999]).

[6] Assuming that the layout is drawn vertically, which means that the y-coordinates represent the hierarchy, the x-coordinate assignment step places the nodes along the x-axis. For horizontal layouts this changes correspondingly to vertical or y-coordinate assignment.

[7] It is guaranteed that new sinks are produced by removing all sinks, because the graph is assumed to be cycle-free.



(a) (b)



(c)

Figure 5.4: Illustration of the "floating balloons" metaphor used to explain the Longest Path Layering. Figure (a) shows two balloons chained to the ground of a water filled basin. Since the water level is low enough, both balloons are floating. Figure (b) shows the basin after the water level has risen. Now, the left balloon has been forced underwater because of its short chain, which represents the longest path from the corresponding node to a sink. Figure (c) shows the GVS implementation of this metaphor. The nodes n_0, n_1, n_4 and n_8 are still "floating" while the others have already been forced underwater and thus are assigned to layers.

Chapter 6

Outlook

" To invent, you need a good imagination and a pile of junk. "

[Thomas A. Edison]

This chapter provides an outlook on the field of graph drawing and tries to predict future trends although such predictions are very difficult to make with any degree of accuracy. Furthermore hints for the further development of GVS are given that should help future developers to improve GVS.

6.1 Trends in Graph Drawing

In the past, graph drawing was mostly performed by special purpose algorithms seeking to integrate many aesthetic criteria. The more criteria an algorithm implements, the less general it becomes. The standard works of graph drawing literature [Battista et al., 1999; Kaufmann and Wagner, 2001] contain many such special purpose algorithms for special applications. When these algorithms are generalized, they usually become very complicated and hard to implement. Most of them fail completely when huge graphs have to be drawn.

The force-directed approaches (see Section 5.1) are based on an intuitive real-world model and are thereby easy to understand. Probably the most important reason for the wide propagation of force-directed methods is that they are very easy to implement. Nearly every graph drawing package presented in Chapter 3 contains at least one force-directed algorithm. Another advantage of force-directed methods is that they do not depend on preconditions of the graph. Furthermore, force-directed placement is easy to extend and is scalable to large graphs.

The major disadvantage of force-directed methods is that they are all iterative processes which may converge slowly or, in the worst case, not at all to an optimal solution. For large and dense graphs, the results of force-directed drawings may become unreadable unless special mechanisms are implemented to prevent this. Unfortunately these mechanisms often destroy the simplicity of the force-directed approaches.

Recent developments in the graph drawing community address improving force-directed methods. Especially the AT&T Research Labs group and the Weizmann Institute of Science have published several papers in the recent past on that topic (see Chapter 5). Also the multi-dimensional scaling research group around Matthew Chalmers at University of Glasgow have developed several improvements to force-directed methods (see Chapter 5). Building on force-directed approaches, methods are being developed for drawing very large graphs with tens of thousands of nodes. This branch within the graph drawing community will likely receive the most attention in the near future, because the rapidly growing datasets in computer science make methods necessary for visualizing these large amounts of data. Considering

the long term development in computer science, the field of graph drawing as a whole will definitely expand for the same reason and because there are still many open problems to be solved.

6.2 Further Development of GVS

Although GVS already has many features, it can always be improved. The following section contains a list of thoughts that may lead to further development of GVS:

- The force-directed placement methods and the numerical methods (see Chapter 5) are highly parallelizable. Special hardware could be used to speed these methods up further. This hardware is already available on standard personal computers in the shape of the graphics board [GPGPU, 2006].

  The Graphics Processing Unit (GPU) provides massive computational power for stream processing, which could be exploited for graph drawing. The difficulty with general purpose GPU processing is to find suitable mappings from graph drawing concepts and data structures to the GPU. Solutions exist for performing the conjugate gradient computation on the GPU (see [Bolz et al., 2003]). Since GVS already includes the JOGL renderer, a platform for high performance graphics processing in GVS is already available.

- GraphML is a very powerful and feature-rich graph file format, but it is hardly implemented in full in any graph drawing package. Fortunately, GraphML is structured so that the very basic information about a graph can be easily extracted from a GraphML file regardless of whether the full feature set is supported.

  The GVS GraphMLGenerator uses the SAX XML parser for GraphML files. Only the tags containing graph structure and a few metadata tags are processed. All further XML tags are ignored. These tags contain meta information about the graph and graph elements such as node properties and edge weights. In order to visualize this information in GVS, it would be necessary to extend the GraphMLGenerator. Currently, most GraphML demo files do not use these extended features, so the effect of implementing a full featured GraphMLGenerator would be very small. In addition, a dot parser could be implemented for parsing dot files (see Section 3.2).

- One of the less noticeable features of GVS is the separation of plugins from the rest of the framework. Since the plugins are not different from other classes in terms of the Java virtual machine, they are just an organizational arrangement. Nevertheless, plugin code should be as lightweight as possible. Numerical solvers, general algorithms and reusable classes, such as metadata objects, should be placed in the framework rather than in plugins.

  Of course, developing a new plugin, such as a new Generator or a new Visualization, is eased when components reside within one single package. When student groups develop new plugins, the plugin source code will likely become redundant. From time to time, commonly used routines should be extracted from the plugins and put into the framework. Furthermore, students should be encouraged to use or customize framework methods, rather than reimplement solutions for their special needs.

- One of the first further algorithms to be implemented could be Chalmers [1996] and its improvement Morrison and Chalmers [2004]. Although they are intented for multi-dimensional scaling (see Section 5.1), they would be a welcome addition to GVS' repository of force-directed algorithms.

Regarding the user interface, many small improvements could be done to provide more convenience (for example a search and filter mechanism for node selection). Nevertheless the existing GVS user interface has reached a fairy user friendly state.

Chapter 7

Concluding Remarks

" Veni, vidi, vici.
[I came, I saw, I conquered] "

[Julius Caesar]

Graphs can be used to describe various entity-relationships. Chapter 1 provided several examples of possible applications of graph drawing and information visualization. Information visualization is a field of science that deals with visualizing abstract information to the human user. Graph drawing plays an important role in information visualization because many natural models can be effectively described by graphs. The main goal of automated graph drawing is to present graphs and their features to the human user in an appealing and understandable way.

More precise definitions of graphs, features of graphs and graph drawing were presented in Chapter 2. A graph describes a relation over entities. The entities are called nodes or vertices and the relation is modeled by a list of edges between nodes. Each node or edge of a graph usually has some semantic meaning which is expressed by metadata. Since graphs can have many nodes and edges, creating an aesthetically appealing and understandable drawing is a very difficult task. Various different factors have to be balanced by layout algorithms when placing nodes, routing edges and presenting metadata. Metadata is often represented by labeling the nodes, the edges or both. A large number of possible layouts, feasible techniques, and semantic applications exist. This thesis defined a taxonomy of graph drawing based on [Battista et al., 1999], [Kaufmann and Wagner, 2001], and [Sugiyama, 2002].

There are many applications for graph drawing and also many graph drawing software packages. Chapter 3 introduces a small range of packages that influenced the design of GVS. One of the most widely proven packages is Graphviz. Graphviz is a collection of command line tools that allow other applications to easily use graph drawing. Graphviz is very popular in the Unix/Linux community. Another important package similar to GVS is JUNG. As a platform-independent Java application, JUNG uses Java Swing and Java2D and provides a simple lightweight Java framework for graph drawing.

The main goals and the software design of the Graph Visualization System (GVS) itself were stated in Chapter 4. GVS is mainly intended for teaching and explaining graph drawing concepts and algorithms. Therefore, GVS uses a flexible, object-oriented software design so that new algorithms and new visualizations can be implemented with minimum effort. GVS is a Java 1.5 application using the Swing library for its user interface and Java2D (or alternatively JOGL) for fast rendering. The main components of the framework are the Generators and the Visualizations. Generators provide the means to generate or load graphs into the system. Visualizations perform the graph drawing and present the layouts to the user. The most important features of GVS are that the the actual creation of the layout can be observed step by step and that multiple different Visualizations can be run simultaneously on the same graphs.

Chapter 5 presented selected details of the GVS implementation. One of the most well-known graph drawing techniques is force-directed placement. Nodes in the graph are represented by masses, edges are represented by springs which introduce forces between the masses. Such a system will re-adjust itself until a minimum stress energy level and hence an appealing layout is reached. Due to their simplicity, flexibility, and ability to produce appealing layouts, force-directed methods are often improved for special purposes. One of these is the Directed graph (drawing) with Constrained Layout (Dig-CoLa) algorithm, which combines hierarchical or flow structures of the graph with the intuitivity of force-directed placement.

The field of graph drawing is still expanding. Chapter 6 provided a short outlook into the future of graph drawing as well as suggestions for further development of GVS. Due to their simple applicability and their ability to produce appealing layouts, it is foreseeable that there will be further research in force-directed techniques. The successful application of Dig-CoLa supports this trend. Further improvements to GVS could be to add better support for numerical computations by performing computations on modern graphics hardware or advancing the GraphML Generator to support the full feature set of GraphML as well as implementing ways to display these new forms of information.

7.1 Concluding Remarks

In this thesis, the author presented his research work on graph drawing and graph drawing software as well as the development process of GVS. During the survey on graph drawing software several more packages than described here were investigated. Although every package has a slightly different feature set, it is very interesting how close the packages are to each other. The most noticeable difference between many packages is only the front-end graphical user interface. It is astonishing that so many so similar packages can exist in parallel, so that commercial packages can make a profit. Regarding globalization and the international software market, it could be assumed that there were only a handful of packages, each completely different from each other and each targeting another sector of graph drawing. In the open source community this statement is partially true because of the dominance of Graphviz (see Section 3.2).

The software designs of the packages, at least the Java graph drawing packages, are very similar. The Model View Controller (MVC) design pattern [Buschmann et al., 2004] is already enforced by using Java Swing, while the development of content generators and plugins are eased by Java itself (see Section 4.4.1 and Section 4.4.8). This would again lead to the conclusion that new graph drawing projects would be based on existing solutions rather than implementing software from the scratch. This is true for users of graph drawing applications, like larger software systems that use graph drawing visualizations in the context of a larger framework, but this was hardly observed within the graph drawing developer community.

Due to the investigation of many graph drawing packages, a contradiction between theory and practice in reliability was observed. Reliability not in the sense of software stability or errors, but in the sense of reliability of availability. Graph drawing packages live and die with the community that maintains them. Often this community is composed only of the creators of the original project and once the project has ended, these people move on, leaving the software to languish on some source code server.

Another aspect of this missing reliability is the complete lack of appropriate developer documentation. In many packages hardly any documentation is available. This further makes it difficult to read the source code of a package. If the effort of learning a framework is higher than the effort of implementing the required functionality from scratch, then it is more feasible to do the latter. Of course, using a larger framework would add the benefit of scalability to the application, but, for the reasons presented above, it is also likely that the whole framework may sink into oblivion.

A long needed standardization of graph drawing would solve this problem. One first step into this direction is the development of GraphML. The next step would be to standardize the output of layout

algorithms and to exactly define an interface between graph drawing and output rendering. This would allow separate development of graph drawing algorithms and graph (output) browsers or editors. Then, one overall, full-featured graph browser could be used instead of every single package developing its own.

In terms of GVS, the GraphML standardization and it being based on the well-known XML technology already saved much work. The code for assembling the GraphML file in GVS is only approximately twenty lines long. In contrast, the dot parser used in JMFGraph (see Section 4.2.2) contains about one thousand lines of code. By participating in the GraphML standard and by emphasizing user interaction and the user interface, GVS has become a favorable solution for teaching graph drawing and explaining graph drawing algorithms.

Appendix A

GVS User Guide

GVS is a graph drawing package whose user interface is designed for simplicity. GVS applies the Java Usability Guidelines [Sun, 2001a,b] and most of the user interface components are self-explanatory. This guide provides information about how to install GVS and gives an overview of the user interface.

A.1 Installation

GVS is a Java 1.5 application, so the Java Runtime Environment 5.0[1] is required for execution. By default, GVS uses the Java2D renderer, which is included in the Java core, so no explicit installation is required.

Currently, the JOGL renderer is also implemented for GVS, although it is deactivated by default so as not to have dependencies on OpenGL. In future, there might be visualizations that explicitly require JOGL; it will be necessary to install the JOGL binaries[2] for these visualizations. At the time of writing, installing JOGL is not necessary, so the steps 3 to 5 of the installation procedure described below are not necessary.

There are official installation manuals for both, Java [Sun, 2006a] and JOGL [JOGL, 2006b]. The following provides a step by step list of the actions to be performed to install GVS:

1. Download the Java Runtime Environment 5.0 appropriate for your system from the Java Sun web page[3]. The following link gives the direct download location:

   `http://java.sun.com/j2se/1.5.0/download.jsp`

2. Run the Java installer. The Java installer will automatically set up the environment for executing Java applications.

3. Download the JOGL precompiled binaries (`jogl.jar`) as well as the JOGL natives appropriate for your system (`jogl-natives-*.jar`) from the Java Community JOGL web page[4]. The following link gives the direct download location:

   `https://jogl.dev.java.net/servlets/ProjectDocumentList`

4. Add the JOGL binaries (`jogl.jar`) to your `CLASSPATH` environment variable.

[1] `http://java.sun.com/j2se/1.5.0/index.jsp`
[2] `https://jogl.dev.java.net`
[3] `http://java.sun.com`
[4] `https://jogl.dev.java.net`

5. The JOGL natives archive contains the operating system dependent libraries connecting JOGL to OpenGL. Extract the contents of this archive and make sure that it can be reached by the application. This can be ensured by adding the natives' directory to the `PATH` environment variable or by copying the natives to an already public place, like `jre1.5.0_05/bin`. Of course there has to be an underlying OpenGL driver installed on the system, which the JOGL natives can access.

6. Extract the GVS package archive to some target directory of your choice.

Now the system should be suitable for running GVS. Make sure that the Java environment variables are set and that the Java executables are in your `PATH`. For more details on the Java installation see [Sun, 2006a].

A.2 Startup

GVS ships as a full-featured Java application. All binaries are contained within the GVS main Java archive `GVS.jar`. Use the following command to launch GVS from the command line:

```
java -enableassertions -jar GVS.jar
```

There is also a `GVS.bat` batch file that will issue this command. The batch file provides additional parameters to the Java Virtual Machine that define the memory usage for GVS. It is important to provide the `-enableassertions` option, which may be abbreviated by `-ea`. This option is required to enable the Java assertion mechanism used by GVS.

A.3 GVS Main Window

If everything has been installed properly as described in Section A.1 and GVS has been started as described in the previous section, the GVS main window should become visible. Figure A.1 shows the GVS main window. The three main components of the GVS main window are:

- The menu bar.
- The graph list.
- The visualization desktop.

The GVS menu bar contains the main commands that can be issued in GVS. The "File" menu is responsible for loading and removing graphs as well as for exiting GVS. The "View" menu allows a visualization from a selected graph to be launched and global options and preferences to be specified. The "Help" menu provides information about GVS.

The graph list contains a list of currently loaded graphs. These are graphs available for launching a visualization. Graphs may be loaded or removed using the "File" menu. New visualizations may be launched either by using the "View" menu or by using the context menu of the graph list. See the next section.

After visualizations have been launched from a particular loaded graph, they become visible on the visualization desktop. The visualization desktop acts as a mini desktop environment that can host several visualizations. See Section A.3.2 and Section A.4.

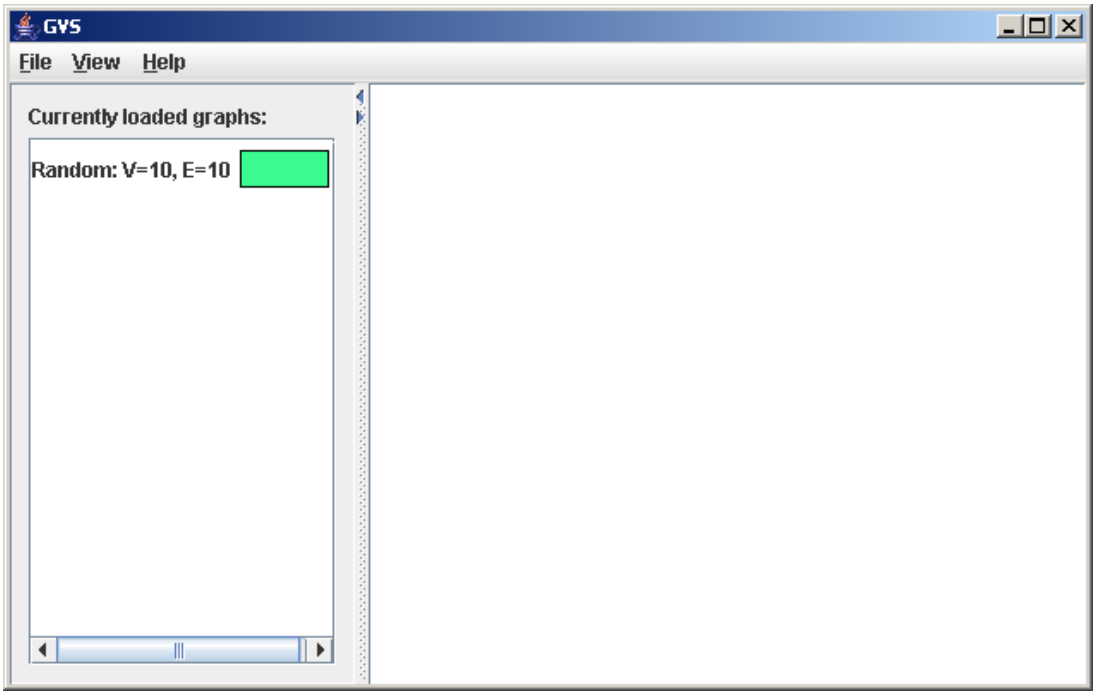


Figure A.1: The GVS main window. Note the three main components of the GVS main window: the menu bar, the graph list ("Currently loaded graphs") and the visualization desktop.

A.3.1 Loading a Graph

When GVS is started, the graph list (see Figure A.1) already contains one default graph. More graphs can be loaded using the "File" menu. "File - Load Graph" provides a list of available Generators that allow new graphs to be loaded into the system. After a graph has been loaded successfully, a random color, the graph color, is associated with the graph and the graph is put into the graph list. The graph color helps distinguish between graphs. Currently, the following three Generators are available for loading a graph:

- The "Random" Generator generates a random graph with certain properties.
- The "Directory Tree" Generator creates a tree from the directory structure below a certain directory of the local file system.
- The "GraphML" Generator loads a graph from a GraphML file. GraphML is an XML-based file format for describing graphs (see [Brandes et al., 2006]).

After choosing a particular Generator, the Generator will request necessary input from the user to perform the generation of the graph. For example, the GraphMLGenerator will of course request the user to select the GraphML file to be loaded. When the generator has finished the creation of the graph, it will be added to the list of loaded graphs in the GVS main window. If an error occurs during graph generation, nothing will be added to the graph list and corresponding information will be displayed.

A graph can also be removed from the graph list to free the resources attached to it. This can be performed by selecting the graph in the graph list and either choosing "Remove graph" from the "File" menu or by using the corresponding action in the graph's context menu. A graph can only be removed if it is not used by any visualizations currently running.

A.3.2 Starting a Visualization

Before a visualization can be started, a graph in the graph list has to be selected to determine which graph should be visualized. There are three ways to start a visualization from a selected graph:

- The "View" menu of the menu bar provides a list of available visualizations. This list is grouped into classes of visualization. Currently these are the "Force-directed", the "Layered" (Sugiyama) and the "Dig-CoLa" approaches.
- The context menu of the selected graph in the graph list provides exactly the same menu as the "View" menu.
- A double click on a graph in the graph list starts the current visualization for that graph. The current visualization is the visualization that was last used. When GVS is started, the current visualization is simply the first visualization available.

After the visualization has been launched successfully, it becomes visible on the visualization desktop. See Figure A.3 for an illustration. Note that a visualization window cannot be moved out of the visualization desktop. A visualization may also refuse its own startup. This will be the case if the selected graph does not fulfill the preconditions of the layout algorithm the visualization uses. In that case, the visualization will display an error message and no window becomes visible in the visualization desktop.

It is possible to start the same visualization of the same global graph multiple times. Each launch will create a separate, independent visualization. Of course, different visualizations can be started from the same global graph, as well as the same visualization can be started from different graphs. Furthermore it is possible, though inefficient in terms of memory usage, to load the same graph several times.

A.3.3 Progress Monitoring

Before a visualization is started, several data structures have to be initialized. The visualization window initially displays the final layout of the graph. Computing the final layout of a graph may take some time, depending on the graph's size and the chosen layout algorithm. Until the layout is fully calculated, a progress monitoring window is shown instead of the final visualization's window.

Figure A.2 shows this progress monitoring window. The user is shown the same information as in the statusbar of the visualization's window (see Section A.4). The difference is that the progress monitoring window does not perform any rendering. The progress monitoring window provides the user with the following three choices:

- The user can wait until the operation is finished. After the operation has finished, the monitoring window disappears and the visualization window is shown.
- The user can safely stop the operation using the "Stop" button. Safely stopping an operation means stopping the operation so that usable results are produced. In the case of launching a visualization, this means that the current large step is finished, but no further large steps are scheduled automatically. Note that stopping safely does not mean that the operation is stopped immediately. It just means that the operation is stopped at the next possible, safe point.
- The user can completely abort the operation using the "Abort" button. Aborting the operation immediately stops the operation, removes all data structures, and frees all resources consumed by the operation. In contrast to a safe stop, no results will be produced. For example, when the launch of a visualization is aborted, no visualization will be shown at all.

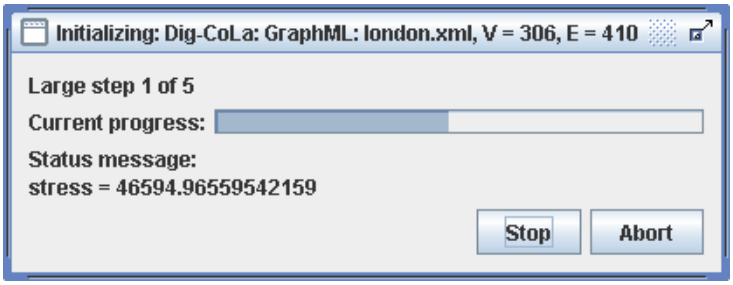


Figure A.2: The GVS progress monitoring window. The window's caption shows the operation that is under way. Here the initialization of the Dig-CoLa visualization of the `london.xml` GraphML graph, which represents the London tube map, is being monitored.

The same progress monitoring window is also used to monitor progress when loading or creating a graph (see Section A.3.1). In particular, creating a large, connected, random graph is a time intensive operation.

A.4 Visualizations

A visualization, or more precisely a GenericVisualization object, will present itself as a visualization window on the visualization desktop of the GVS main window. Figure A.3 illustrates this. The window title contains the name of the visualization as well as the name of the graph it is displaying. The user interface of a visualization window consists of the following three components:

- The visualization toolbar provides tools to execute visualization specific commands.
- The drawing area shows the actual drawing as produced by the visualization.
- The statusbar shows layout algorithm dependent information about the current drawing. If an operation may take more than a few seconds, a progress bar is displayed to inform the user about the progress of the operation.

The "Preferences" button opens a dialog which allows preferences and options for the visualization to be specified. These options mostly control the appearance of the graph in the drawing area, for example, they influence the Drawers. The "Refresh" toolbar button updates the drawing area. This update forces the layout's image to be redrawn to the screen. Note that a redraw operation has neither influence on the graph layout itself nor on the layout algorithm. The toolbar buttons which actually influence the layout are "Undo all", "Undo", "Step", "Large Step" and "Step to end", which are described in the following section. The "Select", "Zoom" and "Pan" toolbar buttons allow the selection of the user interaction mode, which is discussed in Section A.4.2.

A.4.1 Stepping

In contrast to other graph drawing packages, GVS generates the graph drawing in small steps. By selecting the "Step" button in the visualization toolbar, the user gives the command to compute the next layout step. After the layout algorithm has computed the next layout, the current layout is smoothly animated into the new layout (unless of course animation is deactivated). Both operations, computation of the layout and animation, may take some time. Therefore, a progress bar is displayed in the visualization's statusbar to show the progress of the operation.

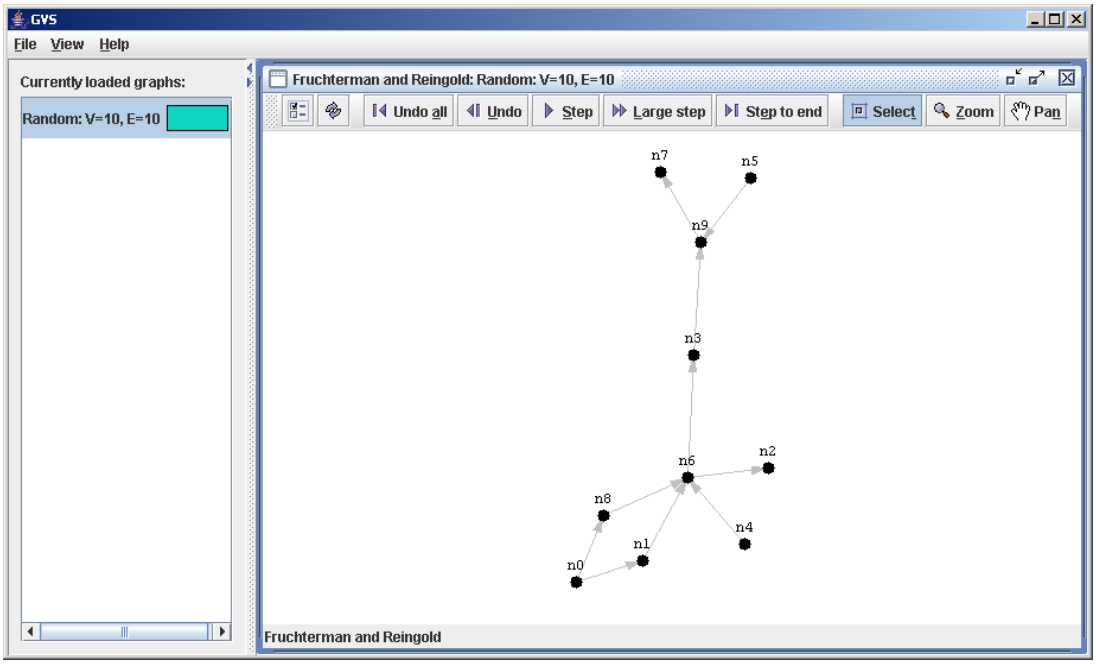


Figure A.3: The GVS Fruchterman and Reingold [1991] visualization launched from the default graph. Note the visualization toolbar at the top, the drawing area, which shows the layout of the graph, and the statusbar at the bottom of the visualization window.

A single step is very small. For iterative layout algorithms it is often necessary to execute tens or hundreds of steps to observe interesting layout changes. For this reason, the "Large step" toolbar button was introduced. A large step command simply tells the layout algorithm to calculate something that actually changes the layout a lot. The size or quantity of a "Step" or a "Large step" is determined by the layout algorithm itself. For example, the Fruchterman and Reingold [1991] layout algorithm defines a large step to be 100 small steps. The exact meaning of "Step" and "Large step" can be found in the tool tip text provided for the corresponding button. The "Step to end" button performs a series of large steps until the layout is finished.[5] Using this functionality it is possible to jump to the final layout.

The "Undo" toolbar button undoes the last "Step" or "Large step" command completely. Internally, a snapshot of the whole visualization is made before a step or large step is executed. An undo command discards the current state of the graph and the layout algorithm and reloads the last such snapshot. Several undo commands in sequence will thereby undo the last steps or large steps. The "Undo all" toolbar button performs undo operations until the initial layout is reached.

The buttons are disabled if the corresponding commands are not available. "Undo" and "Undo all" are only available if a previous version of the graph exists and hence are disabled for the initial layout. "Step", "Large step" and "Step to end" will be disabled when the layout algorithm has fully finished its operations. Note that some layout algorithms, such as iterative stress minimization, may never be finished, although the changes of the layout between two successive steps will become unnoticeably small.

A.4.2 Interaction

The "Select", "Zoom" and "Pan" toolbar buttons define how the user can interact with the drawing area. Only one interaction mode may be active at one time. The following three modes of interaction exist:

[5] For algorithms that never finish, a certain number of large steps is performed.

- Selection Mode allows the user to pick single or multiple nodes and highlight them.
- Zoom Mode allows the user to zoom in or out of the drawing area.
- Pan Mode allows the user to pan and reposition the graph in the drawing area.

Note that if the system is equipped with mouse wheel functionality, the mouse wheel can be used to zoom in or out in every interaction mode. Furthermore, note that none of these modes influences the graph layout or the layout algorithm itself.

Selection Mode

Selection Mode is activated by the "Select" toolbar button. In Selection Mode, the user can highlight a node by clicking on it. The highlight disappears when the user clicks on an empty space of the drawing area. By pressing the "Shift" key while selecting a node, the node will be selected without changing the selection status of other nodes. This allows the selection of multiple nodes. By pressing the "Ctrl" key while selecting a node, the selection status of this node will toggle. This means that the node will be selected if it was not selected before and vice versa. Drawing a selection rectangle around some nodes applies the actions described above on the included group of nodes. There is also a context menu in Selection Mode which allows specific actions to be performed. The context menu provides additional commands for selecting all nodes of the graph, as well as selecting none.

Note that selection is actually performed on the underlying global graph. This means that if a node is highlighted in one visualization, it will be highlighted in any other visualization based on the same graph. In this way, selection can effectively be used to demonstrate the different placements of the same node by different layout algorithms.

Zoom Mode

By default, "Auto Zoom Fit" is activated. This means that the graph drawing is zoomed so that it completely fits onto the drawing area. By entering Zoom Mode using the "Zoom" button, the user can zoom in and out of the drawing. Clicking on the drawing area in Zoom Mode will zoom in, drawing a rectangle around some area will zoom in that specific area. Zooming out (as well as zooming in) is provided via the Zoom Mode context menu. If a mouse wheel is present, it can be used to conveniently zoom in and out as well. The context menu also allows zooming the graph so that it fits into the drawing area.

"Auto Zoom Fit" is deactivated automatically, when the user issues another zooming command. When "Auto Zoom Fit" is deactivated and a layout step is issued, the new layout might not fit into the drawing area or might move out of sight. Either panning (see next section) or the "Zoom Fit" command in the context menu can be used to refocus the drawing. The "Auto Zoom Fit" functionality can be reactivated by checking the corresponding option in the context menu.

Pan Mode

Pan Mode, which can be activated by the "Pan" toolbar button, allows the graph to be repositioned on the drawing area. The Pan Mode context menu provides a command to center the view on the origin. Note that panning the drawing disables "Auto Zoom Fit" as well.

A.5 Usage Example

The scenario is to load a graph from a GraphML file and visualize it by the Fruchterman and Reingold [1991] layout algorithm. The following step by step list describes a typical procedure:

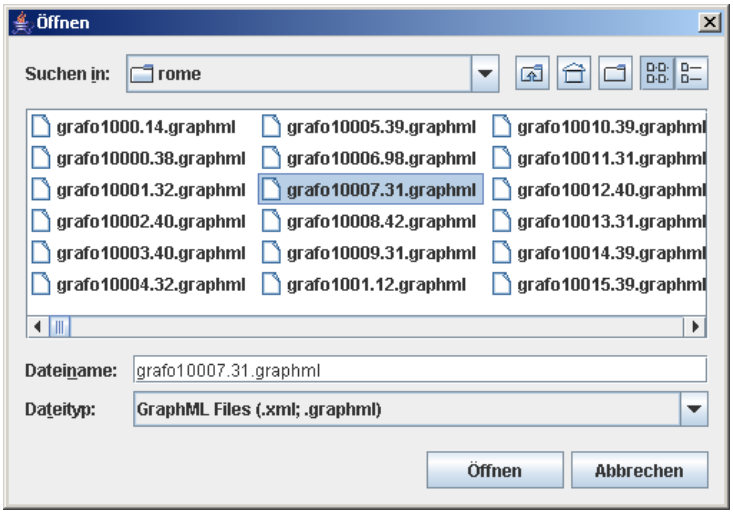


Figure A.4: The file chooser for opening a GraphML file.

1. When GVS is started, the user interface presents itself as shown in Figure A.1.
2. To load the graph from a GraphML file, "File - Load Graph - GraphML File" is used. This opens the dialog illustrated by Figure A.4, which allows the selected GraphML file to be loaded.
3. Select the GraphML file. Here the file `grafo10007.31.graphml` from the Rome graph collection[6] was chosen. After the graph has been loaded successfully, it becomes visible in the "Currently loaded graphs" graph list.
4. To launch the Fruchterman and Reingold [1991] visualization from that graph, select it in the graph list and use "View - Force-Directed - Fruchterman and Reingold". This visualization will bring up a window on the visualization desktop, as shown in Figure A.5. The layout shown when the visualization launches is the final layout. This final layout is produced by performing a certain number of large steps on the graph.
5. To go back to the initial layout before any graph drawing has been performed use "Undo all". Figure A.5 shows a typical GVS initial layout, where all nodes are placed on the unit grid according to their (random) order in the node list.
6. Use the "Step" button to perform a small step. The layout will be smoothly animated from Figure A.6(a) to Figure A.6(b).
7. Further steps and large steps further refine the layout. See Figures A.6(c) to A.6(e).
8. Continue performing steps and large steps until the final layout is reached. The "Step to end" button may be used to reach the final layout as well. See Figure A.6(f).

At any time, preferences or the interaction mode may be changed to further investigate the graph. Furthermore, the undo operations may be used to step back to a previous layout. Note that undo operations reset the graph to the previous state and all information about the current state is lost. This means that a step following the undo will require new computations.

[6] `http://www.graphdrawing.org/download/rome-graphml.tgz`

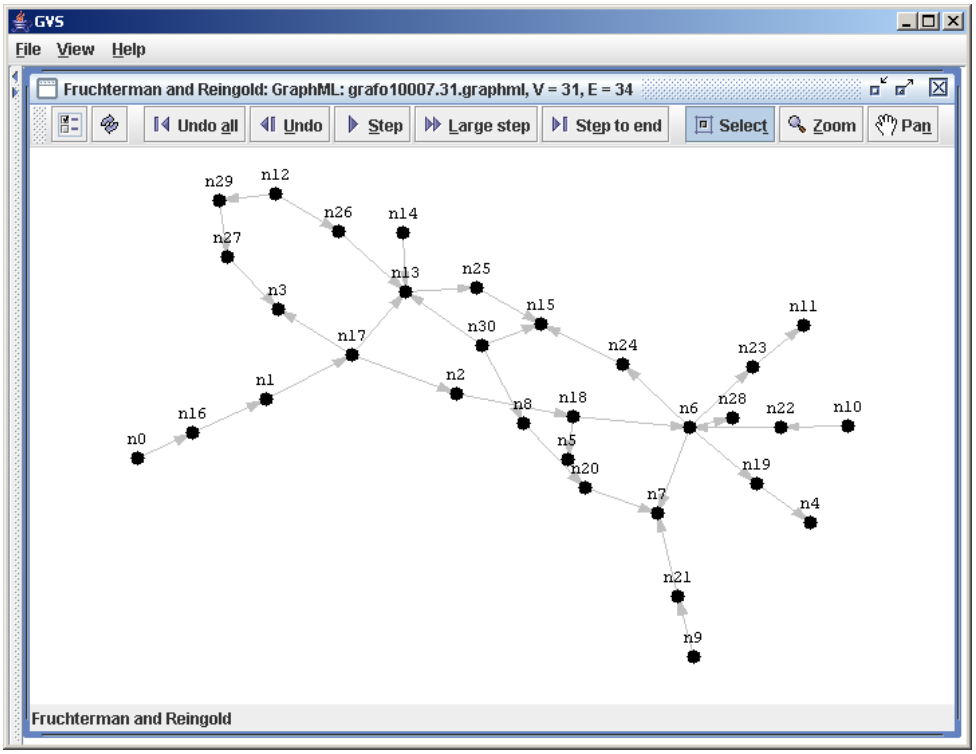


Figure A.5: The final layout presented after the Fruchterman and Reingold [1991] visualization of the `grafo10007.31.graphml` graph has been launched. This shows the graph after a number of large layout steps. Note that the graph list has been minimized to give more room to the visualization.

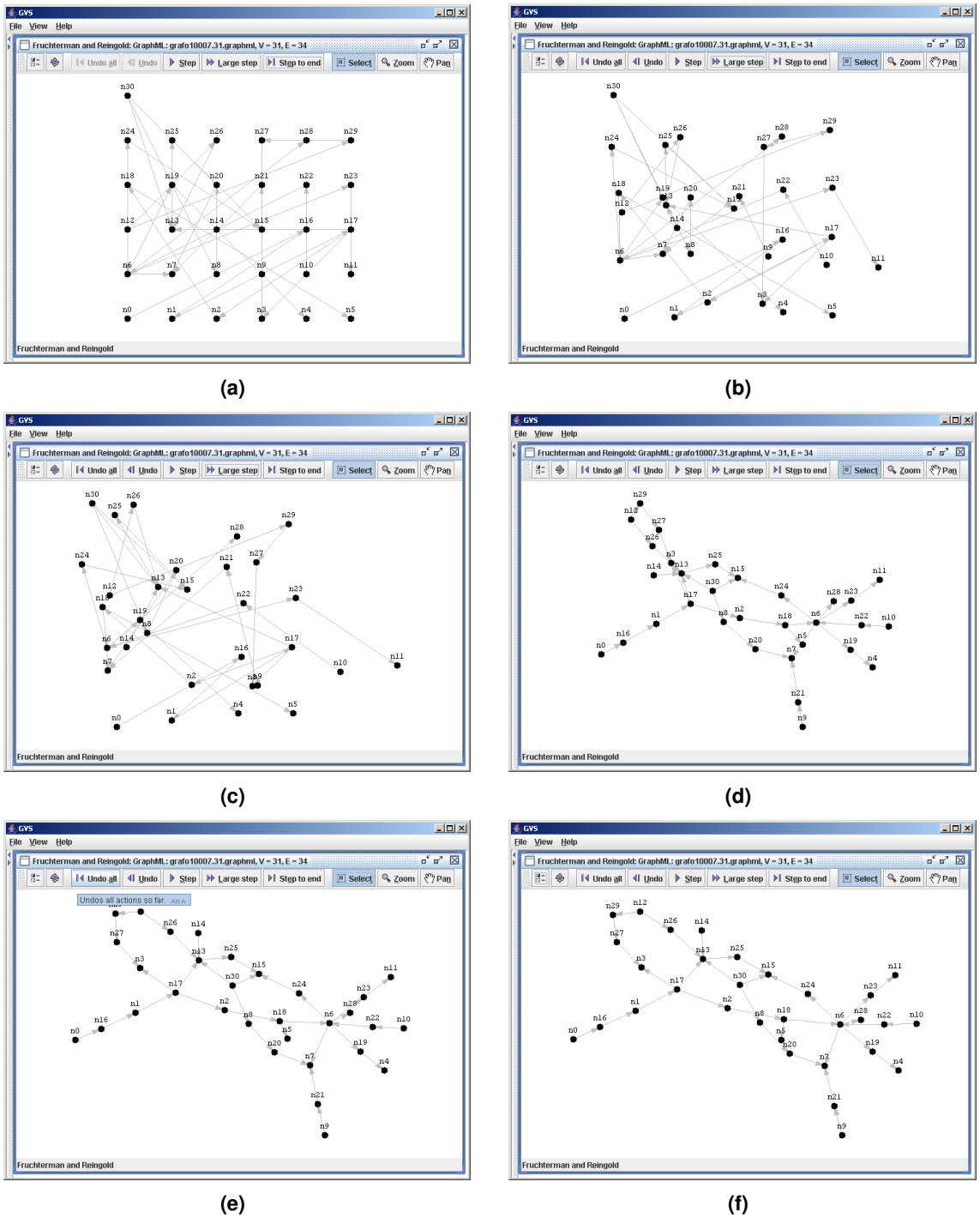


Figure A.6: The layout steps of the Fruchterman and Reingold [1991] layout algorithm for the `grafo10007.31.graphml` graph of the Rome collection. Figure (a) shows the initial layout. Figures (b) and (c) show the layouts after one and two (small) steps. Figures (d) and (e) show the layouts after one and two further large steps. Figure (f) shows the final layout. Note that the graph list (usually on the left) has been minimized to give more room to the visualization.

Appendix B

GVS Developer Guide

Since GVS is a full-featured Java application, nearly all Java development tools will be suitable for developing GVS components. The GVS framework was developed using the Java Eclipse IDE. It is highly recommended to use this very common Java development tool for further development of GVS. This appendix describes how to install Eclipse and make it ready to develop for GVS. Furthermore, the development process of a small visualization application will be demonstrated.

During the implementation of GVS, the Java Swing Tutorial [Walrath et al., 2004] proved especially helpful. All user interface components were implemented accordingly to the Java Swing Tutorial. Where applicable, the Java Usability Guidelines [Sun, 2001a,b] were followed. For general Java development, [Campione et al., 2000] and [Krüger, 2004] were used. Comprehensive documentation of Eclipse can be found in [Eclipse, 2006]. For development using JOGL and OpenGL, [JOGL, 2006b] and [Shreiner et al., 2003] are recommended.

B.1 Installation

In order to develop with GVS, the Java SDK 1.5 is necessary. All development of GVS so far was done using the Eclipse IDE. The Eclipse IDE is required to take advantage of the style checker and code formatter used for developing GVS. For convenience in development it is recommended to obtain the source code and the JavaDoc documentation of both the Java 1.5 SDK and the JOGL API and to configure Eclipse to use both. The following steps make the system ready for developing using GVS:

1. Download the Java SDK 1.5 appropriate for your system from the Java Sun web page[1]. The following link gives the direct download location:

   `http://java.sun.com/j2se/1.5.0/download.jsp`

2. Run the Java SDK installer. If the Java runtime environment has already been installed on the system, the proprietary Java runtime environment of the Java SDK may conflict with the already installed one. Therefore, make sure that the environment variables were set correctly by the installer. It is recommended to use the runtime environment provided by the Java SDK during the development process, because it is more suitable for debugging purposes than the standalone Java runtime environment, which contains no debugging information at all. Also, if JOGL is required, make sure that it is properly installed for the used runtime environment.

3. Download the Eclipse IDE from the Eclipse web page[2]. The following link gives the direct download location:

[1] `http://java.sun.com`
[2] `http://www.eclipse.org/`

`http://www.eclipse.org/downloads`

4. Unpack the Eclipse archive to some place. Although Eclipse is a Java application itself, it ships with its own, standalone runtime environment. Thus, no explicit installer is needed to set up Eclipse.

5. Extract the GVS development workspace archive to some target directory of your choice.

6. Start Eclipse. When started, Eclipse will ask for the workspace directory. This is the `workspace` folder extracted from the GVS development workspace archive.

Now Eclipse should be ready to develop with. Further information about Java development using Eclipse can be found in [Eclipse, 2006].

B.2 Developing a Visualization

This section demonstrates how to develop a Visualization for GVS using the example of the Fruchterman and Reingold [1991] layout algorithm. The development process for a Generator is very similar to the one for a Visualization, so many things described here apply to Generators as well. Implementing a new Visualization, and within it, a new LayoutAlgorithm is considered to be the task that is most likely to be done by future student groups. Detailed information about the classes and interfaces used in GVS can be found in [Prinz, 2006].

Every project must start with the project plan, and every project plan starts with the definitions of the goals that must be reached. In case of the Fruchterman and Reingold [1991] visualization, the following steps are necessary:

1. Create an implementation of the VisualizationPlugin interface, which will be named FruchtermanReingoldVisualizationPlugin.

2. Create an implementation of the LayoutAlgorithm interface, which will be named FruchtermanReingoldLayoutAlgorithm.

3. Register the FruchtermanReingoldVisualizationPlugin class to the plugins recognized by GVS.

4. Implement Fruchterman and Reingold in the context of the FruchtermanReingoldLayoutAlgorithm class.

5. Implement the FruchtermanReingoldVisualizationPlugin to launch a GenericVisualization using an instance of the FruchtermanReingoldLayoutAlgorithm.

6. Document and test the implementation.

B.2.1 Creating the Classes

In GVS, all visualizations should reside in the `edu.iicm.gvs.visualizations` package. Although it is possible to place new visualizations somewhere else, doing so would be inconsistent. There are several sub-packages, which correspond to the visualization groups into which the actual visualizations are categorized. A new visualization should be placed either in one of the existing categories or a new category should be created. In our example, create a new package with the name `fruchtermanreingold` in the `forcedirected` package. All new class files should therefore be created in the package:

`edu.iicm.gvs.visualizations.forcedirected.fruchtermanreingold`

Creating the class files themselves is an easy task using Eclipse. Eclipse allows the specification of interfaces the class files should implement (see [Eclipse, 2006]). The naming scheme is GVS suggests appending the name of the implemented interface after the name of the class. Thus, following two classes need to be created in the package:

- FruchtermanReingoldVisualizationPlugin implementing VisualizationPlugin and
- FruchtermanReingoldLayoutAlgorithm implementing LayoutAlgorithm.

B.2.2 Registering the Plugin

An implementation of a VisualizationPlugin is registered with GVS by entering the full Java class name of the plugin in the GVS file `plugins.txt`. The GVS `plugins.txt` can be found in the `edu.iicm.gvs` package. This file contains the description of all available plugins as an XML snippet. Since GVS does not provide any XML document type definition or schema for this file, it is given the extension `.txt` rather than `.xml`. Visualizations are furthermore categorized into visualization groups. This grouping can be observed in the "View" menu. Although it is not necessary to register a certain VisualizationPlugin under the group corresponding to the group package its files are saved in, doing so is highly recommended. Listing B.1 shows an excerpt from the `plugins.txt` file with the corresponding entry for the FruchtermanReingoldVisualizationPlugin in line 5.

```
1      <visualization>edu.iicm.gvs.visualizations.layered.sugiyama.
           SugiyamaVisualizationPlugin</visualization>
2    </visgrp>
3    <visgrp name="Force-Directed" description="Force-directed (
         spring-based) layout algorithms.">
4      <visualization>edu.iicm.gvs.visualizations.forcedirected.eades
           .EadesVisualizationPlugin</visualization>
5      <visualization>edu.iicm.gvs.visualizations.forcedirected.
           fruchtermanreingold.FruchtermanReingoldVisualizationPlugin<
           /visualization>
6    </visgrp>
7  </visualizations>
```

Listing B.1: Excerpt from the `plugins.txt` file showing how visualizations are registered.

B.2.3 Implementing the LayoutAlgorithm

Implementing the actual LayoutAlgorithm is the creative part of the visualization, because the LayoutAlgorithm is what distinguishes the visualizations from each other. The LayoutAlgorithm interface provides the following methods which have to be implemented.

- The `getTitle` method is used to identify the LayoutAlgorithm to the framework. A short, descriptive string is required here.
- The `isFinished` method is used by the framework to query the status of the layout process. When the LayoutAlgorithm is finished, which means that it requires no more steps, it must return true.

- The `initializeLayout` and `finishLayout` methods are called by GVS to initialize or finish the layout. Within initialize, the LayoutAlgorithm should place the graph's nodes in initial positions and allocate all necessary data structures. Within finish, it should free these data structures.
- The `step` and `stepLarge` methods are called by GVS to tell the LayoutAlgorithm that it should calculate a step or a large step.

The implementation of the `getTitle` method is straightforward. The Fruchterman and Reingold layout algorithm computes the layout iteratively by minimizing the energy of a spring model. This process is continuous and thereby never finished.[3] In such a case, `isFinished` is implemented so that it always returns false. Even though `finishLayout` is never called, this method is implemented so that it would free the algorithm-specific data structures.

The `initializeLayout` method places the nodes on a rectangular grid for initialization. It also allocates a data structure necessary for the subsequent algorithm steps. In order to support the "Undo" operation, a LayoutAlgorithm implementation class has to be completely stateless. All state information has to be bundled and included as metadata with the graph or the graph's items. When a step or large step is executed, this information has to be read from the graph and possibly be modified.[4]

The action of calculating the layout is performed in the `step` and `stepLarge` methods. A Fruchterman and Reingold large step executes 100 small steps in sequence: the implementation of `stepLarge` is straightforward. Since LayoutAlgorithms are stateless, all information they need for execution is passed as a parameter, namely the AlgorithmStatusMonitor. The AlgorithmStatusMonitor provides various information and with it the VirtualGraph upon which the LayoutAlgorithm is to work.

The AlgorithmStatusMonitor additionally allows the LayoutAlgorithm to specify user interface related information. For example, a LayoutAlgorithm should always set the status message when it has finished computing a step or a large step. This status message is displayed in the visualization's statusbar. Furthermore, the texts which appear as tool tips for the "Step" and "Large step" toolbar buttons are defined via the AlgorithmStatusMonitor.[5] See [Prinz, 2006] for details.

B.2.4 Implementing the VisualizationPlugin

Compared to the implementation of the LayoutAlgorithm, the implementation of the VisualizationPlugin is very simple. There are only four methods of interest in the VisualizationPlugin interface:

- The `getName` and `getDescription` methods return strings which describe the plugin.
- The `getAuthors` method provides a list of strings with the names of the authors who implemented the plugin.
- The `createVisualization` method creates the Visualization instance for the provided VirtualGraph and returns it so that the framework can make it visible.

The `getName`, `getDescription` and `getAuthors` methods simply return static information and are easy to implement. The `createVisualization` method is not much more difficult because of the use of the GenericVisualization class. The constructor of the GenericVisualization class is simply equipped with the VirtualGraph and an instance of the FruchtermanReingoldLayoutAlgorithm.

[3] Although the progress of cooling down the system is usually stopped when the system temperature falls below a certain value (see [Fruchterman and Reingold, 1991]).

[4] Note that writing back the corresponding metadata is not necessary because the MetadataObject is already referenced by the graph. See [Prinz, 2006] for further details.

[5] The status message and the "Step" and "Large step" tool tip texts are actually simply further metadata objects stored in the VirtualGraph. The AlgorithmStatusMonitor handles them for convenience.

This automatically creates the user interface and the environment. Now the Fruchterman and Reingold visualization is ready for use in GVS.

Note that the FruchtermanReingoldLayoutAlgorithm and the FruchtermanReingoldVisualizationPlugin could have been joined into a single class. Since the LayoutAlgorithm and the VisualizationPlugin are just interfaces, a single class could simply implement both of them. However, the structure of the plugin is far clearer, if these two components are separated. Furthermore, reusing a LayoutAlgorithm is more intuitive, if it is a single, separate component.

It is also possible not to use the GenericVisualization. Indeed the whole LayoutAlgorithm mechanism is only required so that the GenericVisualization can be used. The VisualizationPlugin may provide its own complete implementation of the Visualization interface. However, doing so would require the whole user interface to be implemented, which is very much effort. Only visualizations using a totally different drawing concept might consider doing so. Before implementing a Visualization from scratch, it is advisable to consider if the task could not be performed by implementing a special drawer component instead. See the [Prinz, 2006] for more information about the extensive GenericVisualization class.

Abbreviations

AGD	Algorithms for Graph Drawing
API	Application Programming Interface
AT&T	American Telephone and Telegraph Corporation
CFD	Computational Fluid Dynamics
CORBA	Common Object Request Broker Architecture
Dig-CoLa	Directed graph (drawing) with Constrained Layout
GEOMI	GEOmetry for Maximum Insight
GIF	Graphics Interchange Format
GraphML	Graph Markup Language
GUI	Graphical User Interface
GVS	Graph Visualization System
HCI	Human-Computer Interaction
HVS	Hierarchical Visualization System
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
JMFGraph	Java Modular Framework for Graph Drawing
JOGL	Java bindings for OpenGL
JPEG	Joint Photographic Experts Group
JUNG	Java Universal Network/Graph Framework
LEDA	Library of Efficient Data types and Algorithms
MVC	Model View Controller
OpenGL	Open Graphics Library
PAC	Presentation Abstraction Controller
PNG	Portable Network Graphics
POV	Persistence of Vision
QPOC	Quadratic Programming with Orthogonal Constraints
RMI	Remote Method Invocation
SAX	Simple API for XML
SDK	Software Development Kit
SDL	Simple Directmedia Layer
SVG	Scalable Vector Graphics
Tcl	Tool Command Language
UML	Unified Modelling Language
VRML	Virtual Reality Modeling Language
WWW	World Wide Web
X3D	eXtensible 3D
XML	eXtensible Markup Language

Bibliography

AGD [2006]. *AGD – A Library of Algorithms for Graph Drawing*. Web Site. http://www.ads.tuwien.ac.at/AGD. (Cited on page 28.)

Adel Ahmed, Tim Dwyer, Michael Forster, Xiaoyan Fu, Joshua Ho, Seok-Hee Hong, Dirk Koschützki, Colin Murray, Nikola S. Nikolov, Ronnie Taib, Alexandre Tarassov, and Kai Xu [2005]. *GEOMI: GEOmetry for Maximum Insight*. In Patrick Healy and Nikola S. Nikolov (Editors), *Proceedings of the 13th International Symposium on Graph Drawing (GD 2005), Limerick, Ireland, September 12-14, 2005*, volume 3843 of *Lecture Notes in Computer Science*, pages 468 – 479. Springer. ISBN 3540314253. ISSN 0302-9743. doi:10.1007/11618058_42. (Cited on pages 34 and 36.)

aiSee [2006]. *aiSee (I see!)*. Web Site. http://www.aisee.com. (Cited on page 40.)

Keith Andrews [1998]. *Visualizing Rich, Structured Hypermedia*. IEEE Computer Graphics and Applications, 18(4), pages 40–42. ISSN 0272-1716. doi:10.1109/38.689661. (Cited on page 5.)

Keith Andrews [2002]. *Visualising Information Structures – Aspects of Information Visualisation*. Professorial thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. ftp://ftp.iicm.edu/pub/keith/habil/visinfo.pdf. (Cited on pages 4, 5 and 46.)

Keith Andrews [2006a]. *Human–Computer Interaction*. Lecture notes, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. http://courses.iicm.edu/hci/hci.pdf. (Cited on pages 1 and 2.)

Keith Andrews [2006b]. *Information Visualisation*. Lecture notes, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. http://courses.iicm.edu/ivis/ivis.pdf. (Cited on pages 4 and 40.)

Keith Andrews, Wolfgang Kienreich, Vedran Sabol, Jutta Becker, Georg Droschl, Frank Kappe, Michael Granitzer, Peter Auer, and Klaus Tochtermann [2002]. *The InfoSky Visual Explorer: Exploiting Hierarchical Structure and Document Similarities*. Information Visualization, 1(3–4), pages 166–181. ISSN 1473-8716. doi:10.1057/palgrave.ivs.9500023. (Cited on pages 2 and 3.)

David Auber [2002]. *Tulip Graph Library Documentation*. Source code documentation. http://dept-info.labri.fr/~auber/projects/tulip. (Cited on page 36.)

David Auber [2003]. *Tulip – A Huge Graph Visualization Framework*, pages 105–126. In Jünger and Mutzel [2003]. The papers in this book were originally published in [Mutzel et al., 2002]. (Cited on page 36.)

Vladimir Batagelj and Andrej Mrvar [2003]. *Pajek – Analysis and Visualization of Large Networks*, pages 77–104. In Jünger and Mutzel [2003]. The papers in this book were originally published in [Mutzel et al., 2002]. (Cited on pages 31, 32 and 33.)

Vladimir Batagelj and Andrej Mrvar [2006]. *Pajek – Program for Analysis and Visualization of Large Networks – Reference Manual version 1.12*. University of Ljubljana. http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/pajekman.pdf. (Cited on pages 31, 32 and 33.)

Giuseppe Di Battista (Editor) [1997]. *Proceedings of the 5th International Symposium on Graph Drawing (GD '97), Rome, Italy, September 18-20, 1997*, volume 1353 of *Lecture Notes in Computer Science*. Springer. ISBN 3540639381. ISSN 0302-9743. (Not cited.)

Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis [1999]. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, Simon & Schuster, Upper Saddle River, New Jersey 07458. ISBN 0133016153. (Cited on pages 9, 10, 12, 13, 15, 18, 19, 20, 23, 24, 29, 47, 49, 64, 66, 67, 69 and 71.)

Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder [2003]. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. In *Proceedings of ACM SIGGRAPH 2003*, volume 22 of *ACM Transactions on Graphics (TOG)*, pages 917–924. ACM Press. ISSN 0730-0301. doi:10.1145/882262.882364. http://www.multires.caltech.edu/pubs/GPUSim.pdf. (Cited on page 70.)

Ingwer Borg and Patrick J. F. Groenen [2005]. *Modern Multidimensional Scaling – Theory and Applications*. Springer Series in Statistics. Springer, second edition. ISBN 0387251502. (Cited on page 5.)

Katy Börner and Yuezheng Zhou [2001]. *A Software Repository for Education and Research in Information Visualization*. In E. Elanissi, F. Khosrowshahi, M. Sarfraz, and A. Ursyn (Editors), *Proceedings of the Fifth International Conference on Information Visualisation (IV01), London, England, July 25-27, 2001*, pages 257–262. IEEE Press. ISBN 0769511953. doi:10.1109/IV.2001.942068. http://iv.slis.indiana.edu/ref. (Cited on page 41.)

Ulrik Brandes, Markus Eiglsperger, and Jürgen Lerner [2006]. *GraphML Primer*. Online document. http://graphml.graphdrawing.org/primer/graphml-primer.html. See also http://graphml.graphdrawing.org. (Cited on pages 14, 50 and 77.)

Christoph Buchheim, Michael Jünger, and Sebastian Leipert [2002]. *Improving Walker's Algorithm to Run in Linear Time*. In M.T. Goodrich and S.G. Kobourov (Editors), *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002), Irvine, CA, USA, August 26-28, 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 344–353. Springer. ISBN 3540001581. ISSN 0302-9743. (Cited on pages 5 and 20.)

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal [2004]. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons Ltd, reprinted edition. ISBN 0471958697. (Cited on pages 26, 34, 45, 46, 48, 53, 54, 56 and 72.)

Mary Campione, Kathy Walrath, and Alison Huml [2000]. *The Java™ Tutorial: A Short Course on the Basics*. Addison-Wesley Professional, third edition. ISBN 0201703939. http://java.sun.com/docs/books/tutorial. (Cited on page 85.)

Liran Carmel, David Harel, and Yehuda Koren [2002]. *Drawing Directed Graphs Using One-Dimensional Optimization*. In M.T. Goodrich and S.G. Kobourov (Editors), *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002), Irvine, CA, USA, August 26-28, 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 193–206. Springer. ISBN 3540001581. ISSN 0302-9743. (Cited on page 63.)

Liran Carmel, David Harel, and Yehuda Koren [2004]. *Combining Hierarchy and Energy for Drawing Directed Graphs*. IEEE Transactions on Visualization and Computer Graphics, 10(1), pages 46–57. ISSN 1077-2626. doi:10.1109/TVCG.2004.1260757. (Cited on pages 62, 63 and 64.)

Matthew Chalmers [1996]. *A Linear Iteration Time Layout Algorithm for Visualising High–Dimensional Data*. In *Proceedings of the 7th IEEE Conference on Visualization (VIS'96), San Francisco, California, USA, October 27 - November 1, 1996*, pages 127–132. IEEE Press. ISSN 1070-2385. doi:10.1109/VISUAL.1996.567787. http://www.dcs.gla.ac.uk/~matthew/papers/vis96.pdf. (Cited on pages 59, 61 and 70.)

Chaomei Chen [2004]. *Information Visualization - Beyond the Horizon*. Springer, second edition. ISBN 1852337893. (Cited on pages 4 and 5.)

Jonathan D. Cohen [1997]. *Drawing Graphs to Convey Proximity: An Incremental Arrangement Method*. ACM Transactionson Computer–Human Interaction, 4(3), pages 197–229. ISSN 1073-0516. doi:10.1145/264645.264657. (Cited on page 60.)

Colt [2006]. *Colt*. Web Site. http://dsd.lbl.gov/~hoschek/colt. (Cited on page 29.)

Trevor F. Cox and Michael A. A. Cox [2000]. *Multidimensional Scaling*. Chapman & Hall/CRC, second edition. ISBN 1584880945. (Cited on page 5.)

Ron Davidson and David Harel [1996]. *Drawing Graphs Nicely Using Simulated Annealing*. ACM Transactions on Graphics, 15(4), pages 301–331. ISSN 0730-0301. doi:10.1145/234535.234538. (Cited on page 61.)

Wouter de Nooy, Andrej Mrvar, and Vladimir Batagelj [2005]. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press. ISBN 0521602629. http://vlado.fmf.uni-lj.si/pub/networks/book. (Cited on page 31.)

S. Diehl (Editor) [2002]. *Proceedings of the International Seminar on Software Visualization, Dagstuhl Castle, Germany, May 20-25, 2001*, volume 2269 of *Lecture Notes in Computer Science*. Springer. ISBN 3540433236. ISSN 0302-9743. (Cited on pages 98 and 101.)

John DiMarco (Editor) [2004]. *Computer Graphics and Multimedia: Applications, Problems and Solutions*. Idea Group Publishing. ISBN 1591401968. (Cited on page 102.)

Doxygen [2006]. *Doxygen*. Web Site. www.doxygen.org. (Cited on page 28.)

Peter F. Drucker [1993]. *Managing for Results*. HarperBusiness (HarperPerennial), reprinted edition. ISBN 0887306144. (Cited on page 43.)

David Duke [2001]. *Modular Techniques in Information Visualization*. In Eades and Pattison [2001], pages 11–18. http://crpit.com/confpapers/CRPITV9Duke.pdf. (Cited on page 4.)

Tim Dwyer [2001]. *Three Dimensional UML Using Force Directed Layout*. In Eades and Pattison [2001], pages 77–85. http://crpit.com/confpapers/CRPITV9Dwyer.pdf. (Cited on page 33.)

Tim Dwyer [2004]. *Extending WilmaScope – A developer's guide to extending the WilmaScope 3D graph visualisation system*. WilmaScope Development Team. http://www.wilmascope.org/doc/ExtendingWilmaScope.html. (Cited on pages 33 and 34.)

Tim Dwyer and Peter Eckersley [2003]. *WilmaScope – A 3D Graph Visualization System*, pages 55–76. In Jünger and Mutzel [2003]. The papers in this book were originally published in [Mutzel et al., 2002]. (Cited on page 33.)

Tim Dwyer and Yehuda Koren [2005]. *Dig-CoLa: Directed Graph Layout through Constrained Energy Minimization*. In John Stasko and Matt Ward (Editors), *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2005), Minneapolis, MN, USA, October 23-25, 2005*, pages 9–9. IEEE Press. ISBN 0780387793. ISSN 1522-404X. doi:10.1109/INFOVIS.2005.10. To appear in 2006. (Cited on page 64.)

Tim Dwyer and Yehuda Koren [2006]. *Dig-CoLa: Directed Graph Layout through Constrained Energy Minimization*. IEEE Transactions on Visualization and Computer Graphics. To appear. (Cited on pages 6, 11, 15, 18, 49, 59, 63, 64 and 66.)

Tim Dwyer, Yehuda Koren, and Kim Marriott [2005a]. *Stress Majorization with Orthogonal Ordering Constraints*. In Patrick Healy and Nikola S. Nikolov (Editors), *Proceedings of the 13th International Symposium on Graph Drawing (GD 2005), Limerick, Ireland, September 12-14, 2005*, volume 3843 of *Lecture Notes in Computer Science*, pages 141–152. Springer. ISBN 3540314253. ISSN 0302-9743. doi:10.1007/11618058_14. (Cited on pages 64, 65 and 66.)

Tim Dwyer, Kim Marriott, and Peter J. Stuckey [2005b]. *Fast Node Overlap Removal*. In Patrick Healy and Nikola S. Nikolov (Editors), *Proceedings of the 13th International Symposium on Graph Drawing (GD 2005), Limerick, Ireland, September 12-14, 2005*, volume 3843 of *Lecture Notes in Computer Science*, pages 153–164. Springer. ISBN 3540314253. ISSN 0302-9743. doi:10.1007/11618058_15. A correction of this work is available in [Dwyer et al., 2006]. (Cited on page 96.)

Tim Dwyer, Kim Marriott, and Peter J. Stuckey [2005c]. *Fast Node Overlap Removal in Graph Layout Adjustment*. Technical Report 2005/173, Monash University School of Computer Science and Software Engineering, Australia. See also [Dwyer et al., 2005b]. A correction of this work is available in [Dwyer et al., 2006]. (Cited on pages 12 and 18.)

Tim Dwyer, Kim Marriott, and Peter J. Stuckey [2006]. *Fast Node Overlap Removal — Addendum*. Technical Report 2005/173, Monash University School of Computer Science and Software Engineering, Australia. Will be published in GD2006. (Cited on page 96.)

Breanne Dyck, Sebastian Hanlon, and Stephen Wismath [2004a]. *GLuskap User's Manual – Three-Dimensional Graph Drawing Software*. University of Lethbridge, Lethbridge, Alberta, Canada. `http://www.cs.uleth.ca/~vpak/gluskap/docs/manual.pdf`. (Cited on page 38.)

Breanne Dyck, Jill Joevenazzo, Elspeth Nickle, Jon Wilsdon, and Stephen Wismath [2004b]. *GLuskap: Visualization and Manipulation of Graph Drawings in 3-Dimensions*. In Giuseppe Liotta (Editor), *Proceedings of the 11th International Symposium on Graph Drawing (GD 2003), Perugia, Italy, September 21-24, 2003*, volume 2912 of *Lecture Notes in Computer Science*, pages 496–497. Springer. ISBN 3540208313. ISSN 0302-9743. `http://www.cs.uleth.ca/~vpak/gluskap/docs/software.ps`. (Cited on pages 37 and 38.)

Peter Eades [1984]. *A heuristic for graph drawing*. Congressus Nutnerantiunt, 42, pages 149–160. `http://www.cs.usyd.edu.au/~peter`. (Cited on pages 15, 19, 49, 59 and 60.)

Peter Eades and Tim Pattison (Editors) [2001]. *Information Visualisation 2001 (invis.au 2001)*, volume 9 of *Conferences in Research and Practice in Information Technology (CRPIT)*. Australian Computer Society (ACS), Sydney, Australia. ISBN 0909925879. ISSN 1445-1336. `http://crpit.com/Vol9.html`. (Cited on page 95.)

Eclipse [2006]. *Eclipse 3.1 Documentation*. `http://www.eclipse.org/documentation`. (Cited on pages 85, 86 and 87.)

E. Elanissi, F. Khosrowshahi, M. Sarfraz, and A. Ursyn (Editors) [2001]. *Proceedings of the Fifth International Conference on Information Visualisation (IV01), London, England, July 25-27, 2001*. IEEE Press. ISBN 0769511953. (Not cited.)

John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull [2003]. *Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools*, pages 127–148. In Jünger and Mutzel [2003]. The papers in this book were originally published in [Mutzel et al., 2002]. (Cited on pages 26 and 27.)

Jean-Daniel Fekete [2003]. *The InfoVis Toolkit*. Research Report RR-4818, Institut National de Recherche en Informatique et en Automatique (INRIA), Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France. ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4818.pdf. (Cited on page 40.)

Jean-Daniel Fekete [2004]. *The InfoVis Toolkit*. In Matt Ward and Tamara Munzner (Editors), *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2004), Austin, Texas, USA, October 10-12, 2004*, pages 167–174. IEEE Press. ISBN 0780387793. ISSN 1522-404X. doi:10.1109/INFVIS.2004.64. http://www.lri.fr/~fekete/ps/ivtk-04.pdf. (Cited on page 40.)

Jean-Daniel Fekete [2006]. *The InfoVis Toolkit*. Web Site. http://ivtk.sourceforge.net. (Cited on page 40.)

Thomas M. J. Fruchterman and Edward M. Reingold [1991]. *Graph Drawing by Force-directed Placement*. Software–Practice and Experience, 21(11), pages 1129–1164. ISSN 0038-0644. (Cited on pages 15, 19, 31, 33, 59, 60, 80, 81, 82, 83, 84, 86 and 88.)

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides [1997]. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, reprinted edition. ISBN 0201633612. (Cited on pages 45, 53, 56 and 57.)

Emden Gansner, Eleftherios Koutsofios, and Stephen North [2002]. *Drawing graphs with dot (dot User's Manual)*. AT&T Research Labs (Graphviz). http://www.graphviz.org/Documentation/dotguide.pdf. (Cited on pages 26 and 27.)

Emden R. Gansner [2004]. *Drawing graphs with Graphviz*. AT&T Research Labs (Graphviz). http://www.graphviz.org/doc/libguide/libguide.pdf. (Cited on pages 26 and 28.)

Emden R. Gansner and Stephen C. North [1998]. *Improved Force-Directed Layouts*. In Sue H. Whitesides (Editor), *Proceedings of the 6th International Symposium on Graph Drawing (GD '98), Montréal, Canada, August 13-15, 1998*, volume 1547 of *Lecture Notes in Computer Science*, pages 364–374. Springer. ISBN 3540654739. ISSN 0302-9743. (Cited on pages 6 and 59.)

Emden R. Gansner, Yehuda Koren, and Stephen North [2005]. *Graph Drawing by Stress Majorization*. In János Pach (Editor), *Proceedings of the 12th International Symposium on Graph Drawing (GD 2004), New York, NY, USA, September 29 - October 2, 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 239–250. Springer. ISBN 3540245286. ISSN 0302-9743. (Cited on pages 59, 60, 61 and 62.)

GEOMI [2005]. *GEOMI JavaDoc*. Source code documentation. http://dept-info.labri.fr/~auber/projects/tulip. (Cited on page 36.)

GEOMI [2006]. *GEOMI (Geometry for Maximum Insight)*. Web Site. http://www.cs.usyd.edu.au/~visual/valacon/geomi. (Cited on page 34.)

GLuskap [2006]. *GLuskap*. Web Site. http://www.cs.uleth.ca/~vpak/gluskap. (Cited on page 37.)

M.T. Goodrich and S.G. Kobourov (Editors) [2002]. *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002), Irvine, CA, USA, August 26-28, 2002*, volume 2528 of *Lecture Notes in Computer Science*. Springer. ISBN 3540001581. ISSN 0302-9743. (Not cited.)

Google [2006]. *Google Directory for Graph Drawing Software*. http://www.google.com/Top/Science/Math/Combinatorics/Software/Graph_Drawing. (Cited on page 25.)

GPGPU [2006]. *General-Purpose Computation Using Graphics Hardware*. Web Site. http://www.gpgpu.org. (Cited on page 70.)

graphdrawing.org [2006]. *Graph Drawing*. Web Site. http://graphdrawing.org. (Cited on page 25.)

Graphviz [2006a]. *The DOT Language*. Web Site. http://www.graphviz.org/doc/info/lang.html. (Cited on page 26.)

Graphviz [2006b]. *Graphviz – Graph Visualization Software*. Web Site. http://www.graphviz.org. (Cited on page 26.)

Graphviz [2006c]. *Graphviz Resources*. http://www.graphviz.org/Resources.php. (Cited on page 28.)

S. Grivet, David Auber, Jean-Philippe Domenger, and G. Melancon [2004]. *Bubble Tree Drawing Algorithm*. In *International Conference on Computer Vision and Graphics (ICCVG 2004)*, pages 633–641. Springer. ISBN 1402041780. http://dept-info.labri.u-bordeaux.fr/~auber/documents/publi/grivetICCVG2004.pdf. (Cited on pages 36 and 37.)

Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Sebastian Leipert, and Petra Mutzel [2002]. *Graph Drawing Algorithm Engineering with AGD*. In Diehl [2002], pages 307–323. (Cited on page 29.)

David Harel and Yehuda Koren [2001]. *A Fast Multi-Scale Method for Drawing Large Graphs*. In J. Marks (Editor), *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000), Colonial Williamsburg, VA, USA, September 20-23, 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 183–196. Springer. ISBN 3540415548. ISSN 0302-9743. (Cited on page 61.)

David Harel and Yehuda Koren [2002a]. *A Fast Multi-Scale Method for Drawing Large Graphs*. Journal of Graph Algorithms and Applications, 6(3), pages 179–202. ISSN 1526-1719. http://www.cs.brown.edu/publications/jgaa. (Cited on page 61.)

David Harel and Yehuda Koren [2002b]. *Drawing Graphs with Non–Uniform Vertices*. Proceedings of Working Conference on Advanced Visual Interfaces (AVI'02) , ACM Press, pp. 157–166. (Cited on pages 12 and 18.)

David Harel and Yehuda Koren [2002c]. *Graph Drawing by High-Dimensional Embedding*. In M.T. Goodrich and S.G. Kobourov (Editors), *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002), Irvine, CA, USA, August 26-28, 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 207–219. Springer. ISBN 3540001581. ISSN 0302-9743. (Cited on pages 11, 19 and 61.)

Patrick Healy and Nikola S. Nikolov (Editors) [2005]. *Proceedings of the 13th International Symposium on Graph Drawing (GD 2005), Limerick, Ireland, September 12-14, 2005*, volume 3843 of *Lecture Notes in Computer Science*. Springer. ISBN 3540314253. ISSN 0302-9743. doi:10.1007/11618058. (Cited on page 25.)

Jeffrey Heer, Stuart K. Card, and James A. Landay [2005]. *Prefuse: A toolkit for Interactive Information Visualization*. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI 2005), Portland, Oregon, USA, April 2-7, 2005*, pages 421–430. ACM Press, New York, NY, USA. ISBN 1581139985. doi:10.1145/1054972.1055031. (Cited on page 41.)

Jeffrey Michael Heer [2004]. *Prefuse – A Software Framework for Interactive Information Visualization*. Master's thesis, Computer Science Division University of California, Berkeley. (Cited on page 41.)

IVC [2006]. *InfoVis Cyberinfrastructure*. Web Site. http://iv.slis.indiana.edu. (Cited on page 41.)

Java2D [2006]. *Java2D API*. Web Site. http://java.sun.com/products/java-media/2D. (Cited on page 49.)

Java3D [2006a]. *Java3D – Project home*. Web Site. https://java3d.dev.java.net. (Cited on page 49.)

Java3D [2006b]. *Java3D Developer's Documentation*. Online documentation. http://wiki.java.net/bin/view/Javadesktop/Java3D. (Cited on pages 42, 49 and 50.)

JGraph [2006]. *JGraph – Java Graph Visualization and Layout*. Web Site. http://www.jgraph.com. (Cited on page 38.)

JOGL [2006a]. *JOGL – Java bindings for OpenGL*. Web Site. https://jogl.dev.java.net. (Cited on page 49.)

JOGL [2006b]. *Jogl – User's Guide*. JOGL Developer Community. https://jogl.dev.java.net/nonav/source/browse/*checkout*/jogl/doc/userguide/index.html. (Cited on pages 42, 49, 75 and 85.)

Brian Johnson and Ben Shneiderman [1991]. *Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures*. In Gregory M. Nielson and Larry Rosenblum (Editors), *Proceedings of the IEEE Conference on Visualization (Vis '91), San Diego, CA, USA, October 22-25, 1991*, pages 284–291. IEEE Press. ISBN 0818622458. doi:10.1109/VISUAL.1991.175815. (Cited on page 5.)

JUNG [2005]. *JUNG API Documentation (JavaDoc) version 1.7.2*. JUNG Development Team. http://jung.sourceforge.net/doc/api. (Cited on page 31.)

JUNG [2006a]. *JUNG – Java Universal Network/Graph Framework*. Web Site. http://jung.sourceforge.net. (Cited on page 29.)

JUNG [2006b]. *Projects Using JUNG*. Web Site. http://jung.sourceforge.net/pmwiki/index.php/Main/ProjectsUsingJUNG. (Cited on page 29.)

Michael Jünger and Petra Mutzel (Editors) [2003]. *Graph Drawing Software*. Mathematics and Visualization. Springer. ISBN 3540008810. The papers in this book were originally published in [Mutzel et al., 2002]. (Cited on pages 25, 93, 95, 96, 99 and 104.)

Michael Jünger, Gunnar W. Klau, Petra Mutzel, and René Weiskirchner [2003]. *AGD – A Library of Algorithms for Graph Drawing*, pages 149–172. In Jünger and Mutzel [2003]. The papers in this book were originally published in [Mutzel et al., 2002]. (Cited on pages 28, 29 and 30.)

T. Kamada and S. Kawai [1989]. *An Algorithm for Drawing General Undirected Graphs*. Information Processing Letters, 31(1), pages 7–15. ISSN 0020-0190. doi:10.1016/0020-0190(89)90102-6. (Cited on pages 15, 19, 27, 31, 32, 33, 49, 59, 60 and 61.)

Michael Kaufmann and Dorothea Wagner (Editors) [2001]. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer. ISBN 3540420622. ISSN 0302-9743. (Cited on pages 9, 10, 12, 13, 15, 19, 20, 21, 23, 49, 67, 69, 71 and 101.)

Wolfgang Kienreich, Vedran Sabol, Michael Granitzer, Frank Kappe, and Keith Andrews [2003]. *InfoSky: A System for Visual Exploration of Very Large, Hierarchically Structured Knowledge Spaces*. In *Proceedings of FGWM 2003 (SIG KM 2003)*. Karlsruhe, Germany. http://km.aifb.uni-karlsruhe.de/ws/LLWA/fgwm. (Cited on page 2.)

Know-Center [2006]. *InfoSky Demo*. Austria's Competence Center for Knowledge Management (Know–Center), InfoSky Demo Download Page. http://en.know-center.at/forschung/wissenserschliessung/downloads_demos/infosky_demo. (Cited on pages 2 and 3.)

Yehuda Koren [2003]. *On Spectral Graph Drawing*. In Warnow and Zhu [2003], pages 496–508. (Cited on pages 19 and 61.)

Yehuda Koren [2005]. *Drawing Graphs by Eigenvectors: Theory and Practice*. Computers & Mathematics with Applications, 49(11–12), pages 1867–1888. doi:10.1016/j.camwa.2004.08.015. (Cited on pages 6, 15, 19 and 61.)

Yehuda Koren and David Harel [2004]. *Axis-by-Axis Stress Minimization*. In Giuseppe Liotta (Editor), *Proceedings of the 11th International Symposium on Graph Drawing (GD 2003), Perugia, Italy, September 21-24, 2003*, volume 2912 of *Lecture Notes in Computer Science*, pages 450–459. Springer. ISBN 3540208313. ISSN 0302-9743. (Cited on page 63.)

Yehuda Koren and David Harel [2005]. *One Dimensional Layout Optimization, with Applications to Graph Drawing by Axis Separation*. Computational Geometry: Theory and Applications, 32(2), pages 115–138. doi:10.1016/j.comgeo.2005.03.003. (Cited on pages 11, 19 and 61.)

Eleftherios Koutsofios and Stephen C. North [1996]. *Editing graphs with dotty*. AT&T Research Labs (Graphviz). http://www.graphviz.org/Documentation/dottyguide.pdf. (Cited on page 28.)

Guido Krüger [2004]. *Handbuch der Java-Programmierung*. Addison-Wesley, fourth edition. ISBN 3827322014. http://www.javabuch.de. This book is written in German language. (Cited on page 85.)

Eriola Kruja, Joe Marks, Ann Blair, and Richard Waters [2001]. *A Short Note on the History of Graph Drawing*. Technical Report TR2001-49, Mitsubishi Electric Research Laboratories, 201 Broadway, Cambridge, Massachusetts 02139, USA. http://www.merl.com/reports/docs/TR2001-49.pdf. Also published in [Kruja et al., 2002]. (Cited on pages 4 and 100.)

Eriola Kruja, Joe Marks, Ann Blair, and Richard Waters [2002]. *A Short Note on the History of Graph Drawing*. In Petra Mutzel, Michael Jünger, and Sebastian Leipert (Editors), *Proceedings of the 9th International Symposium on Graph Drawing (GD 2001), Vienna, Austria, September 23-26, 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 272–286. Springer. ISBN 3540433090. ISSN 0302-9743. Also available in [Kruja et al., 2001]. (Cited on page 100.)

John Lamping and Ramana Rao [1994]. *Laying Out and Visualizing Large Trees Using a Hyperbolic Space*. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST '94), Marina del Rey, California, USA, November 2-4, 1994*, pages 13–14. ACM Press. ISBN 0897916573. doi:10.1145/192426.192430. (Cited on pages 5 and 38.)

LEDA [2006]. *LEDA*. Web Site. http://www.algorithmic-solutions.com/enleda.htm. (Cited on page 29.)

Giuseppe Liotta (Editor) [2004]. *Proceedings of the 11th International Symposium on Graph Drawing (GD 2003), Perugia, Italy, September 21-24, 2003*, volume 2912 of *Lecture Notes in Computer Science*. Springer. ISBN 3540208313. ISSN 0302-9743. doi:10.1007/b94919. (Not cited.)

David G. Luenberger [2003]. *Linear and Nonlinear Programming*. Springer, second edition. ISBN 1402075936. (Cited on page 62.)

Jock D. Mackinlay, George G. Robertson, and Stuart K. Card [1991]. *The Perspective Wall: Detail and Context Smoothly Integrated*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching through Technology (CHI '91), New Orleans, Louisiana, USA, April 27 - May 2, 1991*, pages 173–179. ACM Press. ISBN 0897913833. doi:10.1145/108844.108870. (Cited on page 5.)

Fredmund Malik [2001]. *Führen, Leisten, Leben – Wirksames Management für eine neue Zeit*. Heyne. ISBN 3453196848. (Cited on page 43.)

J. Marks (Editor) [2001]. *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000), Colonial Williamsburg, VA, USA, September 20-23, 2000*, volume 1984 of *Lecture Notes in Computer Science*. Springer. ISBN 3540415548. ISSN 0302-9743. (Not cited.)

Alistair Morrison and Matthew Chalmers [2004]. *A Pivot-Based Routine for Improved Parent-Finding in Hybrid MDS*. Information Visualization, 3(2), pages 109–122. ISSN 1473-8716. doi:10.1057/palgrave.ivs.9500040. (Cited on pages 59, 61 and 70.)

Alistair Morrison, Greg Ross, and Matthew Chalmers [2003]. *Fast Multidimensional Scaling through Sampling, Springs and Interpolation*. Information Visualization, 2(1), pages 68–77. ISSN 1473-8716. doi:10.1057/palgrave.ivs.9500040. (Cited on pages 59 and 61.)

Tamara Munzner [2000]. *Interactive Visualization of Large Graphs and Networks*. Ph.d. thesis, Department of Computer Science and the Committee on Graduate Studies of Stanford University. `http://graphics.stanford.edu/papers/munzner_thesis`. (Cited on page 38.)

Petra Mutzel and Peter Eades [2002]. *Graphs in Software Visualization*. In Diehl [2002], pages 285–294. (Cited on pages 19 and 29.)

Petra Mutzel, Michael Jünger, and Sebastian Leipert (Editors) [2002]. *Proceedings of the 9th International Symposium on Graph Drawing (GD 2001), Vienna, Austria, September 23-26, 2001*, volume 2265 of *Lecture Notes in Computer Science*. Springer. ISBN 3540433090. ISSN 0302-9743. (Cited on pages 93, 95, 96, 99 and 104.)

Petra Mutzel, Michael Jünger, and Stefan Näher [2003]. *AGD User Manual 1.1.1*. Max-Planck-Institut Saarbrücken, Universität zu Köln, Universität Trier, Technische Universität Wien. `http://www.ads.tuwien.ac.at/AGD/MANUAL`. (Cited on pages 29 and 30.)

Gabriele Neyer [2001]. *Map Labeling with Application to Graph Drawing*. In Kaufmann and Wagner [2001], pages 247–273. (Cited on pages 12 and 18.)

Stephen C. North [2002]. *Drawing graphs with neato (neato User's Manual)*. AT&T Research Labs (Graphviz). `http://www.graphviz.org/Documentation/neatoguide.pdf`. (Cited on pages 26 and 27.)

Alexander Nussbaumer [2005]. *Hierarchy Browsers*. Master's thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. `http://www.iicm.edu/thesis/anussbaumer.pdf`. (Cited on page 4.)

Joshua O'Madadhain and Danyel Fisher [2005]. *JUNG (Java Universal Network/Graph) Framework Manual*. JUNG Development Team. `http://jung.sourceforge.net/doc/manual.html`. (Cited on pages 29 and 31.)

Joshua O'Madadhain, Danyel Fisher, Padhraic Smyth, Scott White, and Yan-Biao Boey [2006]. *Analysis and Visualization of Network Data using JUNG*. Journal of Statistical Software. ISSN 1548-7660. `http://www.jstatsoft.org`. To appear. (Cited on page 29.)

János Pach (Editor) [2005]. *Proceedings of the 12th International Symposium on Graph Drawing (GD 2004), New York, NY, USA, September 29 - October 2, 2004*, volume 3383 of *Lecture Notes in Computer Science*. Springer. ISBN 3540245286. ISSN 0302-9743. doi:10.1007/b105810. (Not cited.)

Pajek [2006]. *Networks/Pajek – Program for Large Network Analysis*. Web Site. http://vlado.fmf.uni-lj.si/pub/networks/pajek. (Cited on page 31.)

Shashikant Penumarthy, Bruce W. Herr, and Katy Börner [2004]. *IVC Software Framework Programmer Manual 0.1*. InfoVis Lab at Indiana University, Bloomington, IN 47405, USA. http://iv.slis.indiana.edu/sw/papers/ivc-framework.pdf. (Cited on pages 41 and 42.)

Prefuse [2006]. *Prefuse – Information Visualization Toolkit*. Web Site. http://prefuse.sourceforge.net. (Cited on page 41.)

Wolfgang Prinz [2005]. *Data Flow Systems*. Seminar paper, Institute for Computer Graphics and Vision (ICG), Graz University of Technology, Austria. (Cited on page 4.)

Wolfgang Prinz [2006]. *GVS JavaDoc Implementation Documentation*. Source code documentation. (Cited on pages 51, 53, 86, 88 and 89.)

Helen C. Purchase [1997]. *Which aesthetic has the greatest effect on human understanding?* In Giuseppe Di Battista (Editor), *Proceedings of the 5th International Symposium on Graph Drawing (GD '97), Rome, Italy, September 18-20, 1997*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer. ISBN 3540639381. ISSN 0302-9743. (Cited on page 16.)

Helen C. Purchase [2004]. *Evaluating Graph Drawing Aesthetics: Defining and Exploring a New Empirical Research Area*, chapter 8, pages 145–178. In DiMarco [2004]. (Cited on page 16.)

Helen C. Purchase, Jo-Anne Allder, and David Carrington [2002]. *Graph Layout Aesthetics in UML Diagrams: User Preferences*. Journal of Graph Algorithms and Applications, 6(3), pages 255–279. ISSN 1526-1719. http://www.cs.brown.edu/publications/jgaa. (Cited on page 16.)

Werner Putz [2005]. *The Hierarchical Visualization System*. Master's thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. http://www.iicm.edu/thesis/wputz.pdf. (Cited on pages 4, 44, 45 and 47.)

Python [2006]. *Python – The Official Python Programming Language Website*. Web Site. http://www.cs.uleth.ca/~vpak/gluskap. (Cited on page 37.)

E.M. Reingold and J.S. Tilford [1981]. *Tidier Drawings of Trees*. IEEE Transactions on Software Engineering, 7(2), pages 223–228. ISSN 0098-5589. (Cited on pages 36 and 37.)

George G. Robertson, Jock D. Mackinlay, and Stuart K. Card [1991]. *Cone Trees: Animated 3D Visualizations of Hierarchical Information*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching through Technology (CHI '91), New Orleans, Louisiana, USA, April 27 - May 2, 1991*, pages 189–194. ACM Press. ISBN 0897913833. doi:10.1145/108844.108883. (Cited on page 5.)

Ben Shneiderman [1996]. *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*. In *Proceedings of the IEEE Symposium on Visual Languages, Boulder, CO, USA, September 3-6, 1996*, pages 336–343. IEEE Press. ISBN 081867508X. ISSN 1049-2615. doi:10.1109/VL.1996.545307. (Cited on page 4.)

Ben Shneiderman and Catherine Plaisant [2004]. *Designing the User Interface: Strategies for Effective Human-Computer Interactio*. Addison Wesley, fourth edition. ISBN 0321197860. (Cited on page 4.)

Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis [2003]. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*. Addison-Wesley Professional, fourth edition. ISBN 0321173481. (Cited on pages 55 and 85.)

Robert Spence [2001]. *Information Visualization*. Addison-Wesley. ISBN 0201596261. (Cited on pages 1, 2, 4 and 5.)

John Stasko and Matt Ward (Editors) [2005]. *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2005), Minneapolis, MN, USA, October 23-25, 2005*. IEEE Press. ISBN 0780387793. ISSN 1522-404X. (Not cited.)

Alexander Stedile [2001]. *JMFGraph – A Modular Framework for Drawing Graphs in Java*. Master's thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. `http://www.iicm.edu/thesis/astedile.pdf`. (Cited on pages 46 and 50.)

Kozo Sugiyama [2002]. *Graph Drawing and Applications for Software and Knowledge Engineers*, volume 11 of *Series on Software Engineering and Knowledge Engineering*. World Scientific, New Jersey, NJ. ISBN 9810248792. (Cited on pages 6, 9, 15, 18, 19, 20, 23, 24, 63, 64 and 71.)

Sun [2001a]. *Java™ Look and Feel Design Guidelines*. Addison-Wesley Professional, second edition. ISBN 0201725886. `http://java.sun.com/products/jlf/ed2/book/index.html`. (Cited on pages 1, 75 and 85.)

Sun [2001b]. *Java™ Look and Feel Design Guidelines: Advanced Topics*. Addison-Wesley Professional. ISBN 0201775824. `http://java.sun.com/products/jlf/at/book/index.html`. (Cited on pages 75 and 85.)

Sun [2006a]. *Installation Notes for JDK 5.0 Microsoft Windows (32-bit)*. Web Site. `http://java.sun.com/j2se/1.5.0/install-windows.html`. (Cited on pages 75 and 76.)

Sun [2006b]. *Java2D™ Technology*. Sun Microsystems. `http://java.sun.com/j2se/1.4.2/docs/guide/2d`. (Cited on pages 42 and 49.)

Bernhard Tatzmann [2004]. *Dynamic Exploration of Large Graphs*. Master's thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. `http://www.iicm.edu/thesis/btatzmann.pdf`. (Cited on pages 21 and 56.)

TomSawyer [2006]. *Tom Sawyer Software*. Web Site. `http://www.tomsawyer.com`. (Cited on page 40.)

Trolltech [2006]. *Trolltech Qt*. Web Site. `http://www.trolltech.com/products/qt`. (Cited on page 36.)

Edward R. Tufte [2001]. *The Visual Display of Quantitative Information*. Graphics Press, second edition. ISBN 0961392142. `http://www.edwardtufte.com/tufte/books_vdqi`. (Cited on pages 5, 6 and 20.)

Tulip [2006]. *Tulip*. Web Site. `http://www.labri.fr/perso/auber/projects/tulip`. (Cited on page 36.)

W3C [2006]. *The Semantic Web Initiative*. World Wide Web Consortium (W3C). Web Site. `http://www.w3.org/2001/sw`. (Cited on page 6.)

Kathy Walrath, Mary Campione, Alison Huml, and Sharon Zakhour [2004]. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley Professional, second edition. ISBN 0201914670. `http://java.sun.com/docs/books/tutorial/uiswing`. (Cited on page 85.)

Walrus [2006]. *Walrus – Graph Visualization Tool*. Web Site. http://www.caida.org/tools/visualization/walrus. (Cited on page 38.)

Matt Ward and Tamara Munzner (Editors) [2004]. *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2004), Austin, Texas, USA, October 10-12, 2004*. IEEE Press. ISBN 0780387793. ISSN 1522-404X. (Not cited.)

Colin Ware [2004]. *Information Visualization - Perception for Design*. Morgan Kaufmann (Elsevier). ISBN 1558608192. (Cited on page 4.)

Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill [2002]. *Cognitive Measurements of Graph Aesthetics*. Information Visualization, 1(2), pages 103–110. ISSN 1473-8716. doi:10.1057/palgrave.ivs.9500013. (Cited on pages 12, 13, 16 and 66.)

Tandy Warnow and Binhai Zhu (Editors) [2003]. *Proceedings of the 9th Annual International Conference on Computing and Combinatorics (COCOON 2003), Big Sky, MT, USA, July 25-28, 2003*, volume 2697 of *Lecture Notes in Computer Science*. Springer. ISBN 3540405348. ISSN 0302-9743. (Cited on page 100.)

Web3D [2006]. *Web3D Consortium – Open Standards for Real-Time 3D Communication*. Web Site. http://www.web3d.org. (Cited on page 49.)

Eric W. Weisstein [2002a]. *Conjugate Gradient Method*. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ConjugateGradientMethod.html. (Cited on page 62.)

Eric W. Weisstein [2002b]. *Method of Steepest Descent*. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/MethodofSteepestDescent.html. (Cited on page 62.)

Sue H. Whitesides (Editor) [1998]. *Proceedings of the 6th International Symposium on Graph Drawing (GD '98), Montréal, Canada, August 13-15, 1998*, volume 1547 of *Lecture Notes in Computer Science*. Springer. ISBN 3540654739. ISSN 0302-9743. (Not cited.)

Roland Wiese, Markus Eiglsperger, and Michael Kaufmann [2003]. *yFiles – Visualization and Automatic Layout of Graphs*, pages 173–192. In Jünger and Mutzel [2003]. The papers in this book were originally published in [Mutzel et al., 2002]. (Cited on page 38.)

WilmaScope [2006]. *WilmaScope*. Web Site. http://www.wilmascope.org. (Cited on page 33.)

Josef Wolte [1998]. *Information Pyramids*. Master's thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, Austria. http://www.iicm.edu/thesis/jwolte.pdf. (Cited on page 47.)

yWorks [2005]. *yFiles Developer's Guide*. yWorks GmbH, Vor dem Kreuzberg 28, 72070 Tübingen, Germany. http://www.yworks.com/products/yfiles/doc/developers-guide. (Cited on page 38.)

yWorks [2006]. *About yFiles*. Web Site. http://www.yworks.com/en/products_yfiles_about.htm. (Cited on page 38.)