Dynamic Exploration of Large Graphs

Bernhard Tatzmann

Dynamic Exploration of Large Graphs

Master's Thesis

at

Graz University of Technology

submitted by

Bernhard Tatzmann

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

4th March 2004

© Copyright 2004 by Bernhard Tatzmann

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

Dynamisches Erforschen von großen Graphen

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Bernhard Tatzmann

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

4. März 2004

© Copyright 2004, Bernhard Tatzmann

Diese Arbeit ist in englischer Sprache verfasst.

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

Abstract

In a world where information is one of the most important resources, dynamic exploration of large information structures is becoming more and more important. Visualising information as graphs and providing the possibility to browse them is one possibility to make this task easier. This thesis describes a new graph drawing approach, combining both static and dynamic graph drawing algorithms.

The static graph drawing algorithm is based on the hierarchical drawing approach introduced by Sugiyama and places nodes on concentric circles around a focused node. The user can browse by changing the focus, which causes a change in the layout of the graph. The transition from the old to the new layout is smoothly animated using a dynamic graph drawing algorithm which moves clusters of nodes as rigid objects. This approach, which has been integrated into an existing graph drawing package called *JMFGraph (Java Modular Framework for Graph Drawing)*, is explained in detail. A modular overview of JMFGraph and a user guide complete the thesis.

Kurzfassung

In einer Welt in der Information eines der wichtigsten Güter ist, kommt auch dem dynamischen Erforschen von großen Informationsstrukturen eine immer größere Bedeutung zu. Die Visualisierung von Information mit Hilfe von Graphen und die Bereitstellung der Möglichkeit einer interaktiven Besichtigung ist eine Möglichkeit diese Aufgabe zu vereinfachen. Diese Diplomarbeit beschreibt ein neues Verfahren für die Darstellung von Graphen, das aus einer Kombination eines statischen und eines dynamischen Algorithmus besteht.

Der statische Algorithmus basiert auf Sugiyamas hierarchischem Verfahren für die Darstellung von Graphen und ordnet Knoten auf konzentrischen Kreisen rings um einen fokussierten Knoten an. Der Benutzer kann durch Verändern des Fokus durch den Graphen navigieren, was eine Veränderung des Layouts zur Folge hat. Um den Übergang vom alten zum neuen Layout zu animieren wird ein dynamischer Algorithmus verwendet, der Gruppen von Knoten als feste Objekte bewegt. Dieses Verfahren, welches in ein bereits bestehendes Graphenzeichenpaket mit dem Namen JMFGraph integriert wurde, wird im Detail beschrieben. Ein Überblick über die einzelnen Module von JMFGraph und ein Benutzerhandbuch vervollständigen die Diplomarbeit.

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.

Danksagung

Am Ende meiner Diplomarbeit und meines Studiums angelangt möchte, ich all jenen Menschen meinen Dank aussprechen die mich während der letzten Jahre unterstützt haben.

Zunächst danke ich meinem Betreuer Prof. Keith Andrews, der mich durch seine fachliche Kompetenz, insbesondere was das wissenschaftliche Arbeiten betrifft, durch den oftmals schwierigen Prozess des Verfassens einer Diplomarbeit führte. Er war stets bereit auftretende Fragen zu diskutieren und förderte den Softwareentwicklungsprozess durch entscheidende Hinweise und Anregungen.

Von ganzem Herzen danken möchte ich meiner Freundin Renate, die seit der ersten Stunde meines Studiums an meiner Seite ist. Sie freute sich mit mir über Erfolge und stand mir ebenso in weniger erfreulichen Stunden bei. Sie war mein sicherer Rückhalt und meine Motivation. Ihr habe ich es zu verdanken, dass ich mich hier in Graz nie alleine fühlte.

Zu unermesslichem Dank bin ich auch meinen Eltern verpflichtet, die mich sowohl moralisch wie auch finanziell unvergleichlich unterstützt haben. Ebenso für ihre finanzielle Unterstützung danke ich meiner Tante Uschi, wie auch meinen Großmüttern. Mein besonderer Dank geht auch an meinen Bruder Oliver, seine Frau Birgit, meinen Neffen Emil, sowie an Verena und Brigitte.

Für die gute Zusammenarbeit während des Studiums möchte ich meinen Kollegen Helmut und Martin danken. Außerdem geht mein Dank auch an meine besten Freunde Michael und Andreas, sowie an alle anderen, die mit mir familiär oder freundschaftlich verbunden sind.

Bernhard Tatzmann
Graz, Austria, März 2004

Credits

- Figure 7.2 is taken from [Sch98] and is used with permission.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis a new graph drawing approach is presented, which can be used to dynamically explore very large graphs. It consists of a new static layout algorithm and a dynamic graph drawing algorithm which smoothly animates transitions between layouts. This approach is integrated into an existing framework for graph drawing called *JMFGraph (Java Modular Framework for Graph Drawing)*.

Chapter 2 provides an introduction to graph drawing. Basic definitions concerning graph theory and graph drawing are summarised and a survey on existing static graph drawing approaches is given.

In Chapter 3, the dynamic graph drawing approach is introduced. It focuses on methods to create smooth transitions between layouts rather than creating very similar layouts without many changes. The important term *mental map* is explained and it is shown how clustering can help to improve the results of dynamic approaches.

Chapter 4 surveys existing packages and applications in the field of static and dynamic graph drawing. It also summarises existing file formats for storing graphs.

The JMFGraph framework, upon which this thesis was built, is introduced in Chapter 5. The functions of its many modules are described. Modifications and extensions to the basic framework which were incorporated during this work are also described.

Chapter 6 describes the first part of the new graph drawing approach. An existing dynamic graph drawing technique is adapted for layered drawings. A clustering method is introduced for grouping nodes performing similar transitions. Also the user interface for controlling the dynamic approach and screenshots of example transitions are given.

In Chapter 7, a new Sugiyama-style layout algorithm, the second part of the new graph drawing approach, is introduced. All its steps, including the last step, which places nodes on concentric circles, and has never been used before in this combination, are described in detail. The drawing approach is also compared with circular tree drawing methods. Furthermore, its user interface and some example layouts are shown.

Other new functions, which have been integrated into JMFGraph, are summarised in Chapter 8. It includes the description of a new parser for handling GraphML files, the usage of Lagrange curves for representing edges with control nodes, and the export module for exporting graph layouts to SVG files.

An outlook to future trends concerning graph drawing is given in Chapter 9. Also ideas for further improvements to JMFGraph are discussed.

Appendix A contains a user guide to JMFGraph. It explains the functions of all user interface dialogues and lists all keyboard shortcuts and possible mouse interactions.

Chapter 2

Graph Drawing

Graphs represent information which should be conveyed to the viewer through drawing. The better the information is structured, and the better the drawing is arranged, the more information can be conveyed. The way graphs are drawn is the key to bringing as much information as possible to the user. While good layouts can help to filter out the information quickly, bad layouts (for example with many edge crossings) can make it very hard to see what is being represented.

Every information structure consisting of linked objects can be represented by graphs. Objects are represented as *nodes* and *edges* represent connections between them. Figure 2.1 shows a drawing of London's tube map. If this map were to be represented by a graph, the stations would be represented by nodes and the tube lines by edges. Since the lines do not have distinct directions, undirected edges would be used. A graph with directed edges is shown in Figure 2.2. It shows the evolution of the *Unix* operating system.

As can be imagined there is a wealth of different possibilities for using graphs for visualisation. The most common applications for *graph drawing* in general are visualising process flow diagrams, the architecture of modules in software projects, linked computers in a network, and so on. Different kinds of information are represented by different kinds of graphs. Different kinds of graphs need different kinds of drawing algorithms.

After introducing basic definitions of graph theory, this chapter explains certain characteristics of graph drawing. Then the most important static graph drawing approaches are presented. A comprehensive book on graph theory is [Har72]. [dBETT99], [KW01] and [Sug02] give good surveys about graph drawing methods.

2.1 Graph Theory

First of all, some mathematical definitions of graphs and their properties have to be made. Basic definitions concerning graph theory can be found in [Har72]. However, [dBETT99] describes a graph as it can be found in all graph drawing literature, as a tuple $G = (V, E)$, where V is a finite set of *vertices*, which are also called *nodes*, and E is a finite set of *edges*. Every edge is itself an unordered pair $e = (u, v)$ of two vertices. [Har72] defines *adjacency* as the property of two vertices u and v which are connected by an edge. The edge e itself is called *incident* to its nodes u and v. The number of edges incident to a certain node is called the *degree* of this node.

If two vertices are connected by more than one edge this is called a *multi edge*. A *self-loop* defines an edge which connects a point with itself. If a graph does not contain multi edges and self-loops it is called a *simple graph*.

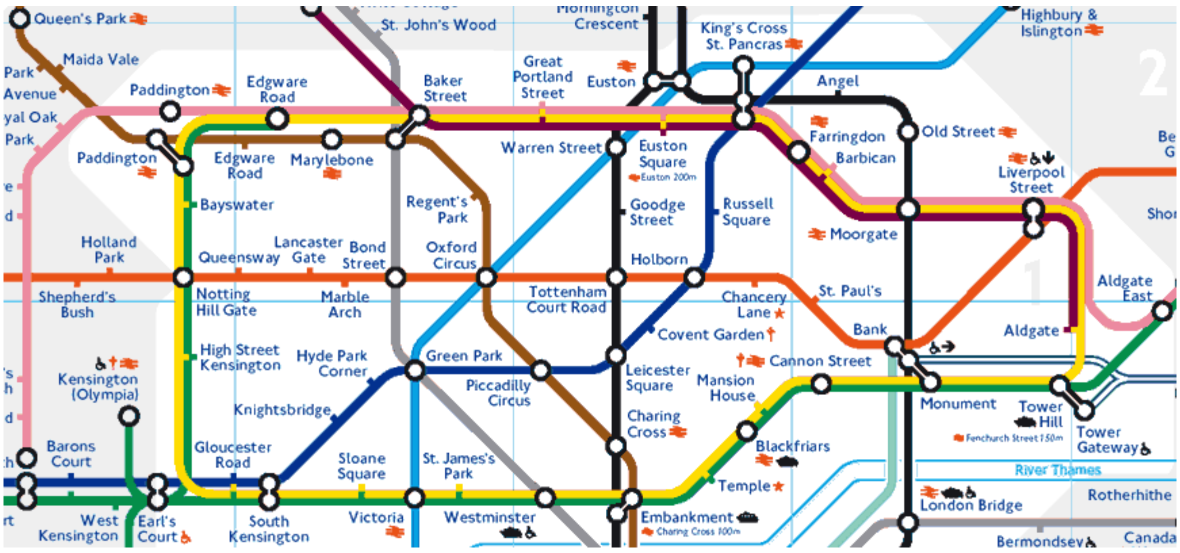


Figure 2.1: Screenshot of London's tube map available at *TubeGuru* [Lon04]. A tube map is a good example of a possible application of graph drawing. Stations would be represented by nodes and the tube lines by undirected edges.

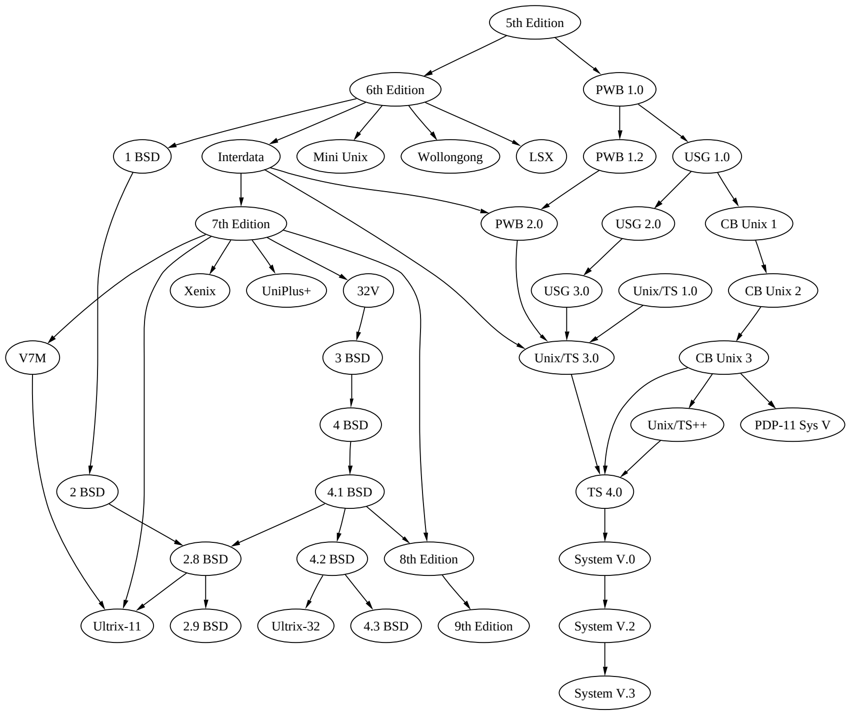


Figure 2.2: Example of a directed graph showing the evolution of the Unix operating system. The layout was generated using the batch graph drawing tool *Graphviz* (see Section 4.1.2).

As mentioned before, the finite set of edges consists of unordered pairs of vertices. If this is the case then the graph is called *undirected*. However, if the pairs of vertices are ordered, meaning the edges have an *origin* and a *destination* then the graph is called *directed*. Directed graphs are usually called *digraphs* in the graph drawing literature.

Besides breaking up graphs into directed and undirected, they can also be divided up into *cyclic* and *acyclic* ones. Before giving the definition of a *cycle*, a *path* has to be defined. A path is a sequence of vertices $v_1, ..., v_n$ where vertices v_{i-1} and v_i are connected by an edge. If, additionally, an edge exists which connects v_1 and v_n then the graph contains a cycle and is called cyclic.

An important subset of graphs consists of graphs which are acyclic and additionally *connected*. Such graphs are called *trees* and are known beside their importance for graph theory [Har72] as data structures and possibility for representing hierarchies in every day life [KW01]. [KW01] splits up the group of trees into *rooted trees* and *free trees*. In free trees every node is as much important as the others. On the other hand rooted trees contain a special marked node called the *root*. Usually the roots do not have any incoming edges. Since trees are acyclic there is always only one path from the root to every other node. As will be seen later, many graph drawing algorithms have been developed especially for trees.

2.2 Characteristics of Graph Drawing

2.2.1 Classes of Graphs

For now graphs have been only described as abstract mathematical objects. If a graph is to be drawn, a location has to be assigned to every node. [CdBT+92] defines the *drawing* of a graph as Γ which maps every node of a graph to a point in the plane. The edges $e = (u, v)$ are represented as Jordan Curves connecting u and v. Other possibilities for representing the edges will be given later in Section 2.2.2.

The mapping of the abstract graph to the plane leads to the distinction of *planar* graphs and *non-planar* graphs. If it is possible to map the nodes to the plane in a way that no two edges cross, this is called a *planar drawing* and the graph is called a planar graph. Otherwise the graph is non-planar. Figure 2.3 shows a non-planar graph called $K_{3,3}$. As it can be seen there is no permutation of nodes possible which leads to a crossing- free drawing. Such graphs are confusing for the viewer, because it is hard to trace the edges.

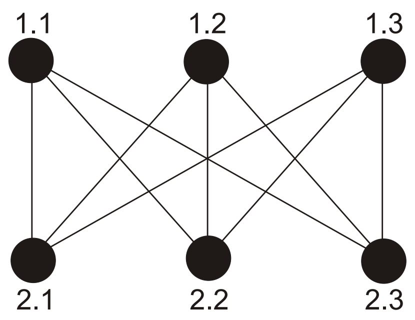


Figure 2.3: The non-planar graph $K_{3,3}$.

[Har72] calls the regions in the plane caused by a planar drawing *faces*. The one which is unbounded is called the *outer face*. Following [KW01] the infinite number of possible drawings of the same planar graph can be subdivided into groups of equal *planar embeddings*. Two drawings belong to the same planar embedding if the edges forming the boundary of the faces have the same circular order. Figure 2.4 illustrates this by showing two different embeddings of the same planar graph.

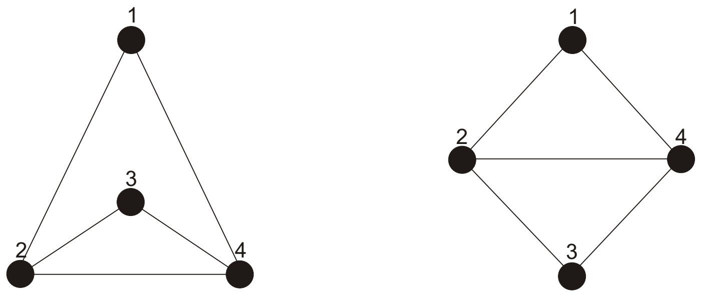


Figure 2.4: Two different embeddings of the same planar graph.

Another property of a graph which has to do with the drawing of the graph splits graphs into *upward* and *downward* graphs. A drawing is called *upward* if the representation of every edge (u, v) of the underlying graph $G = (V, E)$ is drawn as a curve (or straight line) with monotonically nondecreasing y-value while starting from u and going to v. A directed graph is called upward if and only if such an upward drawing can be obtained [KW01].

Following [dBETT99] graphs are called *biconnected* if they do not have any *cutvertices*, where cutvertices are nodes which would cause a graph to become *disconnected* if they were removed. Referring to this the *blocks* of a graph are the maximal sub graphs that are still biconnected.

2.2.2 Conventions

Drawing Conventions are certain properties which *must* be fulfilled by a drawing. These properties deal with representation and placement of nodes and edges. [Sug02] divides drawing conventions into conventions concerning the placement of the nodes and conventions concerning the rooting of the edges. The latter is again divided into line types and the reliance on the coordinate system. A complete list of all conventions given in [Sug02] together with explanations can be found in Table 2.1.

Nodes could, for example, be placed freely in the plane, on certain layers, or on a grid. Edges could be represented by straight lines, polylines, or curved lines. A certain convention for nodes and a certain convention for edges together build the full drawing convention. Figure 2.5 shows a drawing placing the nodes on an orthogonal grid and edges as polylines routed along the lines of the coordinate system.

2.2.3 Constraints

Conventions define rules for graph drawing which are applied to the entire drawing of a graph. However, sometimes it is the case that certain regions or subgraphs have to be treated in a special way because the semantics require it [CT94]. Since graph drawing algorithms in general do not understand the semantics of the underlying graph, *constraints* can be used to give additional drawing rules to graph drawing algorithms for those parts of the drawing.

A list of possible constraints is published in [TdBB88] and summarised in [CT94], [dBETT99] and [Sug02], were they are introduced as *semantic rules*:

Type	Drawing Convention	Explanation
Node Placement	Free	Nodes can be placed in plane without any restrictions on coordinate systems, grids or layers.
	Parallel lines	Nodes are placed on or between parallel lines which are also called layers.
	Concentric circles	One node is chosen to be in the middle of the drawing and the other nodes have to be placed on or between concentric circles around it.
	Radial lines	Nodes are placed on or between radial lines emanating from a point in the middle of the drawing.
	Orthogonal grids	Nodes can only be placed on an integer grid.
	Polar grids	Nodes are placed on the intersections defined by the concentric circles and radial lines of a polar grid.
Edge Representation	Straight-line	Edges are drawn as straight-lines connecting the nodes.
	Polyline	Edges are drawn as sequence of one or more straight-lines.
	Curved line	Edges are drawn as curved lines of a certain types.
Reliance of Edge Routing on Coordinate System	Dependent	Edges are drawn along the lines of the coordinate system. For example, given an orthogonal grid and polylines, every single line has to be drawn either horizontally or vertically along a grid line.
	Independent	The edges can be drawn freely without restriction of the coordinate system.

Table 2.1: Taxonomy of graph drawing conventions made by [Sug02].

- **Centre:** One or more important nodes should be placed in the middle of the drawing.
- **Dimension:** This can be used to manually specify the representation size of certain nodes.
- **External:** In contrast to *Centre* this is used to place nodes at the boundary of the drawing.
- **Neighours (or Cluster):** Nodes belonging semantically together can be placed in a common neighbourhood by using this constraint.
- **Shape:** A subgraph can be drawn with a given shape.
- **Stream (or Sequence):** Several nodes are drawn along a straight line, which makes it easier to trace possible edges connecting them.

2.2.4 Aesthetics

Like conventions, *aesthetics* are rules which should be automatically fulfilled by the graph drawing algorithm without any user interaction. Aesthetics describe layout properties which lead to nice drawings in general. The number of edge crossings is an example of aesthetics. Besides reducing the number of edge crossings a couple of other ideas for improving the readability of drawings are introduced in [STT81]. This list is completed in [TdBB88] and [Sug02], where aesthetics are called *structural rules* in contrast to the *semantic rules* above.

The most important aesthetics are:

- **Area:** The area needed by the entire drawing should be as small as possible. This ensures that, given a certain zoom factor, as much as possible of the drawing can be seen on screen.
- **Angle:** The angle between the edges of one node should be maximised.
- **Aspect Ratio:** The ratio of the drawing's length and breadth should be balanced.
- **Bends:** The number of edge bends especially for orthogonal drawings should be minimised.
- **Convexity:** Faces of planar drawings should be drawn as convex polygons.
- **Crossings:** Edge crossings make it difficult to trace paths. So all crossings should be removed if the graph is planar, if not the number should be minimised.
- **Symmetry:** Symmetry in a hierarchical drawing, where sons should be placed symmetrically, and symmetry of the entire layout.
- **Total edge length:** The average of all edge lengths should be as small as possible.

It is not generally possible to fulfill all aesthetics at the same time. Sometimes the improvement of one property causes a worsening of another. For example, crossing minimisation often leads to an increase of edge lengths. So it is an optimisation problem to achieve the right mixture of all aesthetics.

2.2.5 Computational Complexity

Computational complexity has always been important for static graph drawing algorithms and it does not lose its importance nowadays, where dynamic algorithms are becoming more popular. As can be imagined interaction with dynamic systems should happen in real time, so computing the layout of the next drawing step has to be done quickly. Since many drawing algorithms or subalgorithms are optimisation problems and are often NP-hard, heuristics are often used.

2.3 Static Graph Drawing Methods

2.3.1 Orthogonal Graph Drawing

Orthogonal graph drawing is motivated by the aesthetic that the angle between the edges of one node should be maximised. Besides the maximisation of the angles of adjacent edges orthogonal graph drawing algorithms also consider many other aesthetics, like the minimisation of edge crossings, bends and total edge length, and constraints, like specified placement of certain nodes. It is based on the drawing convention that places the nodes on an orthogonal grid, uses polylines as edge representations and routs the edges along the coordinate systems. Figure 2.5 shows an orthogonal graph drawing.

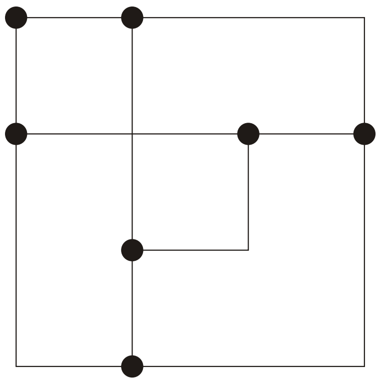


Figure 2.5: Example of orthogonal graph drawing.

Orthogonal drawings have been developed for *general undirected graphs*. The method is also called the topology-shape-metrics approach [dBETT99] or graph theory approach [Sug02] and is divided into three steps:

Planarisation: The *planarisation step* takes a general undirected graph and produces a planar representation. Since general graphs might be non-planar, the number edge crossings is reduced first and the remaining crossings get removed by inserting dummy vertices at positions where the crosses occur. The substeps of planarisation are the following:

1. First of all the given graph $G = (V, E)$ is split into its blocks (for a definition of blocks see Section 2.2.1).
2. For each block the maximal subgraph which is still planar is extracted. The crossing edges which have not been used yet will be inserted later on.
3. To achieve a planar drawing with a minimum of nested faces, the following is done. The blocks are inserted into a tree starting with the largest as root. Then recursively blocks with common cutvertices (of the entire graph) are inserted as children. When this is finished the face having the most cutvertices of every block becomes its outer face.
4. This step is only necessary for non-planar graphs where the set of crossing edges (not used in step 3) is not empty. These edges get inserted now, while trying to keep the number of crossings as small as possible.

Normalisation and Orthogonalisation: In this step an *orthogonal representation* is computed from the planar representation of the previous step. Since orthogonal drawings route their edges only horizontally or vertically, the maximal possible degree of every node is four. If a node exceeds this property it has to be replaced by two or more nodes so that the maximum degree is reduced to four. This mechanism is called *normalisation*. Now the orthogonal representation can be computed. This is done with several heuristics, always trying to keep the number of bends as small as possible. As output of this step every edge has a list of angles, which describe the bends. Real coordinates will not be assigned to every node until the final step.

Compaction: The *compaction* converts the orthogonal representation into a grid representation of the same shape, with real coordinates. This is also the step where a minimisation of edge lengths is done. This is achieved by converting every face into a rectangle. Then a linear integer program is used to minimise the edge lengths of all rectangles. When these lengths are computed the positions of all nodes are determined and the orthogonal representation is ready to be drawn.

For further reading [TdBB88] and [Sug02] can be recommended. In [KW01] many heuristics are presented.

2.3.2 Drawing on Physical Analogies

Some graph drawing methods compute representations with the help of *physical analogies*. These kinds of graph drawing algorithms use the drawing convention that places the nodes freely in the plane and uses straight lines which are routed independently of the coordinate system as edge representations. They are quite popular for several reasons. Since they use analogies like metal rings (nodes), springs (edges) and magnetic fields they are very intuitive. Furthermore they are easy to implement and produce good results [KW01]. In Figure 2.6 the *Kamada-Kawai algorithm*, which will be described later on, has been used to generate a drawing.

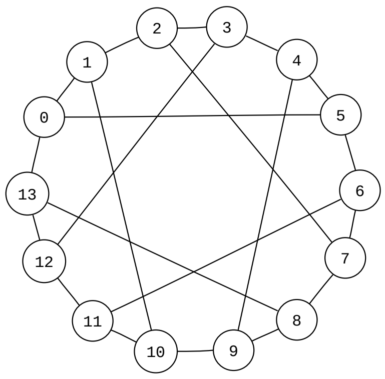


Figure 2.6: Drawing based on physical analogies, created with *Graphviz* using the *Kamada-Kawai algorithm*.

Two physical analogies are used: *force-directed placement* and *energy-based placement*. The first tries to minimise the internal energy of the system by iteratively modifying the node positions responding to forces between the nodes. Instead of moving nodes the latter minimises these energies directly [KW01].

Force-Directed Placement

The method presented in [Ead84] was the first force-directed placement method and is called the *spring embedder*. Following [Sug02] the method can be used for general undirected graphs. It models the graph as a system of iron rings and springs. There are two different kinds of spring, one connecting adjacent nodes and the other connecting nodes which do not have a common edge. His algorithm first places the nodes in random order. Then the forces working at each node caused by the springs are computed and the nodes are moved. This is repeated a fixed number of times.

Another method based on the spring embedder was published in [SM95], *The Magnetic Spring Model*. In contrast to the spring embedder this method can be used for directed graphs and trees as well. As before undirected edges are represented by springs, while directed edges are modeled now as magnetic springs. These magnetic springs force the iron rings (nodes) to rotate in a magnetic field. Several different kinds of magnetic fields, such as parallel polar or concentric fields, are used. The edges can be compared with compass needles rotating in the magnet field of the earth. The freedom of varying parameters like the strength of the field or the forces of the springs make it possible to fulfill numerous different aesthetics. The algorithm itself is in principle the same as the one used by the spring embedder method.

Energy-Based Placement

The *Kamada-Kawai algorithm* presented in [KK89] and summarised in [KW01] uses springs that are proportional in length to the length of the shortest path between two nodes. Adjacent nodes are connected by springs proportional to one. This leads to a sum of all potential energies. Since this is an energy-based placement method, the energy of the system is minimised by directly minimising this sum. This is done with a modified Newton-Raphson algorithm, which moves the node with the longest gradient in each iteration step.

Sometimes another method called *simulated annealing* [DH96] is used to minimise the energy of the system. This method was originally developed for VLSI layout and is very powerful. On the other hand, it is very slow and so it can not be used for interactive graph drawing systems [Sug02]. It often happens that minimisation methods only find local minimas. Simulated annealing tries to overcome this problem by learning from an analogy to the process of cooling down a liquid, a process called annealing. The idea is that as liquid should not be cooled down too quickly, the system energy should not be minimised too fast. This is achieved by introducing the *temperature* parameter to indicate the energy of the system. This makes it possible that new arrangements of the graph not leading to an immediate minimisation of the system energy get a second chance later on depending on their temperature.

For a survey of force directed drawing methods see [Coh97]. A new algorithm for drawing huge graphs based on the minimisation of an energy function is described in [KCH02]. The algorithm is called *ACE*, which stands for *Algebraic multigrid Computation of Eigenvectors*. It minimises an energy function called *Hall's energy* by expressing it as an eigenvalue problem. With this method it is possible to draw graphs with a million nodes in less than twenty seconds.

2.3.3 Hierarchical Graph Drawing

Hierarchical graph drawing algorithms are designed to represent general digraphs. These methods use the graph drawing convention that places nodes on layers and uses straightline or polyline edges which are rooted independently of the coordinate system (see Figure 2.7). Usually, drawings are made upward or downward. Sugiyama presented the basic method for hierarchical graph drawing in [STT81], upon which all other methods are based. The method was extended in [ES90] and is traditionally divided up into four steps:

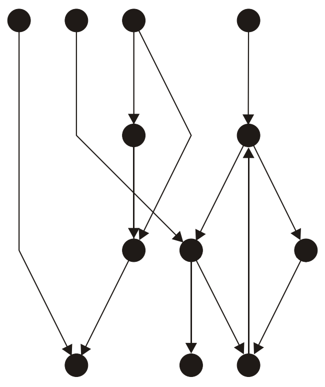


Figure 2.7: An example of hierarchical graph drawing, where nodes are placed in layers. Layout generated with *JMFGraph*.

1. Cycle Removal:

   Since upward or downward drawings are only possible (see Section 2.1) if graphs are acyclic, any cycles have to be temporarily removed. This is done, either by replacing a cycle by a single node or by reversing edges pointing in the wrong direction and later restoring the reversed edges. The set of reversed edges should be as small as possible. The minimisation of this set is called the *feedback arc set problem*. Since this problem is NP-hard a number of heuristics have been invented. Some of these algorithms are described in [ES90].

2. Layering:

   In this step every node of the graph is assigned to a certain layer $L_1, ..., L_n$. Since the graph is now acyclic, this can be done in a way that for every edge $u \rightarrow v$, u is in layer L_i and v is in layer L_j such that $i < j$. This implies that all edges are pointing downward. The original method by Sugiyama proposed in [STT81] layers the graph by inserting all nodes which do not have any incoming edges, called *root nodes*, in the first layer. Then the nodes which have just been inserted are removed from the data structure and the new root nodes are inserted into the next layer. This is repeated until there are no remaining nodes. A number of other methods satisfying different requirements, such as making layerings not too wide and not too high, have been proposed. Important algorithms are the *Longest Path Layering* and the *Coffman-Graham-Algorithm* which are described in [ES90]. Another important task achieved in this step is the insertion of *dummy nodes*. Dummy nodes are needed when edges connect nodes in non-adjacent layers. Such edges are assigned dummy nodes at each layer they cross.

3. Crossing Reduction:

   As has already been explained in Section 2.2.4 one of the most important aesthetics is reducing the number of edge crossings. This step tries to find a permutation of the nodes in every layer such that the total number of crossings is minimised. Since this combinatorial problem is again in NP, heuristics have to be used. One method known as the *layer by layer sweep* moves upward and downward through all adjacent layers, using a heuristic to minimise the number of crossings between those two layers. The most popular heuristics are the *barycentre heuristic*, *median* *heuristic*, *split heuristic*, *switch heuristic* and *sifting algorithm*, all summarised in [KW01]. A second type of method approaches the problem by considering all levels simultaneously, the *k-layer crossing minimisation approach*. Two examples for this method are Tutte's algorithm [ES90] and global sifting [MSM99].

4. X-Coordinate Assignment:

   Since the crossing reduction step only computes relative x-coordinates by computing the order of the nodes, the last step assigns absolute x-coordinates to them. In doing this some further aesthetics can be fulfilled. The nodes can be aligned in a way that bends are reduced and symmetry is improved, without changing the ordering of the previous step. It should not be forgotten, that the width of the drawing should not become too large. Sugiyama suggests two solutions to this problem [STT81]. The first, a deterministic algorithm called *Quadratic Programming Layout Method*, tries to optimise the problem by expressing the aesthetics as sums which have to be minimised. The other method is a heuristic approach called *Priority Layout Method*, which tries to optimise by sweeping through the layers. Finally, any dummy nodes have to be removed and reversed edges restored.

All methods using these steps are called *Sugiyama style algorithms*. The original method, presented in [STT81], integrates the Cycle Removal step into the layering process and introduces a final step after the X-Coordinate Assignment, which maps the layout to the drawing space.

Besides the hierarchical drawing algorithms using the convention with nodes placed on layers, possibilities using concentric or radial node placements have been investigated. Sugiyama suggested in [STT81] using a radial node placement. In such layouts the first and last layer are adjacent and so graphs with many feedback loops can be drawn better.

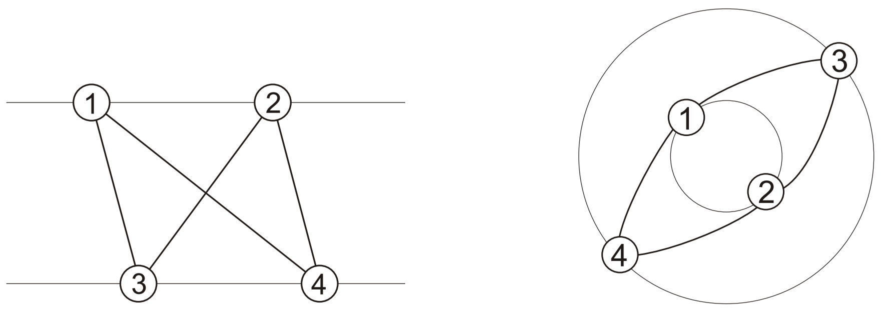


Figure 2.8: Layout with parallel lines and concentric circles of the same graph. The concentric layout method can reduce the number of edge crossings. [RM88]

[Car80] presents a drawing approach for cyclic directed graphs, which places nodes on concentric circles. The layering is done using a so-called *number hierarchy*, where nodes with a minimum distance to all other nodes are placed on the smallest circle. A crossing reduction heuristic, called *the Relative Degree Algorithm*, is also introduced. In [RM88] another method is presented using hierarchical layouts with concentric lines. They use the full angle of the circle to reduce the number of crossings. A simple graph such as the one shown in Figure 2.8, which can not be drawn without crossings by using two parallel line layers, can be drawn without crossings with their method, using two concentric circles.

As it can be imagined, the improvement in the number of crossings decreases with an increasing number of layers [Sug02].

2.3.4 Tree Drawing Methods

Trees are a special case of graphs which are very common. According to [dBETT99] three different kinds of methods are used for drawing trees:

1. Layered Drawings:

   Since trees can be seen as acyclic graphs (with edges pointing away from the root) , the hierarchical drawing approach of Section 2.3.3 can be easily adopted for trees. Since trees have no cycles, layering is done by taking the nodes' depths. Edge crossings do not occur if children of different parents have the same ordering as their parents in the previous layer. Absolute x-coordinates are assigned with respect to the aesthetics of balance, symmetry and least distance between the children of one node.

2. Circular Drawings:

   By placing the root node in the middle of the drawing and mapping the layers to concentric circles *circular drawings* can be achieved. To avoid crossings between subtrees, a sector called *annulus wedge*, is assigned to every subtree. In principle these sectors are proportional in size to the number of nodes in the subtree. Generally this method leads to pleasing planar circular drawings [HMM00].

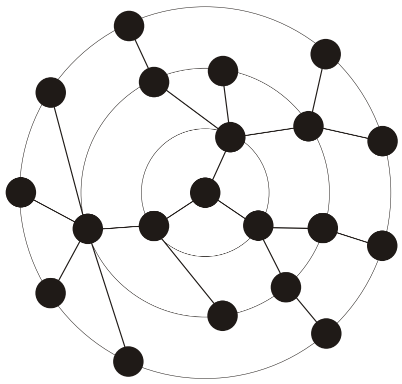


Figure 2.9: Circular drawing of a tree, where the root node is placed in the centre and layers radiate outwards in concentric circles. Layout generated with *Graphviz*.

   In [YFDH01] this method is applied to general undirected graphs. This is done by ignoring edges creating cycles (which would not be in a tree), while the node's positions are computed. They call the nodes which are connected by such edges *non-tree neighbours*.

3. Orthogonal Drawings:

   The *orthogonal* method is only applicable to binary trees, where each node has exactly two children. Such a child node is either placed beside its parent, to its right, on the same horizontal line or exactly below. This is repeated by *divide and conquer* until all nodes are aligned. Similar to radial drawings every subtree is assigned its own area so that no overlapping occurs.

2.3.5 Visibility Approach

The *visibility approach* uses the drawing convention that places nodes freely in space, connected by polylines which are routed independently of the coordinate system. Following [dBETT99] it is divided into three steps. *Planarisation* is the first step and is the same as used in orthogonal graph drawing. The second step, the *visibility* step, represents nodes with horizontal and edges with vertical line segments. A vertical line starts at the horizontal line belonging to its first node and ends at the horizontal line assigned to its second node. It is not allowed that vertical edges cross any other lines. The last step replaces each horizontal line by its node and the vertical lines by polylines connecting the nodes.

2.3.6 Augmentation Approach

The augmentation approach starts again with the *planarisation* step. Followed by the *augmentation* step, which adds additional edges and nodes to the graph obtained from the first step. This is done to convert faces with more than three edges to triangles. Now it is possible to draw the graph as a *triangulation* by using the edges of the second step. Finally the dummy edges and nodes can be removed [dBETT99].

Chapter 3

Dynamic Graph Drawing

In contrast to Chapter 2 where static drawing algorithms are described, this chapter deals with the problem of graphs which change over time. Graphs can change for several reasons. If the graph represents a structure, for example a computer network, which changes over time, nodes and edges have to be added or removed. Another possible application is a graph drawing toolkit where the graph could be modified by the user. A third possibility is the one used in the JMFGraph framework described in this thesis, where the graph changes because the user navigates through it, viewing only a small subset at any one time.

A straightforward method for handling these changes would be to compute a new layout from scratch, which could be totally different from the previous one, and redraw the graph. However, this approach has the problem that users can loose their *mental map*. To overcome this problem two different methods have been invented. The first one tries to minimise the changes between the old and the new drawing by modifying only the parts of the drawing where the underlying graph has changed and keeping those parts that have remained the same. The second method uses animation to smoothly transform the old graph into the new one. Since the latter is used in the JMFGraph framework this method will be described throughout this chapter.

3.1 Mental Map

The term *mental map* was first used in [ELMS91]. When a user watches a graph being drawn, it takes a little while until it is possible to orient. After orientation the user has a mental map. When the drawing changes dramatically, the user looses orientation and the learning process has to be started again.

[MELS95] describes three properties which help to understand the mental map in a more formal way. The first, called the *orthogonal ordering*, describes how nodes are aligned. For example node u is, to the left of v and below w. The *proximity relations* describe the distances between the nodes. The *topology* depicts the regions in space created by the graph. Using a transition does not keep these properties unchanged, but does give the user the possibility to follow the modifications.

As it can be seen in Figure 3.1, the insertion of a single edge can radically change the layout of the nodes. An instant switch from the first to the second drawing without any transition would cause users to loose their mental map and have to reassimilate the new drawing. Using a transition, like in Figure 3.2, it would be easy to follow the movements of the nodes.

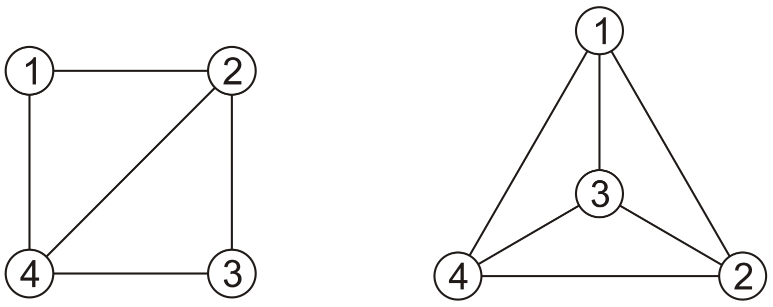


Figure 3.1: Insertion of a single edge can cause a major change in layout [Sug02].

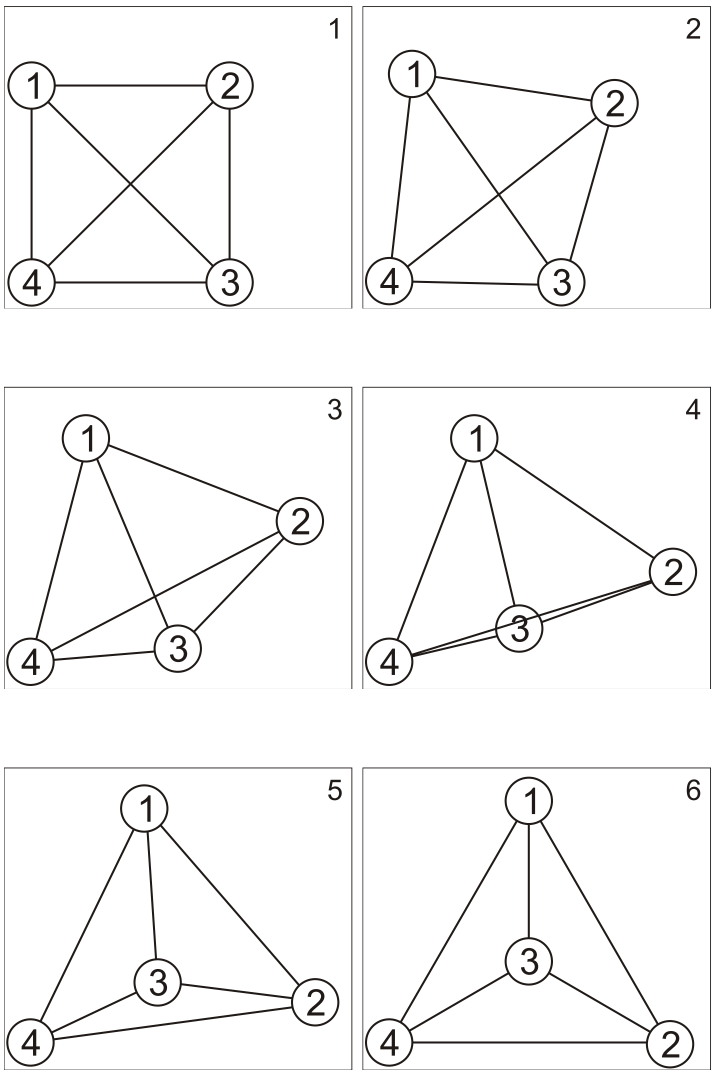


Figure 3.2: Using a smooth animated transition maintains object constancy and allows viewers to track changes and preserve their *mental map*.

3.2 Animated Graph Drawing

A straightforward method to transform one layout into another would be simple linear interpolation of the node positions. According to [FH01] most existing approaches use this method. However, this approach can cause several problems. As can be seen in Figure 3.3 transforming layout (a) into (e) with linear interpolation leads to a collapse of the drawing during animation at (c). To avoid this and other problems Friedrich and Eades summarise some criteria for good animations [FE02].

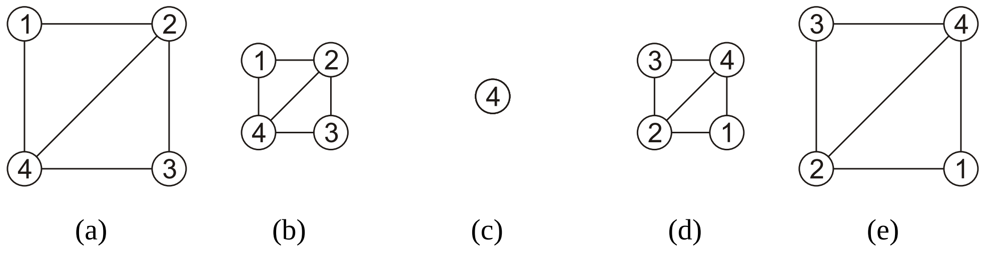


Figure 3.3: When linear interpolation is used for the transition from (a) to (e), all nodes meet in the middle and the drawing collapses at point (c).

3.2.1 Criteria and Measures for Good Animations

In [FE02] the following criteria, which should help the user to follow the changes in the layout, are presented:

- **Use movements that are easy to follow:**

  It should be as easy as possible for the user to trace the movement of nodes and edges.

- **Use structured movements:**

  If the cognitive abilities of the human brain are respected, the effort for the user in following the movements can be minimised. The first thing helping the user is to use movements that are as *symmetric* as possible. The second point concerns the *relative positions* of objects as described in Section 3.1. If these relative positions of objects in the first layout do not differ much from those in the second layout, they should also be kept during the transition because the human brain can easily follow movements of structures. This principle is called *object constancy* and is introduced in [RCM89]. Finally, humans are familiar with projections of the movement of *three dimensional objects*. A rotating cube for example can be recognised even if only its two dimensional projection is seen. Nodes can be clustered into objects, which perform movements in all three dimensions to reach their final positions.

- **Use smooth transitions:**

  The movements of nodes and edges should be *smooth*. Therefore the step size should not be too large and not too small and the speed should not be too fast and not too slow. Steps which are too large or a speed which is too high would make it hard to keep track. On the other hand very small steps or too slow transitions could cause a loss of attention.

- **Avoid showing non-existing structures:**

As was shown in Figure 3.3 it can happen that certain nodes or edges collapse during transition. If that happens it could seem that there were structures that do not exist. Showing such structures should be avoided.

- **Use intermediate frames with proper layout:**

  The layout of the drawings during the animation should fulfill the aesthetic criteria presented in Section 2.2.4.

Besides giving some aesthetic criteria for animated graph drawing, there are also some possibilities of measuring the quality of the animations presented in [FE02].

- **Number of edge crossings:**

  A count of the number of edge crossings during animation is one measure of the quality of the animation. Since edge crossings should not appear in static graph layouts, they should also be avoided in intermediate frames.

- **Minimal distances of nodes belonging to different structures:**

  If certain nodes are not close to each other in the initial and final layout, they should also stay apart during animation. Minimal distances can be measured.

- **Changes in relative positions:**

  If nodes have similar relative positions in the initial and final layout, this should not change during animation. This measure evaluates the change of those relative positions while they are moved from start to end positions.

- **Symmetry of movement:**

  Since symmetrical movements are easier to follow, the symmetry of movements should be measured as well.

- **Path lengths of node movements:**

  Shorter paths are easier to follow than longer paths. So path lengths are another measure for quality. However, minimising path lengths alone would lead to linear paths for all nodes.

- **Speed of animation:**

  Finally the animation speed, which should be neither too slow nor too fast, could be measured as well.

3.2.2 Animation Using Rigid Motion

The animation process presented in [FE02] is based on the idea of structured movements. Nodes are treated as one *rigid object*, which moves through three dimensional space trying to bring all nodes as near as possible to their final positions. The algorithm can be used for all kinds of graphs. Since dynamic graph drawing algorithms also have to deal with removed and added nodes and edges the algorithm is split into four steps:

1. Fade out disappearing nodes and edges:

   Nodes and edges which are present in the initial layout but do not belong to the final layout have to be removed from the drawing before any kind of movement can begin. Since this is also part of the animation, this has also be done in a smooth way which does not confuse the user. These objects are *faded out* at an appropriate speed.

2. Move nodes using rigid motion:

   The second step uses an affine transformation to bring all nodes as close as possible to their final positions. An affine transformation consists of four operations: *translation*, *scaling*, *rotation* and *shearing*. Transforming a two dimensional point with an affine transformation matrix is done with the following formula:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_{11} \\ t_{21} \end{pmatrix}$$

   The transformation matrix which is needed to transform the points to their final positions, can be found by inserting the start coordinates for (x, y) and the end coordinates for (x', y') in the equation above. So every node has the following pair of equations:

$$x' = a_{11} \cdot x + a_{12} \cdot y + t_{11}$$

$$y' = a_{21} \cdot x + a_{22} \cdot y + t_{21}$$

   Since there are six unknowns, three pairs of equations will be needed. This means that three non-collinear pairs of nodes are needed to determine the affine transformation matrix. Since most of the time there will be more than three nodes, which do not have a common matrix that transforms the nodes exactly to their end positions, a function is needed that minimises the errors. In [FE02] this is done by minimising the sum of the squared Euclidean norms of the vectors starting at the computed end position and ending at the real end position of every node:

$$e = \sum_{n=1}^{N} (a_{11} \cdot x_n + a_{12} \cdot y_n + t_{11} - x'_n)^2 + (a_{21} \cdot x_n + a_{22} \cdot y_n + t_{21} - y'_n)^2$$

   When the affine transformation matrix has been found the interpolation process can be started. Therefore a parameter t is used that runs from 0 to 1. The step size determines the speed of the animation. The translation vector can be easily interpolated by computing $t \cdot (t_{11}, t_{21})$. The interpolation of the 2×2 matrix is more complicated. Since the node position at the beginning of the animation has to be exactly the one defined by (x, y) the interpolation of this matrix has to start with the identity matrix, being smoothly transformed into the computed affine matrix. This interpolation can cause problems. As mentioned in [FE02], the linear interpolation from the identity matrix I to an affine transformation matrix M, which describes a rotation of π leads to a null matrix as intermediate matrix:

$$0.5 \cdot \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}}_{I} + 0.5 \cdot \underbrace{\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}}_{M} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

   In such a case all nodes would collapse into one point. To avoid this, the rotational part Q has to be extracted from M.

$$M = Q \cdot S$$

   This operation is called *polar decomposition*. Several methods for doing this are presented in [SD92]. The rotational part can then be interpolated over the angle, which can be computed from the extracted rotation matrix, and S can be interpolated linearly:

$$Q_i = \begin{pmatrix} cos(t \cdot \varphi) & -sin(t \cdot \varphi) \\ sin(t \cdot \varphi) & cos(t \cdot \varphi) \end{pmatrix}$$

$$S_i = (1 - t) \cdot I + t \cdot S$$

3. Move nodes to their end positions:

   Since the second step can not move all nodes exactly to their final positions most of the time, this step is needed. [FE02] suggests several methods for this last movement. The easiest method is to move nodes by individual *linear interpolation*. Another suggested approach is to split the movement into a first movement in the x-direction and a second in the y-direction.

   [FE02] also proposed a modification of a *dynamic force-directed algorithm* to fulfill this last positioning step. While static force-directed approaches (described in Section 2.3.2) only display the final layout, dynamic force-directed algorithms also show the intermediate frames. Such a dynamic force-directed algorithm can be found in [HEW98]. When using force-directed algorithms the nodes' final positions are determined by the algorithm. Friedrich and Eades needed an algorithm, to move nodes to positions already determined by the computed final layout. So they had to modify the dynamic force-directed method in such a way, that forces are used to attract the nodes to their final fixed positions. Since this approach is more computationally difficult than linear interpolation method, their problem was to make it efficient enough to keep the transitions smooth.

4. Fade in new nodes:

   Nodes which exist in the final layout, but were not present in the initial layout are faded in this last step. To help the user this has to be done again with appropriate speed.

3.2.3 Clustering-Based Animation

The method described in Section 3.2.2 works well as long as all nodes perform the same movement. If certain nodes or whole subgraphs have to be moved in different directions, this method does not produce appropriate animations anymore (see Figure 3.4). Friedrich and Houle [FH01] improved the method above by introducing clustering methods into the animation process. Nodes performing similar movements are assigned to the same cluster (see Figure 3.5).

Since trying out all different possibilities of grouping nodes together would lead to exponential running times, they suggest two different heuristics.

- **K-Means Clustering**

  K-means clustering is a well-known method used in various fields. It first creates an initial partitioning $P = P_1, ..., P_k$ of the objects. Then an average property vector of every partition is computed. A new partitioning is created by assigning every object to the partition with the most similar property vector. This is repeated until no objects change their partition anymore. K-means clustering is very easy and leads to good results.

  Using k-means for animation with rigid motion works as follows. After performing the initial partitioning the affine transformation matrix for every cluster is computed. A new partitioning is created by assigning every node to the partition with the affine matrix that moves the node

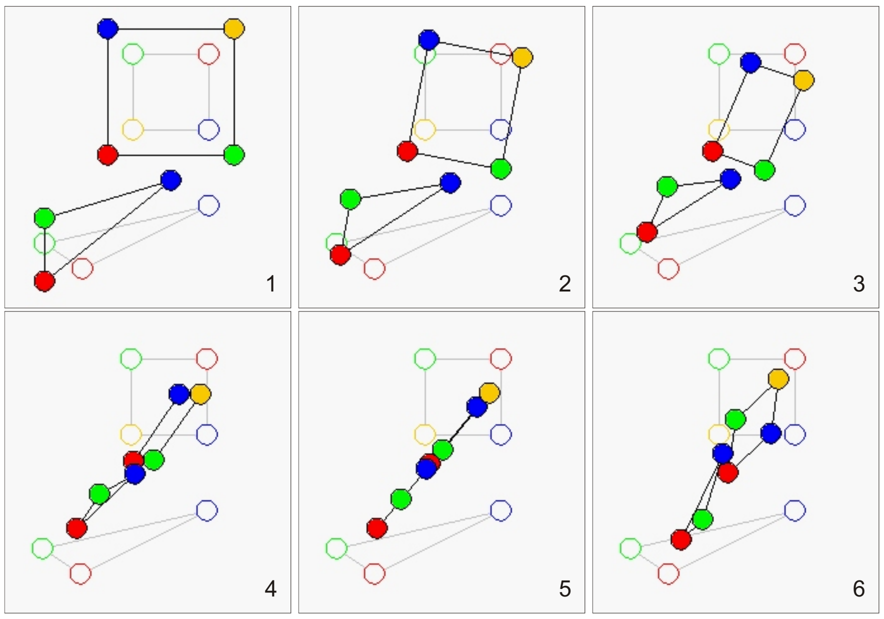


Figure 3.4: Animating two objects with different movements as one rigid object. The affine transformation matrix which is used for all points can not bring the nodes to their exact end positions.

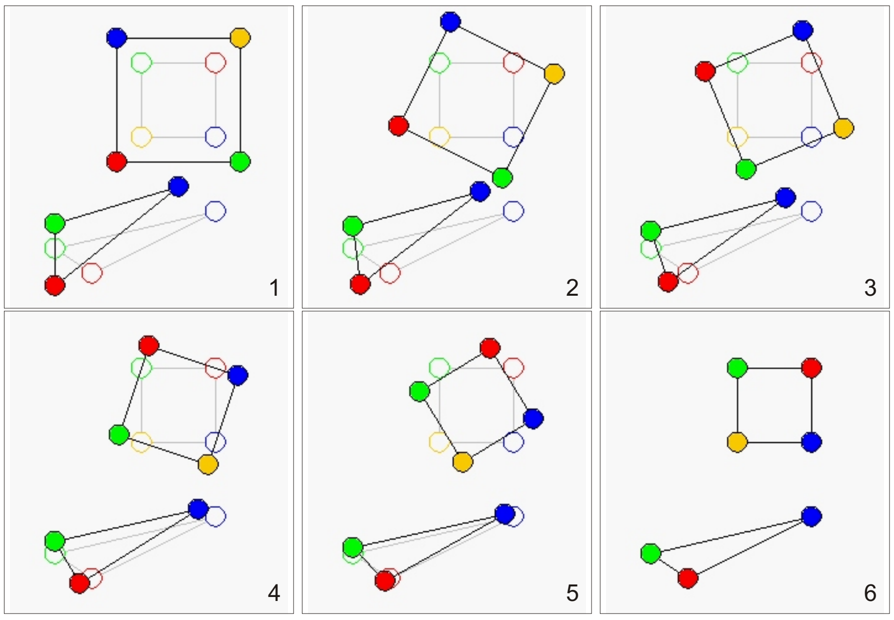


Figure 3.5: Animating two objects with different movements using *clustering*. Every cluster gets its own transformation matrix, which can bring the nodes to their final positions.

closest to its end position. This is done until the total error, which is the sum of all distances between the real final positions and the one reached by the transformation matrix, converges.

Friedrich and Houle [FH01] found that the algorithm finds the individual movements most of the time. They use ten clusters as a maximum, because too many different movements would be hard to track. Since the initial partitioning has a large effect on the results of the clustering process, they also suggest also a second kind of heuristic.

- **Distance-Based Clustering**

  Distance-based clustering methods respect the distances of the nodes in the initial or in the final layout. They are based on the idea that nodes lying close together will also perform similar movements. In [FH01] a *delaunay-triangulation* with *edge elimination* is used. First the triangulation of the set of nodes is computed. Initially every triangle is a partition. Then triangles lying next to each other are merged, by removing the edge between them, if they have similar transformations. The choice of the threshold which determines if the transformations are similar enough is very important for the performance of this method. The method can be used on its own as a clustering method or can be used for computing the initial partitioning for the k-means method.

3.2.4 Intersection-Free Animation

The methods above were designed for general graphs. Edge crossings can occur during animation even if there are no crossings in the initial and final drawing. In [KP03] a method for planar graphs is presented that even generates intermediate frames without edge crossings. They give two restrictions. The start and end layout have to have the same outer faces and the same planar embeddings. Additionally it is fine if the two graphs are *isomorphic*. Isomorphic means, that there exists a one-to-one correspondence between the nodes of both graphs, so that each node of the first graph has a corresponding node in the second graph with the same adjacencies (see Figure 3.6)[Har72]. If they are not isomorphic the disturbing edges are removed in a preprocessing step. The algorithm is divided into the following four steps:

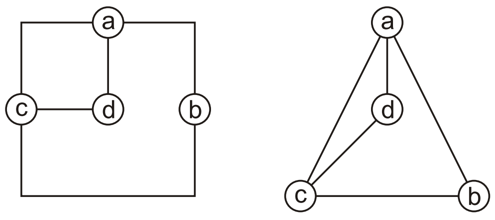


Figure 3.6: Isomorphic graphs, initial and final layout [KP03].

1. Transformation using rigid motion:

   This step works exactly like step two in the method presented in Section 3.2.2. It uses an affine transformation to move the nodes, which are treated as a rigid object, as close as possible to their end positions. For finding the best affine transformation matrix they also use the method presented in [FE02], which minimises the sum of the squared Euclidean distances between real and computed end positions.

2. Insertion of additional nodes at bends:

   The second step inserts nodes at all positions, where edges have bends (see Figure 3.7 (a)). If one edge is given such additional nodes in one drawing, it is also given them in the other drawing. If curved edges are used, they have to be approximated by straight lines.

3. Triangulation of faces:

   In this step the two layouts are triangulated. To do this, first the initial and the final layout, with additional nodes inserted in the last step, are triangulated independently from each other. As it can be seen in Figure 3.7(b) this leads to different faces in both drawings. To find out where the differences are, both layouts have to be overlayed. At positions where this overlay leads to crossing edges, *Steiner points* are inserted to make the triangulations compatible. The compatible triangulation of both graphs can be seen in Figure 3.7(c).

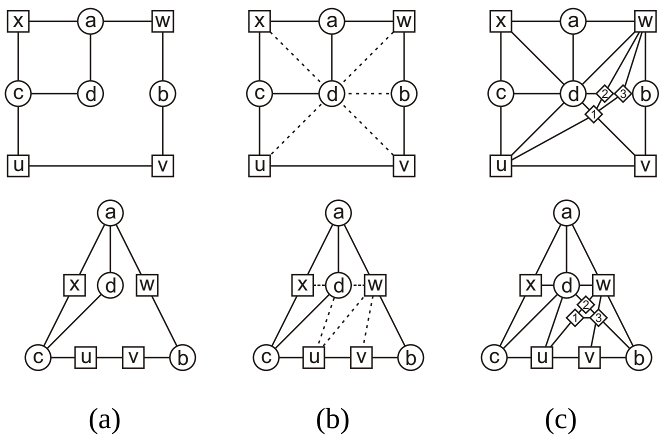


Figure 3.7: Step 2 and 3: (a) Insertion of additional nodes, where edges have bends. (b) Independent triangulation of both graphs. (c) The triangulation of (b) is extended using *Steiner points* to include the same faces [KP03].

4. Morphing of faces:

   Once the layouts have been triangulated, the faces can be interpolated. This process is called *morphing*. Simply speaking every face is represented by its barycentre. Then matrices with these barycentres for the faces in the initial and final layout are created. The animation can then be processed by interpolating these two matrices. The barycentre of every face is interpolated with respect to the positions of its neighbours.

3.2.5 Animation using Polar Coordinates

Another method, which is not as complex as the methods described above, but goes further than simply using straight line movement, is presented in [YFDH01]. As has already been described in Section 2.3.4 they use a modified tree drawing algorithm to layout graphs. To make transitions between those drawings, they work with polar coordinates instead of rectangular coordinates. When nodes move to another position on the same layer, they travel along the concentric arc. If they change layer, their movements are spiral-shaped (see Figure 3.8). They do not use explicit clustering methods, but nodes with the same parent node always move together without changing their order (if it is not required).

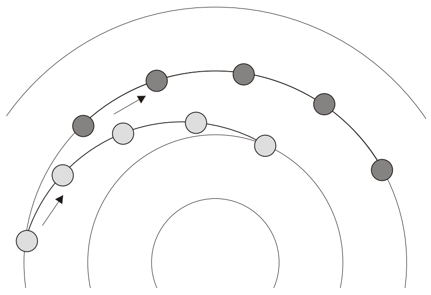


Figure 3.8: Radial movement of nodes, within the same layer, and moving to another layer [YFDH01].

Another interesting aspect of the method proposed in [YFDH01] is the speed of transition. Nodes do not move at constant speed from their initial to final position. Instead, they accelerate at the beginning starting from zero and slow down to zero at the end. This gives the user more time to prepare to track the movement.

[vWN03] deals with appropriate speed of movements. It does not describe the right speed of the transition between two drawings, but the path of a camera that moves from one point in a drawing to another one. This is done by zooming out first, panning to the new position, and then zoom in. The appropriate speed for this is computed by using a spherical space.

Chapter 4

Graph Drawing Packages and Applications

This chapter describes graph drawing packages and specific applications. Since there is a wealth of different packages and applications a survey of those most important in general, and in relation to dynamic drawing algorithms will be presented. File formats used for storing graphs are introduced.

4.1 Graph Drawing Packages

Graph drawing packages can be subdivided into graph drawing editors, toolkits, frameworks, libraries and so on. Some are freely available for download, even as source code, some are accessible via online applets, and some are commercial. Most of them are graph drawing libraries that are available as standard demo applications integrating the most common operations. They all implement one or more of the standard static graph drawing methods described in Chapter 2.

4.1.1 GDToolkit - Graph Drawing Toolkit

GDToolkit [GDT03] is a C++ library, for drawing and layouting graphs. GDToolkit is based on LEDA, which is a C++ library, that supports certain data structures, algorithms and tools for creating applications. The current release 3.0 includes three different packages.

GAPI (*Graph Application Programming Interface*) provides different kinds of graphs with drawing algorithms. The drawing algorithms include orthogonal drawings, upward drawings, and the visibility approach. The second package included is called BLAG (*Batch Layout Generator*). Batch layout generators take input files describing the graph, apply a layout algorithm, and generate an output file. Except for choosing the files and the algorithm, there is no user interaction most of the time. The third package includes interactive demo applications. The *gdtdemo* application can be seen in Figure 4.1.

GDToolkit is free available for non-commercial users. Precompiled binaries for *Linux*, *Solaris* and *Windows* can be downloaded.

4.1.2 Graphviz

Graphviz [GV03] is open source graph drawing software. Like GDToolkit it includes a graph editor and batch layout generators. There are three different layout engines called *dot*, *neato*, and *twopi*. dot

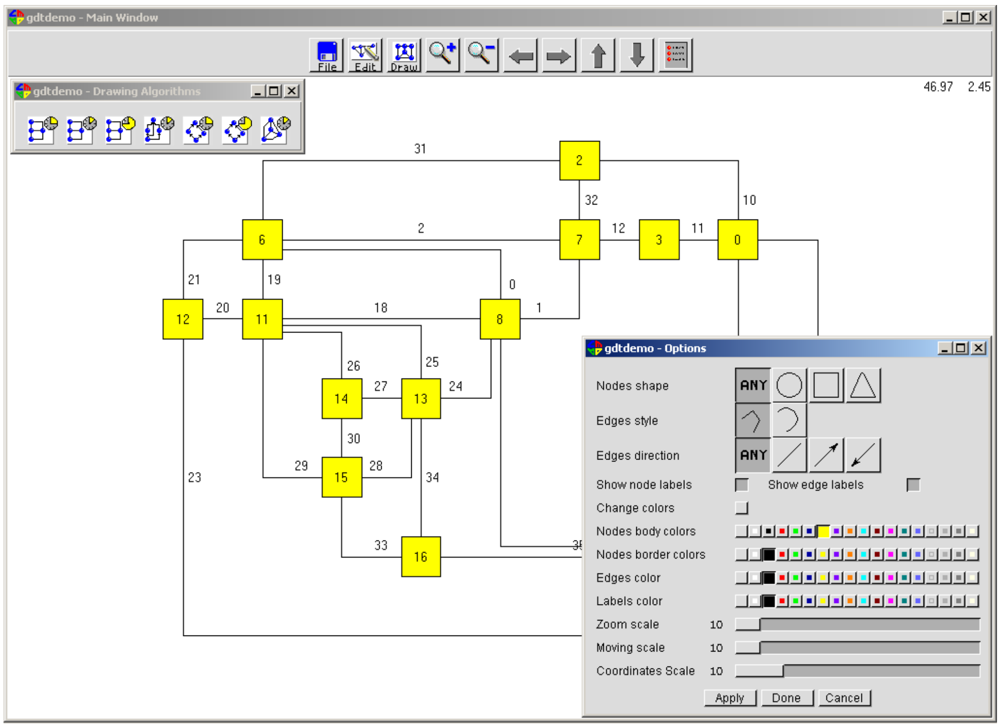


Figure 4.1: *GDToolkit*: Screenshot of the demo application *gdtdemo*. A panel for choosing between different drawing algorithms and window for changing the properties of nodes and edges can be seen [GDT03].

can be used to generate hierarchical layouts for directed graphs with curved lines. neato and twopi can both generate layouts for undirected graphs. The first uses an energy-based placement method, while the latter generates circular layouts.

The only supported format for the input files is also called dot. It was invented especially for Graphviz and is described in Section 4.3. Output can be chosen from many different file formats. As well as there are pixel-oriented formats like GIF and PNG, there are also vector oriented formats like PostScript and SVG.

The latest version, Graphviz 1.10, includes executable binaries for *Linux*, *Unix* and *Windows*. Older version are also available for *Solaris*, *Mac*, *Irix* and so on.

4.1.3 Graph Visualisation Toolkits

Tom Sawyer Software offers three commercial graph drawing toolkits. *Graph Analysis Toolkit* (GAT), *Graph Editor Toolkit* (GET) and *Graph Layout Toolkit* (GLT). GET and GLT are described in [DFM$^+$02].

The Graph Analysis Toolkit includes libraries which can be used for developing graph analysis software and are available in Java and C++. GET is also offered in Java and for developing in .NET. It provides various tools for developing graph editors. Standard operations like zooming and panning, event handling, menus and so on can be used. GLT offers standard layout algorithms: hierarchical, orthogonal, symmetrical and circular drawing approaches are available. This library is only offered in C++.

All packages are usable for common operating systems like *Linux*, *Windows*, *Solaris*, *Mac OS* and *Irix*. An online demo in the form of a Java applet can be accessed after registration to try out the most common operations [GVT03].

4.1.4 aiSee

aiSee is based on the code of the free source tool VCG [San94] which is still available for *UNIX* and an old *Windows* version [VCG04]. VCG was developed to produce fast layouts of graphs generated by other programs. There are several possibilities for controlling the layout algorithm to produce the layouts as fast as the user wants. If those computations can not meet the time constraints set by the user, the program automatically continues with a faster algorithm. Besides the euclidean view of the graph, also a fisheye view is provided, which distorts the layout and enlarges the part in the middle.

A trial version of the current release 1.90 of aiSee can be downloaded for most common operating systems [aiS03]. Non-commercial users can even get license keys for usage until 2005. aiSee provides Sugiyama-style algorithms with different constraints like producing maximum or minimum depth, or minimising the number of backward edges. The tool provides a huge number of options for controlling the layout like choosing different crossing reduction heuristics, numbers of algorithm iterations or edge styles. For exporting the layout there are several file formats, including PostScript, SVG and PNG, available. See Figure 4.2 for a fish-eye view of a civilisation model in aiSee.

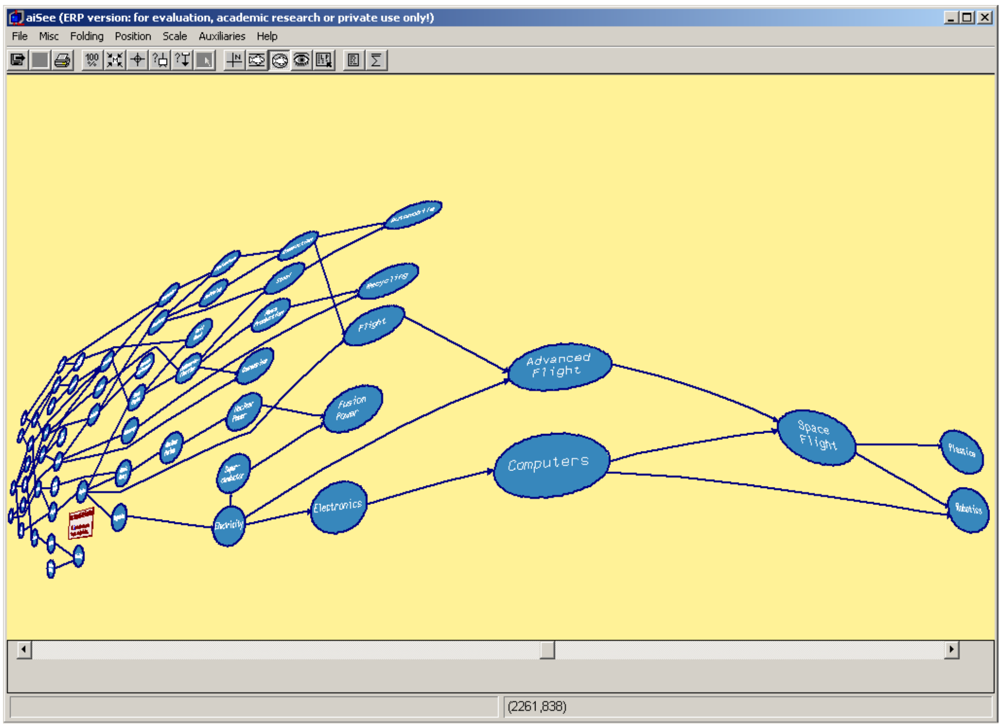


Figure 4.2: aiSee, which is based on the code of the tool VCG [VCG04], provides a huge number of possibilities for controlling the layout algorithm. Here a fish-eye view of a civilisation model with Sugiyama-style layout can be seen [aiS03].

4.2 Dynamic Graph Drawing Software

It is often not enough that tools provide only static graph drawing methods. Sometimes dynamic graph drawing methods can be necessary as well. The most common task where they are needed is the inspection of very large graphs with many thousands of nodes. Other possibilities for applying dynamic graph drawing methods are changing graphs because of added and deleted nodes or switches to other layout algorithms. This section presents some example tools which use dynamic graph drawing methods.

Following [HE98] there are two main approaches for handling graphs with more than a thousand nodes: navigation and clustering. Programs using navigation only show a small subgraph at once and provide navigation functionality. Clustering based tools reduce the number of nodes by condensing subgraphs to single nodes, which represent a cluster. The tool described in Section 4.2.2 uses navigation. It provides a survey view of the graph to help the user to navigate. If the focus is changed, the tool pans smoothly to the new focused node. The tool in Section 4.2.3 uses the clustering approach. If clusters are created or destroyed this gets animated. There are also tools which do not use standard graph drawing methods to viusalise large graphs, for example by using hyperbolic views. An example of these tools is introduced in Section 4.2.4.

Also the online demo of the algorithm helping to preserve the mental map by creating crossing free transitions, which was summarised in Section 3.2.4, is presented. First of all, a tool is introduced which uses dynamic graph drawing for making smooth transitions when applying a new layout algorithm or zooming.

4.2.1 AGD - Algorithms for Graph Drawing

AGD [GJK+01] is a C++ library for drawing, manipulating and layouting graphs. It is available as a precompiled binary package containing the libraries and tool demos (see Figure 4.3). Like the GDToolkit, AGD is also based on LEDA.

The library offers a huge number of different possibilities. Graphs can be created by hand or by choosing the number of nodes and the preferred type of graph. They can be made for example connected or biconnected. It can be tested if they are planar or simple. Loops can be removed, and so on. AGD supports certain force-directed layout methods and planar layout methods such as orthogonal layout and the visibility approach. Every time the layout changes, or the user zooms, the transition is animated smoothly.

AGD-R 1.3 is the current version. It is freely available for non-commercial use for *Linux*, *Solaris* and *Windows* at [AGD03].

4.2.2 daVinci

daVinci [FW94] was a non-profit graph drawing package for *Unix* workstations until version 2.1 [daV03]. Since then it is distributed as commercial software called *daVinci Presenter*. For private use and universities time limited licenses are available. The newest version 3.0.5 can be downloaded for *Linux*, *Windows*, *Solaris*, and *FreeBSD* [dVP03].

daVinci Presenter can be used as graph editor and for generating layouts. The layouts can be exported in several file formats. It supports possibilities for hiding sub graphs and selected edges. Besides that, it allows the user to either navigate through the graph by using the arrow keys or by using a survey view, which can be opened in a second window (see Figure 4.4). All of these transitions are animated by panning from one node to another. It does not compute a new layout. Animation can be disabled and the speed can be changed.

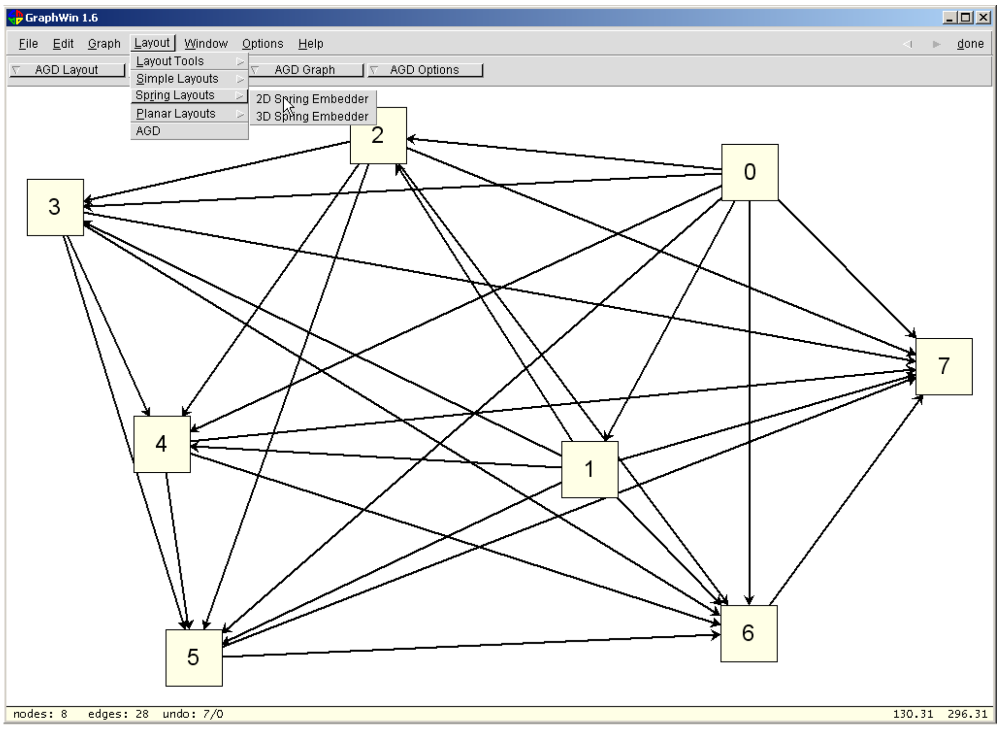


Figure 4.3: Screenshot of the AGD - Library standard tool demo. Every time the layout algorithm changes or zooming is performed the transitions are animated smoothly [AGD03].

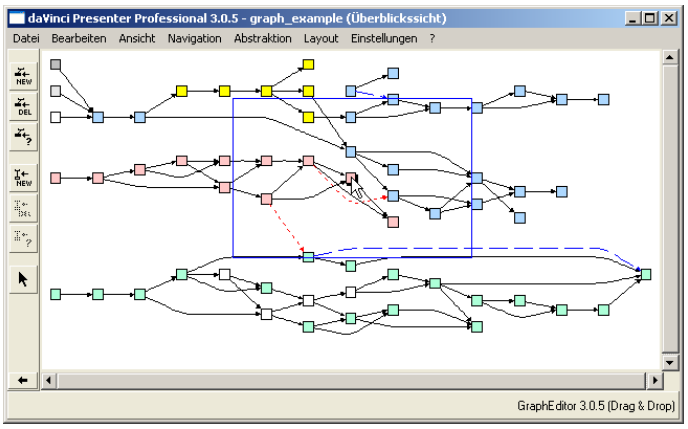
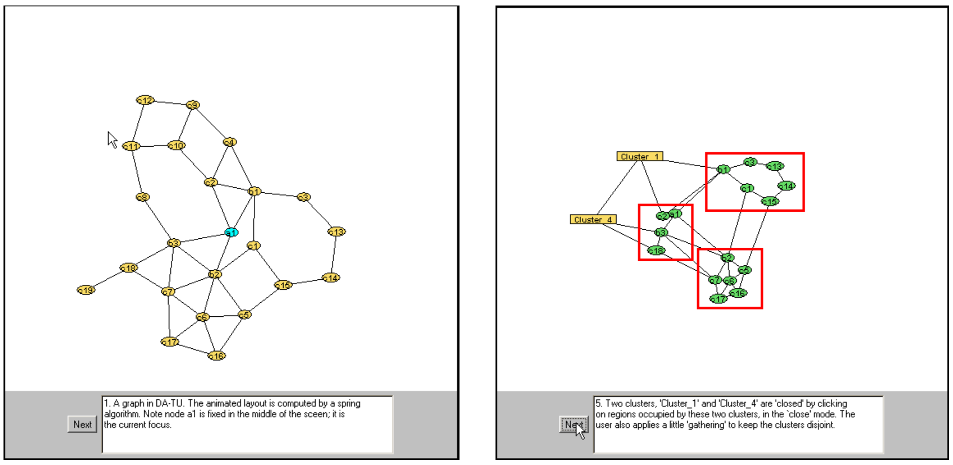

daVinci Presenter Professional 3.0.5 - graph_example (Überblickssicht)
Datei Bearbeiten Ansicht Navigation Abstraktion Layout Einstellungen ?
NEW
DEL
GraphEditor 3.0.5 (Drag & Drop)

Figure 4.4: Survey window of daVinci Presenter. Nodes and edges can be selected to navigate through the graph. Navigation is animated in the main window by smooth panning to the new position [dVP03].



1. A graph in DA-TU. The animated layout is computed by a spring algorithm. Note node a1 is fixed in the middle of the sceen; it is the current focus.

5. Two clusters, 'Cluster_1' and 'Cluster_4' are 'closed' by clicking on regions occupied by these two clusters, in the `close' mode. The user also applies a little 'gathering' to keep the clusters disjoint.

(a) (b)

Figure 4.5: Online demo of DA-TU, which uses clustering to visualise large graphs. (a) shows the original graph and (b) shows the graph with two clusters condensed to single nodes. All modifications are animated smoothly using a dynamic spring embedder [DAT03].

4.2.3 DA-TU

DA-TU [HE98] is a system which simplifies exploration of large graphs by using clustering. The system works with general undirected graphs and uses force-directed placement for layouting. Users can select certain nodes to be clustered. Those nodes are then replaced by a single node (see Figure 4.5), which has the edges of all the nodes it replaces. Clusters can also be destroyed again, revealing the hidden nodes. The user can also interact with the system by adding and deleting single nodes.

What makes the tool interesting relating to dynamic graph drawing is that all modifications performed on the graph are animated using a dynamic spring embedder. The user can see how nodes react to the changed forces in a stepwise manner, which helps the user not to loose orientation. An online demo of DA-TU which provides something like a interactive slideshow of different modifications on a graph can be seen at [DAT03].

4.2.4 H3 - Laying Out Large Directed Graphs in 3D Hyperbolic Space

H3 [Mun97] is another tool for visualising large graphs. In contrast to all the tools described above, it does not use standard graph drawing techniques. Instead of using euclidean space it works with hyperbolic coordinates, since hyperbolic space offers much more volume than euclidean space. To compute the layout the spanning tree of the graph is computed. A spanning tree includes all nodes existing in the graph but only such edges which could appear in a tree. Child nodes are placed on hemispheres around the parent nodes. The layout computed in infinite hyperbolic space is then projected into finite euclidean space for display. The nodes in the middle are in focus and are drawn much larger than those at the border (see Figure 4.6). When the user changes the focus by selecting another node, the new focus node rotates smoothly into the middle.

The program is non-commercial, free for download and open source. Binaries of the current version 1.1.2 are available for *Linux* at [H303]. Older version can be downloaded for *Windows* and *Solaris* as well.

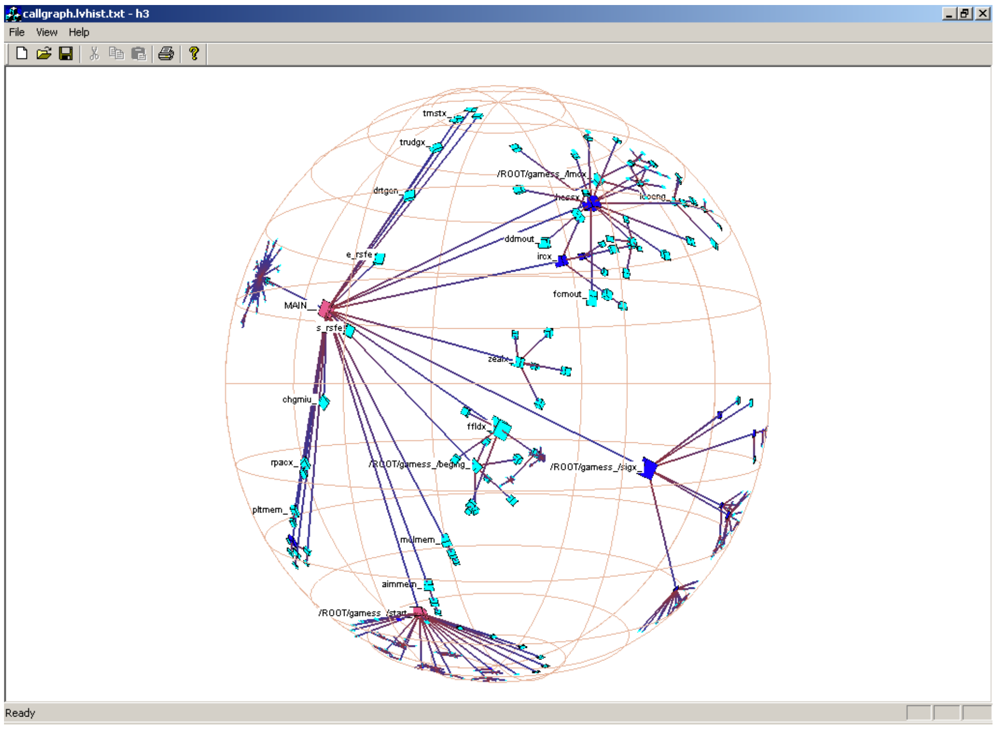
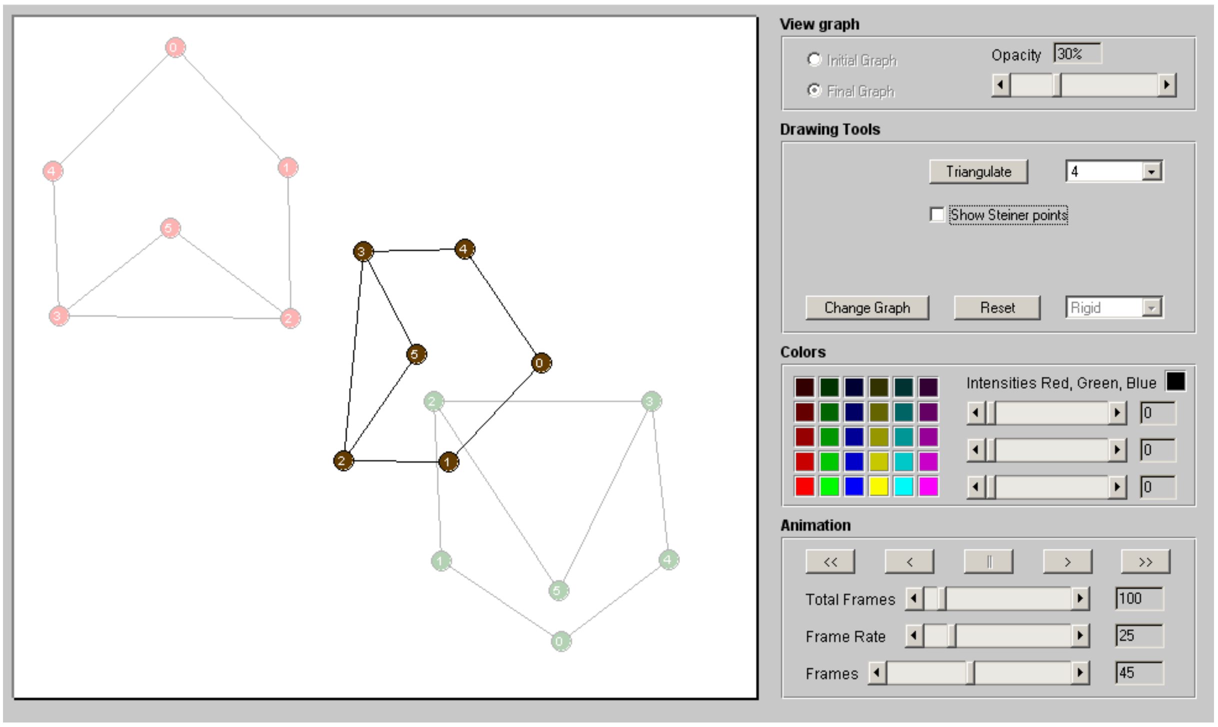


Figure 4.6: 3D hyperbolic viewer H3. The tool visualises huge graphs by computing the layout in hyperbolic space, placing child nodes on hemispheres and projecting this to euclidean space. The focus can be changed smoothly by clicking on the nodes [H303].



Opacity 30%

Triangulate 4

Rigid

Intensities Red, Green, Blue 0 0 0

Total Frames 100

Frame Rate 25

Frames 45

Figure 4.7: Demo applet of the algorithm creating intersection-free morphings, described in Section 3.2.4. As it can be seen, rigid motion moves the object from its initial to its final position without destroying the objects structure [GM03].

4.2.5 Intersection-Free Morphing Demo Applet

This section describes the tool demo for intersection-free morphing of planar graphs [KP03]. The algorithm has already been described in section 3.2.4. As it has been mentioned the algorithm can be useful when the layout of a planar graph changes and a smooth transition, preserving the mental map, should be performed.

The demo is available as a Java applet at [GM03]. First a graph has to be drawn with its initial and final layout. Nodes and edges can be placed freely in the plane. When enabling the animation the applet shows the smooth transition of the nodes from their starting to their end positions. Different types of animation can be chosen used for the transition of the full graph, which is done in the first step of the algorithm. Collapsing of graphs can be avoided by using rigid motion as can be seen in Figure 4.7. The total number of frames and the frame rate can also be adjusted. Additionally, the triangulations of the graphs computed by the system for creating intersection free transitions can be seen.

4.3 Graph File Formats

Graph file formats are needed by graph drawing tools to store and exchange graphs. The latter is becoming more important as graphs are being shared over the world wide web. At the moment this is still not that easy, because most tools use their own formats developed for their own special needs. So attempts have been made to find a standard format satisfying all the needs of the different tools.

That this is not easy becomes clear, when thinking about the differences between the tools. Some support multiple edges or self loops, others do not. Some support animations, others do not. The most important formats as well as the most recent attempt to find a common standard are described in this section. This section is based on the survey of graph drawing formats given in [Hah02].

4.3.1 dot

As mentioned in Section 4.1.2, the *dot* file format was invented for the graph drawing tool *Graphviz*. The formats syntax and semantic are specified in [KN93]. It supports directed and undirected edges as well as subgraphs. A large number of parameters are defined, which can be used to specify the visualisation of the entire graph, nodes or edges, or their labels. While shape and rendering properties can be defined, the format does not support possibilities to determine absolute or relative node positions. As one of the earliest graph file formats, dot is currently propably the most widespread and can save a lowest common denominator format.

4.3.2 GML

GML (Graph Modeling Language) [GML03] was invented for the graph drawing tool *Graphlet*. It was a first attempt for introducing a common graph file format. Based on the existence of several text and graphics interchange formats, such as Postscript or HTML, GML was first discussed at the Graph Drawing Symposium in 1995. Since that it has been used by many tools and there are many graphs available in this format [Him96].

The files are structured very simply, by defining nodes with ids and edges using those ids to identify source and target. Other attributes can be added to the structure and are simply ignored by programs that do not understand them.

4.3.3 XGMML

XGMML (eXtensible Graph Markup and Modeling Language) [XGM03] is an XML[1] graph file format, which is based on GML. Conversion from GML to XGMML is very easy and can be done using XSL[2] . XSL can also be used to transform XGMML files to other graph file formats. The format was invented for a program visualising web sites as graphs.

4.3.4 GraphXML

GraphXML [HM00] is another XML graph file format invented to interchange graphs. It was created to serve the needs of information visualisation. It supports standard tags for shape and rendering properties as well as tags for positioning nodes. Users can extend functionality by creating their own DTD[3] files. Besides the static graph description this format also supports dynamic properties such as storing the history of the actions applied to the graph by the user. This history can be used to animate changes made to the graph.

[1]XML (eXtensible Markup Language)
[2]XSL (eXtended Stylesheet Language)
[3]DTD (Document Type Definition)

4.3.5 GXL

GXL (Graph eXchange Language) [HWS00] was developed to share data among software reengineering programs. Every tool represents its data with a graph and hands it over to the other tools. However it can be used for graph visualisation programs as well. It supports multiple edges between nodes. Nodes and edges can have attributes with names and values. The current DTD and XML schemas can be found at [GXL03].

4.3.6 GraphML

GraphML is the latest attempt to introduce a common graph file format. The initiative was started at the 8th Symposium on Graph Drawing (2000) [BMN00]. There it was decided that the format should be structured in layers including structure, topology, shape, geometry and rendering. It should also be extendible to satisfy the needs of special applications. At the Graph Drawing Symposium in 2001 a progress report [BEH$^+$01] was presented. Future work will include modular extension to the basic features.

GraphML includes support for various kinds of graphs but does not define any properties concerning the visualisation. Extensions do not effect tools which do not know the extensions. XML schemas and DTDs can be found at [Gra03].

4.4 Graph Drawing Applications

This section introduces applications for graph drawing. However, most designed for a specific field of use such as visualisation of process flow diagrams, architectures of modules in software projects or computers linked in a network. [Sug02] introduces five kinds of applications, including documentation, monitoring, browsing, graph editors and idea support tools. They all use either static or dynamic graph drawing techniques. The survey of applications given in this section focuses mainly on dynamic approaches.

4.4.1 gnuTellaVision

gnuTellaVision [GTV03] is a good example of an application of dynamic graph drawing. The layout and motion algorithms [YFDH01] were presented in Sections 2.3.4 and 3.2.5. The tool uses these techniques to allow browsing through the file sharing network Gnutella.

The focused host can be seen in the middle, while the hosts to which it is connected lie on concentric lines around it. Every host is represented by a node whose size is proportional to the number of files stored. Edges symbolise the connections between the hosts. If new hosts are discovered the layout is adapted. By clicking on another node, the user can choose a new centre of focus. To preserve the mental map all transitions are performed smoothly. The tool allows the user to understand the structure of the network very easily and quickly.

4.4.2 Visual Thesaurus

The *Visual Thesaurus* by *plumbdesign* is a tool using graph drawing to visualise semantic connections between the words of the english language. The tool accesses *WordNet* [Wor03], which is an online semantic network providing lexical references between words. Users can type in new words or click on words already existing on screen to find out which other words are related to the one chosen. The user can decide which relationships, such as antonyms, “see also”, “is similar to” and so on, should

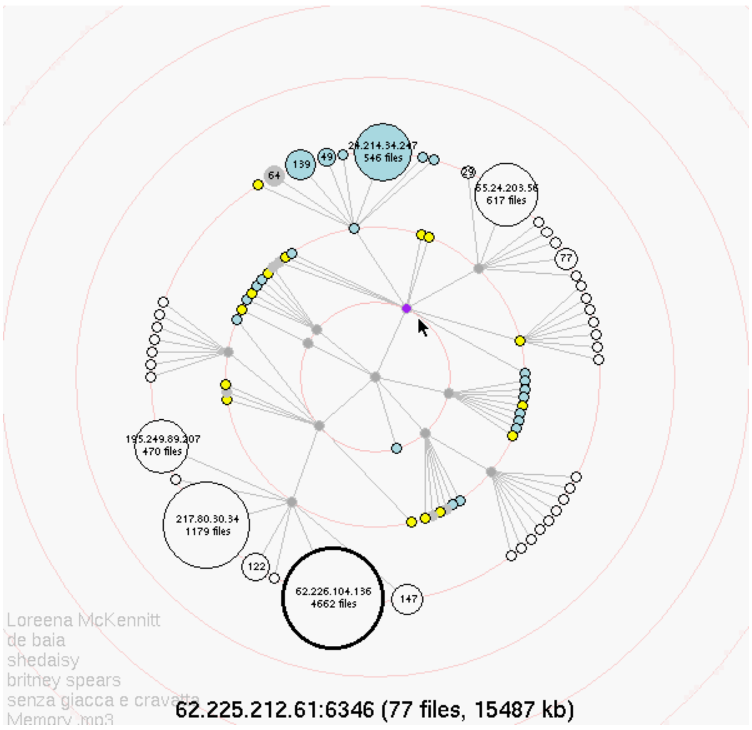
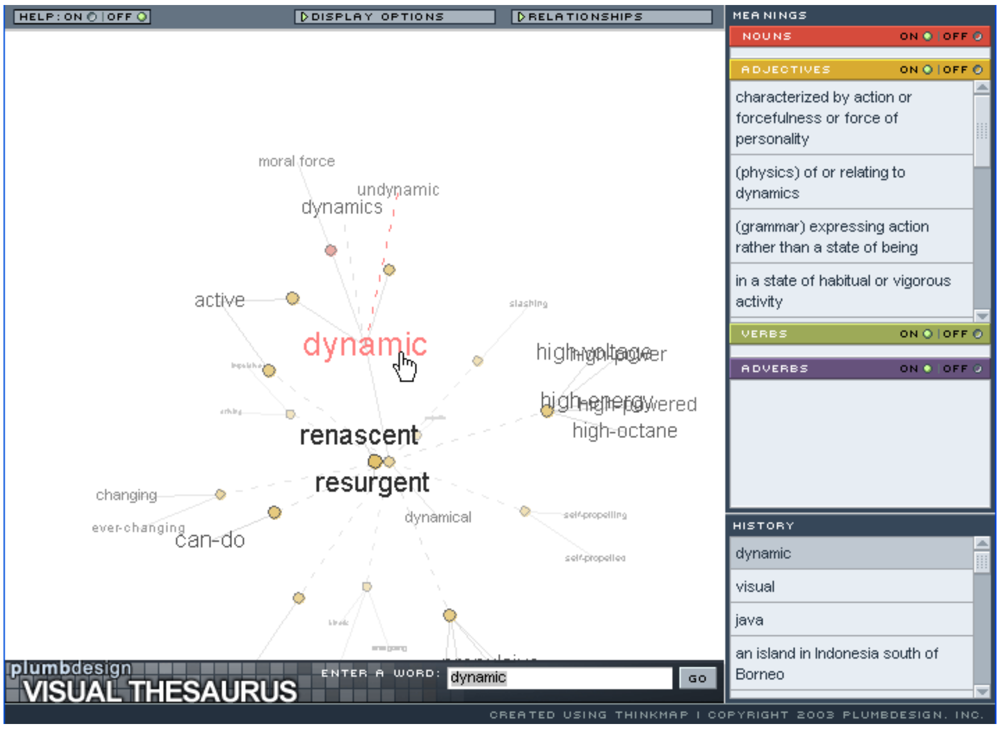


Figure 4.8: gnuTellaVision is an application of dynamic graph drawing. It allows browsing through the file sharing network Gnutella. Users can explore the network by focusing on hosts. Transitions to new layouts are smoothly animated [GTV03].



characterized by action or forcefulness or force of personality

(physics) of or relating to dynamics

(grammar) expressing action rather than a state of being

in a state of habitual or vigorous activity

dynamic

visual

java

an island in Indonesia south of Borneo

Figure 4.9: The Visual Thesaurus online edition [Vis03]. The application uses dynamic graph drawing to visualise lexical references between english words. It shows the users words which are related to the one chosen and allows navigation within the semantic network of WordNet [Wor03].

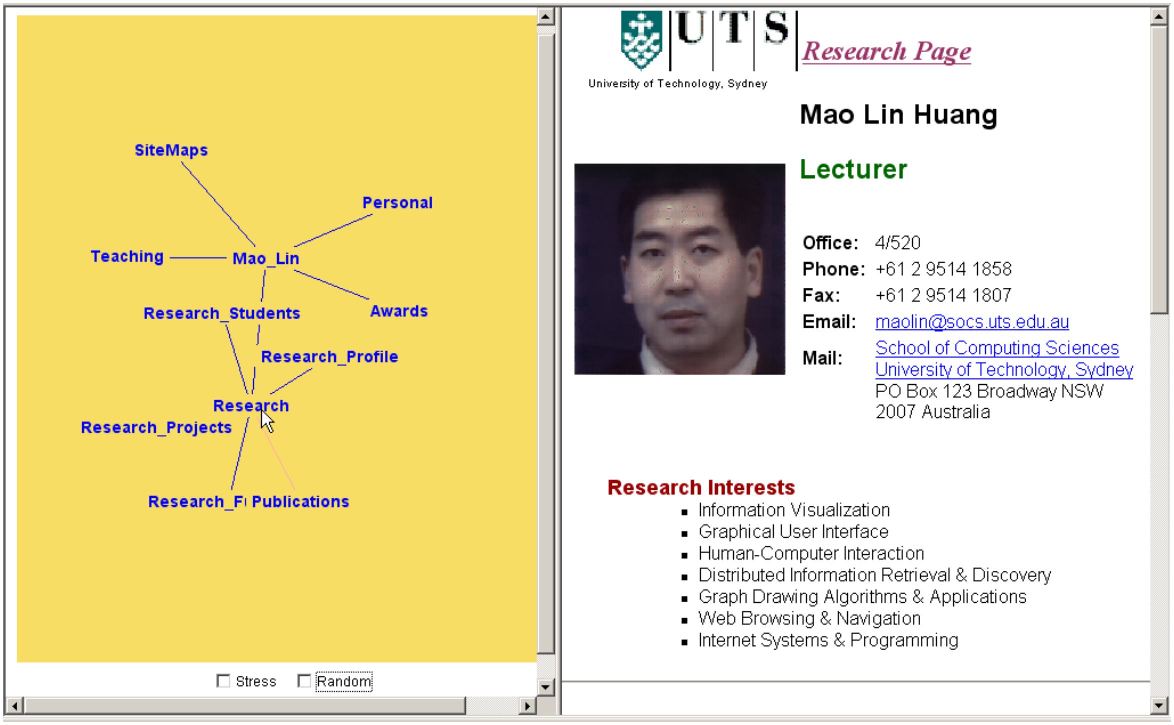
be displayed. These words are placed by force-directed placement around the focused word. Nouns, adjectives, verbs, and adverbs are assigned different colors. By moving the mouse over a word an explanation can be requested.

The tool can be used as a Java online edition [Vis03] (see Figure 4.9) or as commercial desktop edition. The desktop edition is available for several *Windows* operation systems and *Macintosh OS*.

4.4.3 Navigating Product Catalogues

The tool *OFDAV* (Online Force-Directed Animated Visualization) is based on a dynamic spring algorithm presented in [HEW98]. The idea of the tool is to present a small subset of a large hierarchy at any one time. So it does not need to know the entire graph and can display the focus, its neighbourhood, and nodes which have been already visited to help in backtracking. It uses smooth animation and fading.

Since the technique can handle large hierarchies, it can be used in various fields. In [HZ02] it is introduced for navigating product catalogues of online shops. The user can navigate through the hierarchy and select products which are displayed in a second frame. At [Lin03] it is used as a navigation tool for the author's homepage (see Figure 4.10).



Mao Lin Huang

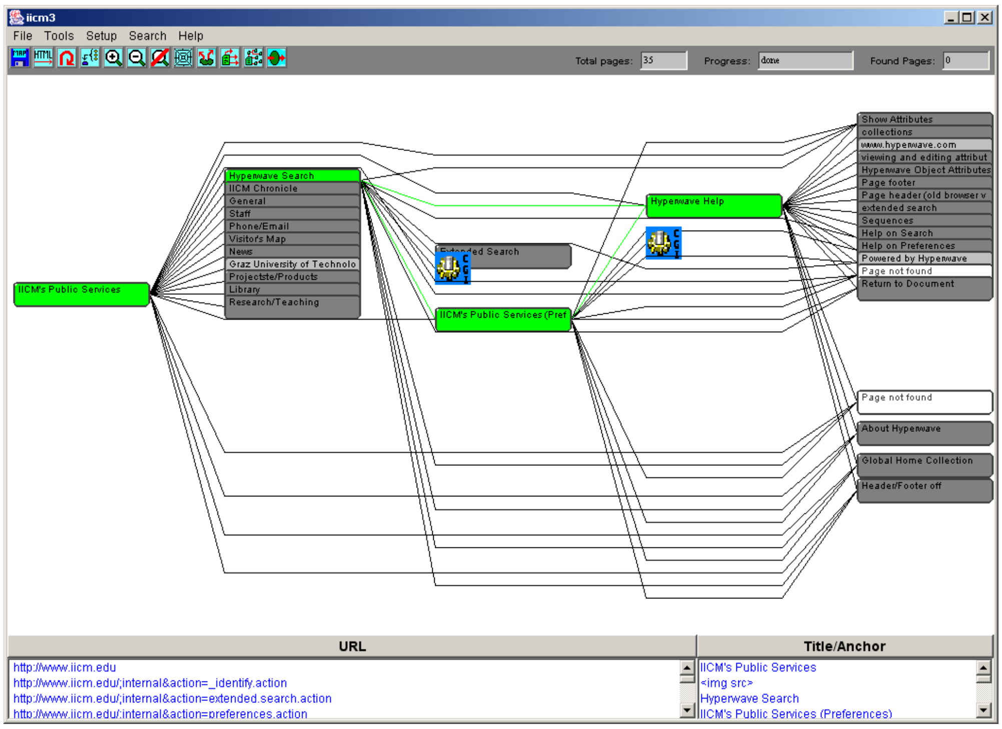
Lecturer

Office:	4/520
Phone:	+61 2 9514 1858
Fax:	+61 2 9514 1807
Email:	maolin@socs.uts.edu.au
Mail:	School of Computing Sciences University of Technology, Sydney PO Box 123 Broadway NSW 2007 Australia

Research Interests

- Information Visualization
- Graphical User Interface
- Human-Computer Interaction
- Distributed Information Retrieval & Discovery
- Graph Drawing Algorithms & Applications
- Web Browsing & Navigation
- Internet Systems & Programming

Figure 4.10: OFDAV (Online Force-Directed Animated Visualization) uses a modified spring algorithm to help navigate through product catalogues of online shops. As can be seen here, it can also be used as navigation help for home pages. The right frame displays the sites selected in the left-hand graph frame [Lin03].



URL	Title/Anchor
http://www.iicm.edu	IICM's Public Services
http://www.iicm.edu/;internal&action=_identify.action	<img src>
http://www.iicm.edu/;internal&action=extended.search.action	Hyperwave Search
http://www.iicm.edu/;internal&action=preferences.action	IICM's Public Services (Preferences)

Figure 4.11: Ptolomaeus creates a map of a certain subset of the internet starting at a site defined by the user. This figure shows a sub-map of the IICM institute homepage at Graz University of Technology [Pto03].

4.4.4 Ptolomaeus - The Web Cartographer

Ptolomaeus uses a robot for investigating the web. The user can choose a start site, a search depth and define certain filters. After the robot has finished the structure is represented by a graph displaying the web sites as nodes and links as edges. Then the user can explore the graph by selecting nodes, zooming and panning. Figure 4.11 shows part of the IICM homepage at Graz University of Technology visualised as a graph using Ptolomaeus.

The tool was developed to avoid the "lost in the hyperspace" [BLV98] syndrome. This describes the problem of finding information in a network, having multiple and broken links, without knowing its structure. Ptolomaeus 2.1 is available as Java application for *Windows*, *Solaris* and *Linux* at [Pto03].

4.4.5 PersonalBrain

PersonalBrain is a dynamic visual representation of the structure of a particular web site. Like the tools described above it represents the web site structure as a graph which can be used for navigation. While the graph is shown in one frame, selected sites are displayed in another frame. In contrast to other tools it does not use any kind of standard graph drawing technique neither for the graph layout nor for transitions. Transitions are animated very fast so that it is difficult to track the movements of the nodes except the one which is actually selected.

Figure 4.12 shows a visualisation of the web site structure of cristella.com [cri03] generated using the tool *SiteBrain*, which was the old name for PersonalBrain. The current version 2.01 of

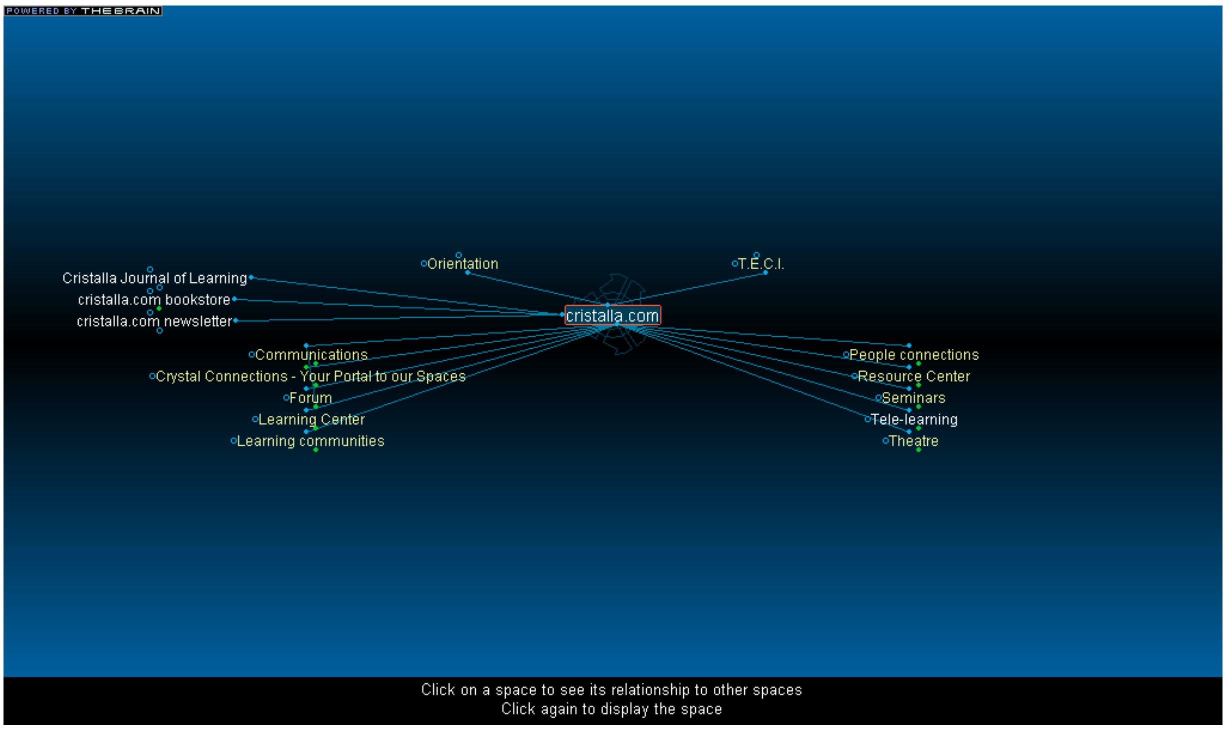


Figure 4.12: Visualisation of the web site structure of cristella.com with SiteBrain available at [cri03]. SiteBrain is now known as PersonalBrain [Per03] and can be used as navigation help for web sites.

PersonalBrain can be downloaded at [Per03]. A trial version is available for Windows.

Chapter 5

The JMFGraph Framework

This chapter introduces the *JMFGraph* graph drawing framework into which the new algorithms and functionality presented in this thesis was integrated. The *Java Modular Framework for Graph-Drawing* (JMFGraph) was initially developed by Alexander Stedile in his Master's thesis [Ste01].

The original version was implemented in Java 1.3. It uses SWING for the graphical user interface and does not include any third party libraries. Since it is implemented in Java it uses the object-oriented programming approach and consists of several modules. The most important modules include a module providing several static graph drawing algorithms, a module for reading graphs from input streams, and a module for changing the representations of nodes and edges. Additionally, a graphical user interface allows interaction and manipulation of graphs.

The new version of the software is now implemented in Java 1.4.2. New modules providing dynamic graph drawing and export were added. To try out a new concentric graph layout, the module for graph drawing algorithms was extended. The graph reading module was extended to support the GraphML file format. Finally, the representation module was extended to draw edges as Lagrange curves.

The JMFGraph framework is divided into modules which are implemented as Java packages.

5.1 Mediator

The `Mediator` is the central unit in the framework. It communicates with nearly all other modules and keeps track of them. It receives and forwards requests coming from one module to another.

To illustrate how the `Mediator` works, consider a request made by the user to change the layout algorithm. The `Graphical User Interface` sends the id of the chosen algorithm to the `Mediator`, which forwards the request to the `Layout Algorithms` module. The `Layout Algorithms` module creates a new instance of the preferred algorithm and hands it back to the `Mediator`. The `Mediator` then forwards the algorithm to the `Drawing` module which updates the drawing. It also sends requests for new motion algorithms to the `Motion Algorithms` module and forwards the algorithm to the `Drawing` module.

The `Mediator`, which is implemented as a single Java class, also takes care that every dialogue which has been opened using the menu of the user interface is only instantiated once. This is done by providing singleton instances.

This single class module has not changed substantially from the old version to the new. Only methods for handling the new requests have been added.

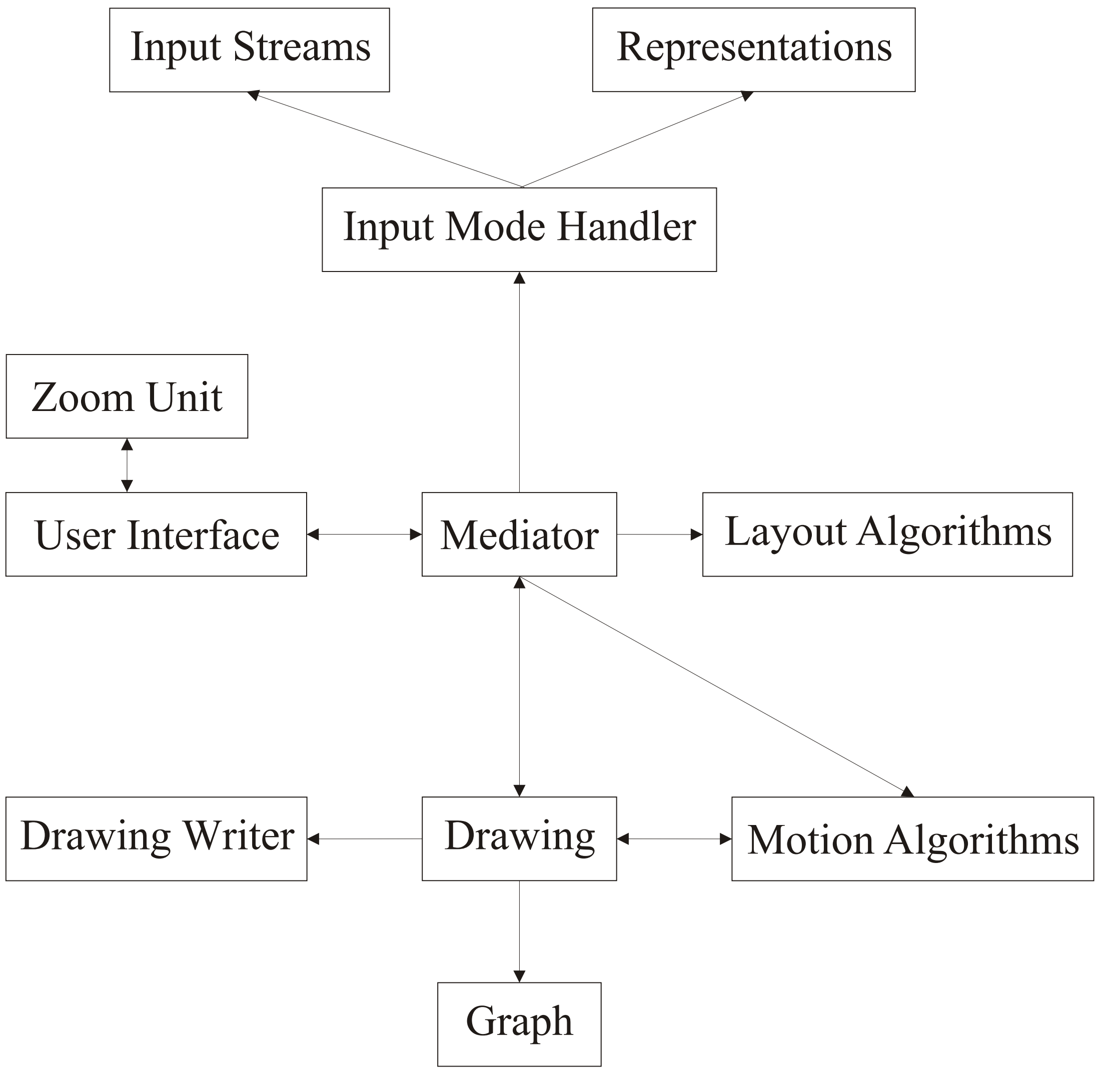


Figure 5.1: The modular architecture of the JMFGraph framework.

5.2 Graphical User Interface

The `Graphical User Interface (GUI)` provides a range of possibilities for the user to interact with the program. It consists of a frame including an area displaying the graph, an options panel, a menu bar and a status bar. The displayed graph can be inspected by using zooming and panning. If the user requests zooming, the `GUI` communicates with a special zoom unit, which handles the scaling factors. The options panel and the menu bar provide functionality and dialogues for manipulating the graph. Besides opening graphs and exporting graph layouts, the user can select layout algorithms, motion algorithms, node and edge representations and graph orientation. The dialogues can be used to change properties of layout and motion algorithms, drawing depth and font style. A special dialogue for displaying the layout created by every step of the layout algorithms is available. All functions can be activated using the menu, keyboard shortcuts, or the mouse. Keyboard strokes and mouse events are handled by this module. For everything else, the `GUI` needs the `Mediator` to perform its tasks.

All dialogues and menu items concerning the properties of the layout algorithms, motion algorithms and changing the drawing depth did not exist in the old version of the module. Also the status bar, the possibility for exporting the graph layout and for choosing the file format of the graph to open have been added. Finally, the mouse wheel support, which can be used for zooming and changing the angles of the concentric layout algorithm is new.

5.3 Zoom Unit

As has already been said above, the `Zoom Unit` is needed by the `Graphical User Interface` if the user wants to change the zoom factor of the displayed drawing. This can be done by calling functions of a scaler which provides several scaling factors. It stores the keys which can be used for zooming in and out, and the modifier keys to change the factor.

5.4 Input Mode Handler

The `Input Mode Handler` module keeps track of different modes for reading graphs. Currently there is a demo mode, which uses a dummy input stream reading a graph from a Java class, and a file mode. The file mode supports two different graph file formats. The `Input Mode Handler` installs the mode requested by the `Mediator`. So, if the `Mediator` wants to access an input source, it is connected to the source which had been installed previously.

The `Input Mode Handler` also installs representations for nodes and edges, communicating with the `Representations` module. Generally, the `Input Mode Handler` receives requests from the Mediator and sends requests onto the modules `Input Streams` and `Representations`.

In the old version file mode was called dot mode, because only the dot file format was supported. When adding the second graph file format this was changed in accordance with Chapter 8 of [Ste01].

5.5 Input Streams

The `Input Streams` module provides access to input sources containing graph descriptions. It includes several parsers for different sources. In demo mode it parses graphs, which are included in the program hardcoded. In file mode it offers two specific parsers at the moment.

The new parser for the GraphML file format, which was described in Section 4.3.6, only supports the most important tags. Node tags and edge tags, but no data tags are handled. Since the framework only implements layout algorithms for directed graphs, all edges are stored as directed edges.

5.6 Representations

The `Representations` module contains graphical representations for nodes and edges. There are two representations for nodes available at the moment. The first draws nodes as dots with labels directly above. The second adds a background to the node label and makes the label clickable. It also provides a special alignment when using the concentric layout algorithm. It places the label so that the focused node, the node and its label lie on the same straight line.

The module includes two edge representations. The first draws edges as simple polylines with an arrow head at the end to show the direction. The second represents edges as `Lagrange` curves controlled by the edges' dummy nodes. A Lagrange curve is a curved line which runs through all control points.

The original version included the node representation with the simple non-clickable label. As edge representation, only the polyline representation was provided.

5.7 Layout Algorithms

The `Layout Algorithms` module provides static graph drawing layout algorithms. The module includes a factory, which hands over the algorithm requested by the `Mediator`. The algorithms are then used to compute the node positions in the drawing plane.

The current implementation includes three different Sugiyama-style layout algorithms following the approach described in Section 2.3.3. The first is a standard static algorithm. The second was invented by the developer of the original version of the program and is described in his thesis [Ste01]. The algorithm computes the layout around a focused node selected by the user. In addition to these two algorithms, already included in the original version, a third new algorithm is available. It uses the concentric drawing approach and is described in Chapter 7.

5.8 Drawing

The `Drawing` module is responsible for visualising the graph. If the `Mediator` sends a request for a drawing update, the `Drawing` module applies the layout algorithm and the node and edge representation to the graph and draws it. This is done by calling the paint methods of every node and edge object in the `Graph` module. As a new feature, this module can also send a request to the `Drawing Writer` module to hand over an output stream. Then it can call the write methods instead of the paint methods to write the layout to an output stream.

For dynamic graph drawing this module communicates with the new `Motion Algorithms` module. During a transition the `Drawing` module hands over responsibility for calculating the intermediate node positions and the draw command to the `Motion Algorithms` module.

Besides the methods needed for the communication with the `Motion Algorithms` module and those for writing the drawing, the `Drawing` module also has new features for controlling the drawing depth and painting the new concentric layout. One of those features is for example a method used for drawing the concentric circles in the background of the new concentric layout.

5.9 Motion Algorithms

This module is responsible for visualising a transition between two graph layouts. Therefore it provides several algorithms and sub-algorithms similar to the static layout module. The `Mediator` requests a certain motion algorithm, which is created and then applied to the `Drawing` module, which uses it.

Since the dynamic graph drawing approach is the main part of the new version of the framework, this module is totally new. It includes a factory for creating the algorithms and one dynamic graph drawing algorithm which is described in the next chapter.

5.10 Drawing Writer

The `Drawing Writer` module is responsible for writing out the layout of a graph as it is displayed on the screen to a file. It consists of a factory creating output streams and writers implementing a certain file format. At the moment there is one writer, for the SVG (Scalable Vector Graphics) [SVG04] file format. A writer provides several methods for writing certain graphical shapes.

Exporting a graph layout is also a new feature and this module is new to the JMFGraph framework.

5.11 Graph

The `Graph` module is responsible for storing the actual graph. It includes a data structure holding the graph and provides several functions like methods for inserting nodes and edges or checking graph properties. The only function which has been added to this module is a method concerning the drawing depth of the graph. The method copies the subset of the graph needed for a given drawing depth and a given focused node.

Chapter 6

Aesthetic Animated Transitions

This chapter describes the introduction of dynamic graph drawing into JMFGraph. The framework already included a browsing mode and a layout algorithm for a focused node. However, the transitions from one layout to the next, when the focus changes, were not animated. The user was unable to track changes from one layout to the next and needed to reassimilate each layout afresh. This can be seen in Figure 6.1.

Thus, it was decided to implement a motion algorithm, helping the user to trace the node's movements during transition. The practical experiences with this approach will be described in the next section. For a further improvement, a clustering algorithm was integrated, which assigns nodes performing similar motions to the same cluster. The results of the integration of this algorithm are also described.

6.1 Motion Algorithm

The dynamic graph drawing approach chosen for smooth transitions was first introduced in [FE02]. An overview of the algorithm's theoretical background is given in Section 3.2.2. Before integrating the approach into JMFGraph, a demo application was developed to try out the algorithm.

6.1.1 Demo Application

The demo application implements steps 2 and 3 of the algorithm. As described in Section 3.2.2 the second step moves the nodes as one rigid object. This is done by computing the affine transformation matrix which brings all nodes as close as possible to their final positions. The nodes are moved by interpolating the transformation matrix and multiplying it with the nodes' starting coordinates. Since the second step can not move all nodes to their final positions, the third step does this using linear interpolation.

The application provides a list of sample transitions. The user can choose one and watch the animation. The transitions show nodes moving from a certain start position to a final position using the approach described above. There are no edges connecting the nodes and no nodes are faded in or out, because no nodes are added or removed.

6.1.2 JMFGraph using Rigid Motion

Since the algorithm worked fine in the demo application, it was integrated into JMFGraph. Steps 1 and 4, which fade out nodes from the first layout not belonging to the second layout and fade in new

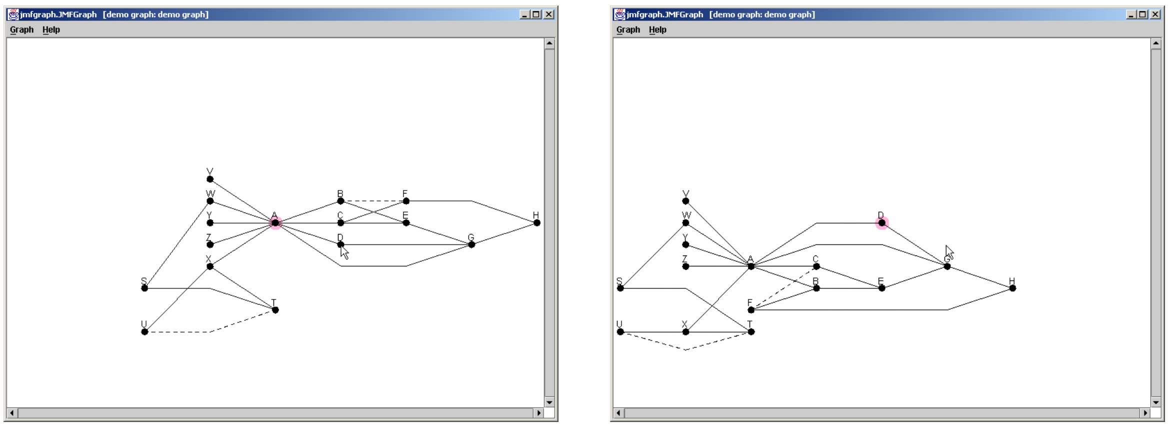
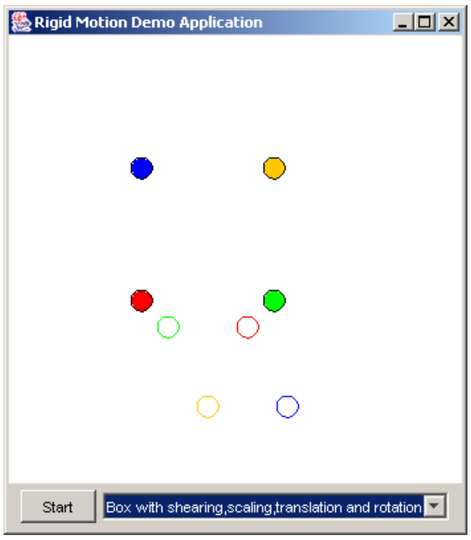


Figure 6.1: Browsing a graph with the old version of JMFGraph. Selecting another node changes the focus and modifies the layout. The transition was not animated and caused users to loose their mental map.



Box with shearing,scaling,translation and rotation

Figure 6.2: The demo application for testing the dynamic graph drawing approach of [FE02]. It provides a list box of sample transitions, from which the user can choose.

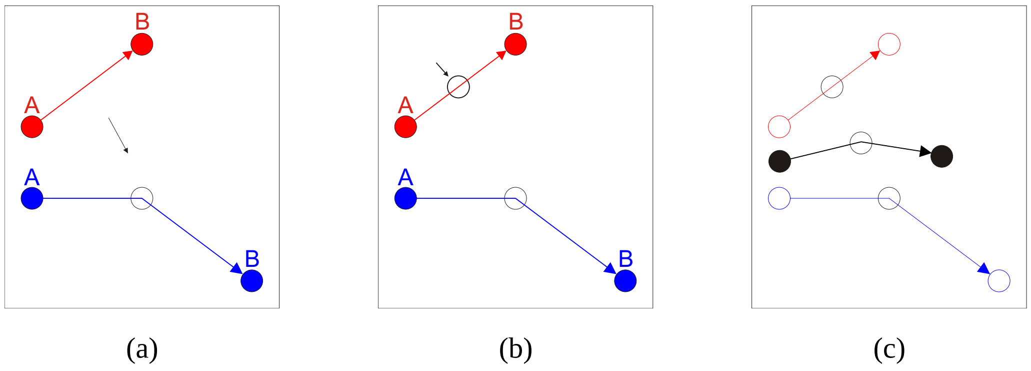


Figure 6.3: (a) Edges can have different lengths in the old (red graph) and new (blue graph) layout. (b) To make transitions possible dummy nodes have to be attached to the shorter line. (c) Then the corresponding dummy nodes can be used for the motion as well as the "normal" nodes.

nodes which were not in the first drawing, were also introduced.

The question arose as to how to animate the edges. Like nodes, edges occurring in both drawings are visible through out the animation, others are faded out or in. The problem arose that edges, which are represented by polylines, do not always have the same lengths in both drawings. So it might happen that an edge has a length of one in the first drawing and a length of two in the second drawing (see Figure 6.3).

As can be seen in Figure 6.3(b), this problem can be solved by inserting additional dummy nodes. As many dummy nodes have to be inserted, such that every dummy node in one drawing has a corresponding dummy node in the other drawing. The dummy nodes should be inserted so that they are equally distributed along the edge. All dummy nodes are then treated like "normal" nodes, and are included into the computation of the affine transformation matrix.

In Figure 6.3(b) another potential problem can be seen. In this drawing, all three nodes (including the new dummy node) lie on the same straight line. However, the computation of the affine transformation matrix requires three non-collinear points. Most of the time this edge will not be the only edge, so there will be enough non-collinear nodes. If there are many nodes, the clustering approach can help assign non-collinear nodes to the same clusters and compute their transformation matrix. However, the problem can not be avoided entirely. To solve this problem, a heuristic approach was tried, which slightly changes the nodes' coordinates. Since this can be seen in the drawing and does not work all the time, it was decided to perform simple linear interpolation in this rare case.

It is important to have the edges in the transition. However, the user should concentrate on the nodes and if the graphs are very dense, edges can cause considerable distraction. So it was decided to decrease the edge opacity during transition. The user can change this setting through the GUI.

6.2 Motion Clustering

To improve the results achieved by the rigid motion algorithm, it was decided to use a motion clustering algorithm. The k-means algorithm chosen was introduced in [FH01], and is described in Section 3.2.3.

The algorithm groups nodes performing similar motions together into clusters. The number of clusters can be set by the user. The user can choose a value between one and ten. This sets the

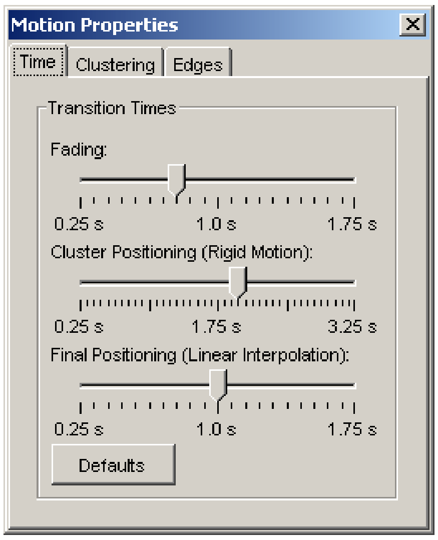


Figure 6.4: JMFGraph's Motion Properties dialogue for setting properties concerning the dynamic graph drawing algorithm.

maximum number of clusters. Since every cluster has to have at least three nodes to be able to compute the transformation matrix, the program checks that the number of clusters does not exceed the number of nodes divided by three.

No special strategy for the initial partitioning is used. Clusters are filled one after the other.

6.3 User Interface

The user can control the dynamic algorithm through the motion properties dialogue (see Figure 6.4) provided by the GUI. First the dialogue provides sliders to set the speed for fading and transitions. The user can adjust this by setting the times used to fade in and fade out nodes and edges, to perform the rigid motion, and to do the final positioning. The user can adjust the number of clusters and the opacity of the edges during transition. It is also possible to disable the motion algorithm if the graph should be browsed without any dynamic transitions.

6.4 Example of Dynamic Transition

Referring to Figure 6.1, where no motion algorithm is used for the transition, this section shows a transition using the dynamic approach. The layout is computed with a drawing depth of three, which means that all nodes having a distance of up to three from the focused node are included in the layout (see Section 7.1 for further explanations on the drawing depth).

Figure 6.5 shows the transition using the motion algorithm with one cluster. It can be seen that the affine transformation can not transform the nodes to their final positions. The whole graph is translated, rotated sheared and scaled to a position near the end position. In this case, the entire graph is animated according to the average motion over all nodes, which can achieve only very approximate positioning. The real positioning is performed by the linear interpolation step starting after picture 5 until the final layout is reached in picture 8. Picture 2 shows the fade out process of the removed nodes and edges and picture 9 shows the fade in of the new nodes and edges.

Figure 6.6 shows the same transition using a maximum of four clusters. The rigid motion step can transform the nodes to their final positions. No motion is performed by the linear interpolation step. Pictures 2 and 7 show the fading steps.

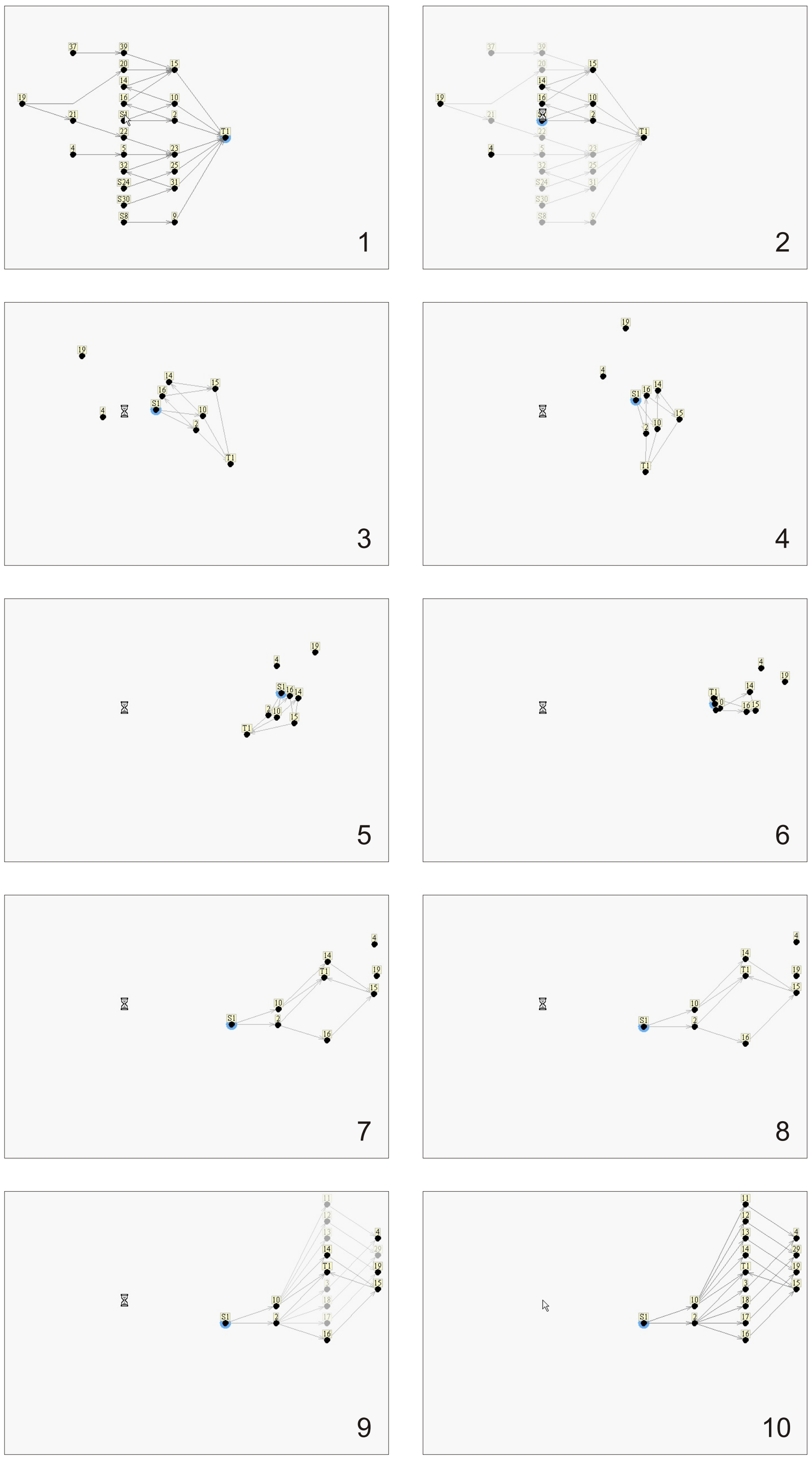


Figure 6.5: Smooth transition using one cluster for the rigid motion step. Since motion is averaged over all nodes only a very approximate positioning for each individual node can be achieved (pictures 3-5). The final positioning (pictures 6-8) is performed in the linear interpolation step. Pictures 2 and 9 show the fading process.

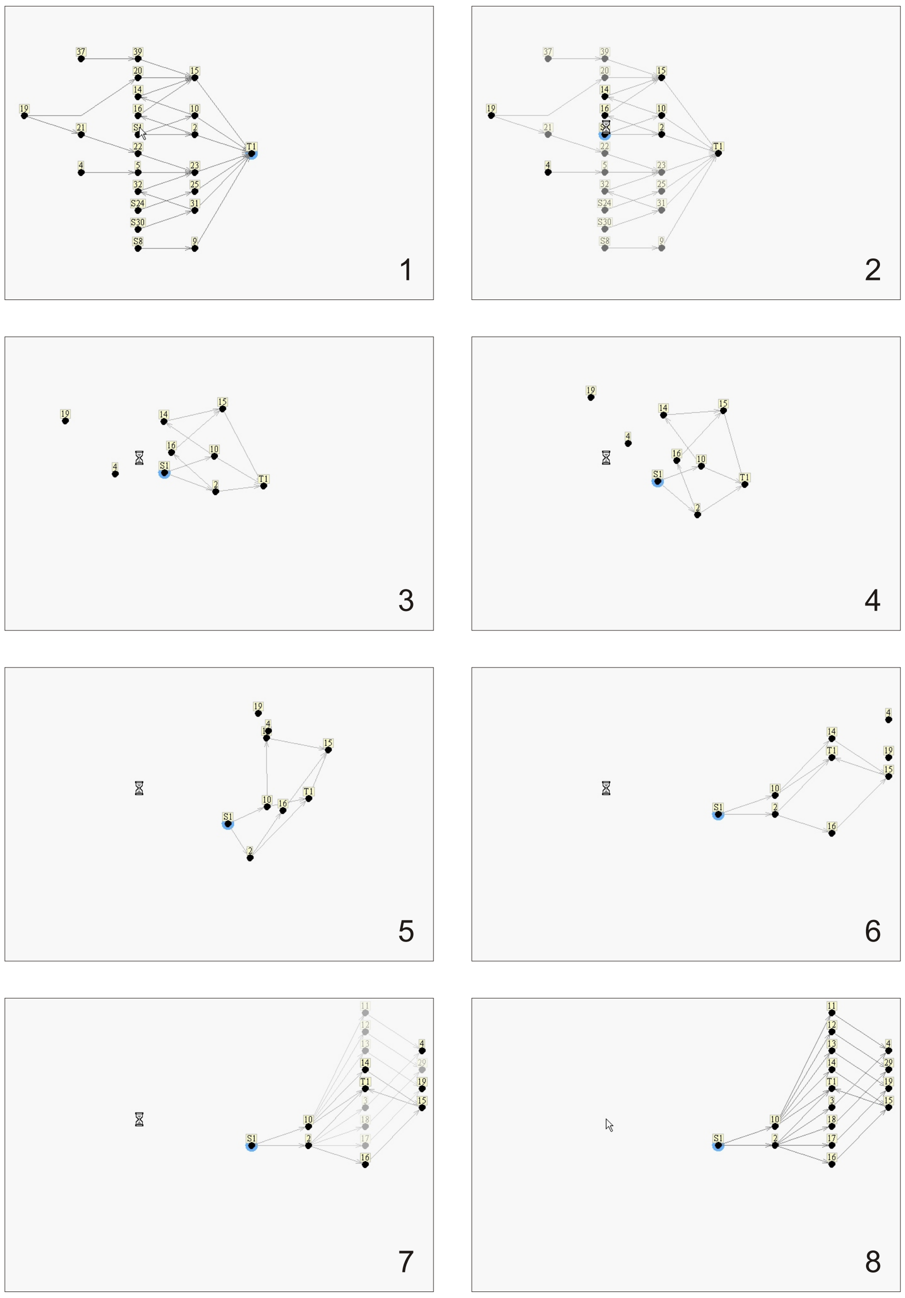


Figure 6.6: Smooth transition using four clusters for the rigid motion step (pictures 3-6). The nodes can be transformed exactly to their final positions. No linear interpolation has to be done. Pictures 2 and 7 show the fading process.

Chapter 7

Concentric Sugiyama-Style Layout

This chapter describes a new Sugiyama-style graph drawing layout algorithm. It uses a drawing convention placing nodes on concentric lines and representing edges either as polylines or curved lines. Since JMFGraph supports dynamic browsing, a suitable concentric drawing algorithm was also developed.

In the literature, concentric layout algorithms are mostly used for drawing trees (see Chapter 2). The concentric algorithms presented for (directed) graphs either adapt the algorithms for trees [YFDH01] or follow other strategies using the full angle of 360 degrees ([Car80],[RM88]). The algorithm needed for JMFGraph has to split the full angle into two sectors because incoming nodes should be displayed in one half and outgoing nodes in the other half.

7.1 Drawing Depth

In JMFGraph, the dynamic graph drawing approach is used to help the user to explore very large graphs. This is done by showing the user only a subset of the entire graph around the focussed node and animating the transitions from one subset to the next when the focus changes. The user can control the size of the shown subset by setting a drawing depth. Since the framework works with directed graphs, there is an incoming and an outgoing drawing depth. The drawing depth defines the maximum distance from the focused node that a certain node can have to be included in the drawn subgraph. Assuming an outgoing drawing depth of two, all nodes that can be reached by following two outgoing edges starting at the focused node would be included in the drawing.

The algorithm for computing the subset of nodes to be included works like a breath first search algorithm. First, a breadth first search is performed in the direction of incoming edges starting at the focused node. All nodes, which can be reached by following the incoming edges of the nodes found in the previous iteration step are included. This is done until the maximum incoming drawing depth is reached. To avoid loops, nodes which have already been visited are stored. The same is done for the outgoing drawing depth, by following outgoing edges. Since the graph can be cyclic, the same nodes can be found by both the incoming and the outgoing search procedure. When the search for nodes has finished, it has to be decided which edges should be visualised. All edges of both directions belonging to the full graph and connecting two nodes, which both belong to the computed subset are shown.

7.2 Concentric Sugiyama-Style Layout Algorithm

The new concentric layout algorithm follows the hierarchical graph drawing approach described in Section 2.3.3. It is split into the four steps of the original Sugiyama algorithm presented in [STT81]: layering with integrated cycle removal, crossing reduction, x-coordinate assignment and a fourth step mapping the layout to the drawing space.

It computes the layout with respect to a focused node selected by the user, which is placed in the middle of the drawing. The other nodes are aligned on concentric circles around the focus node. Incoming nodes are placed to the left of the focus node and outgoing nodes to the right.

7.2.1 Layering

The layering step assigns every node to a certain layer. The focused node is placed in layer L_f, with $1 \leq f \leq n$ and n defining the maximum number of layers. Incoming nodes are assigned to layer L_i with $1 \leq i < f$ and the outgoing nodes are placed in layer L_o with $f < o \leq n$.

It first has to be decided which nodes belong to the incoming half and which nodes belong to the outgoing half. In principle, nodes which can be reached by following outgoing edges starting at the focus node belong to the outgoing set and nodes, which can be reached by following incoming edges belong to the incoming set. Since the graph can be cyclic, some nodes can be reached by incoming and outgoing edges. To solve this problem, it was decided to use a breadth first searching algorithm. The search starts at the focused node and proceeds in both directions. A node is assigned to the search set, which finds it first.

Once the set of nodes is split, a layering algorithm can be applied to both sides of the focused node. Two layering approaches are implemented. They are both called hourglass algorithms, because the shape of the layout with the focused node in the middle and the incoming and outgoing nodes at the sides resembles an hourglass.

Hourglass Longest Path Layering Algorithm

The *Hourglass Longest Path Layering Algorithm* is the longest path layering algorithm introduced in [ES90], applied to both incoming and outgoing set of nodes. The original longest path layering algorithm, already implemented in the original version of JMFGraph, works as follows. It assigns all *sinks*, which are nodes with no outgoing edges, to the lowest layer L_1. All other nodes v are assigned to the layer L_m, where m is the length of the longest path between v and a sink [ES90]. On condition that no edges between nodes in the same layer are allowed, this algorithm assigns every node to its lowest possible layer.

The algorithm is applied to the two sets of nodes as follows. For the incoming node set the sinks are assigned to layer L_{f-1}. In this case, sinks are nodes with no outgoing edges except those going to the focused node or to the outgoing node set. All other incoming nodes are assigned to layer L_{f-1-m} with m being the length of the longest path between the incoming nodes and their sinks. So, for the incoming node set the algorithm works as the original one, but with reversed indices. For the outgoing node set, nodes with no incoming edges (except those coming from the focused node or the incoming node set), are assigned to layer L_{f+1}. All other nodes are placed in layer L_{f+1+m} where m is the length of the longest path between the nodes and the *sources*. Figure 7.1(a) shows the hourglass longest path layering algorithm applied to a sample graph.

A similar algorithm was used by Jürgen Schipflinger in his Master's thesis [Sch98]. He used a Sugiyama-style algorithm to display the incoming and outgoing links of a selected document in the

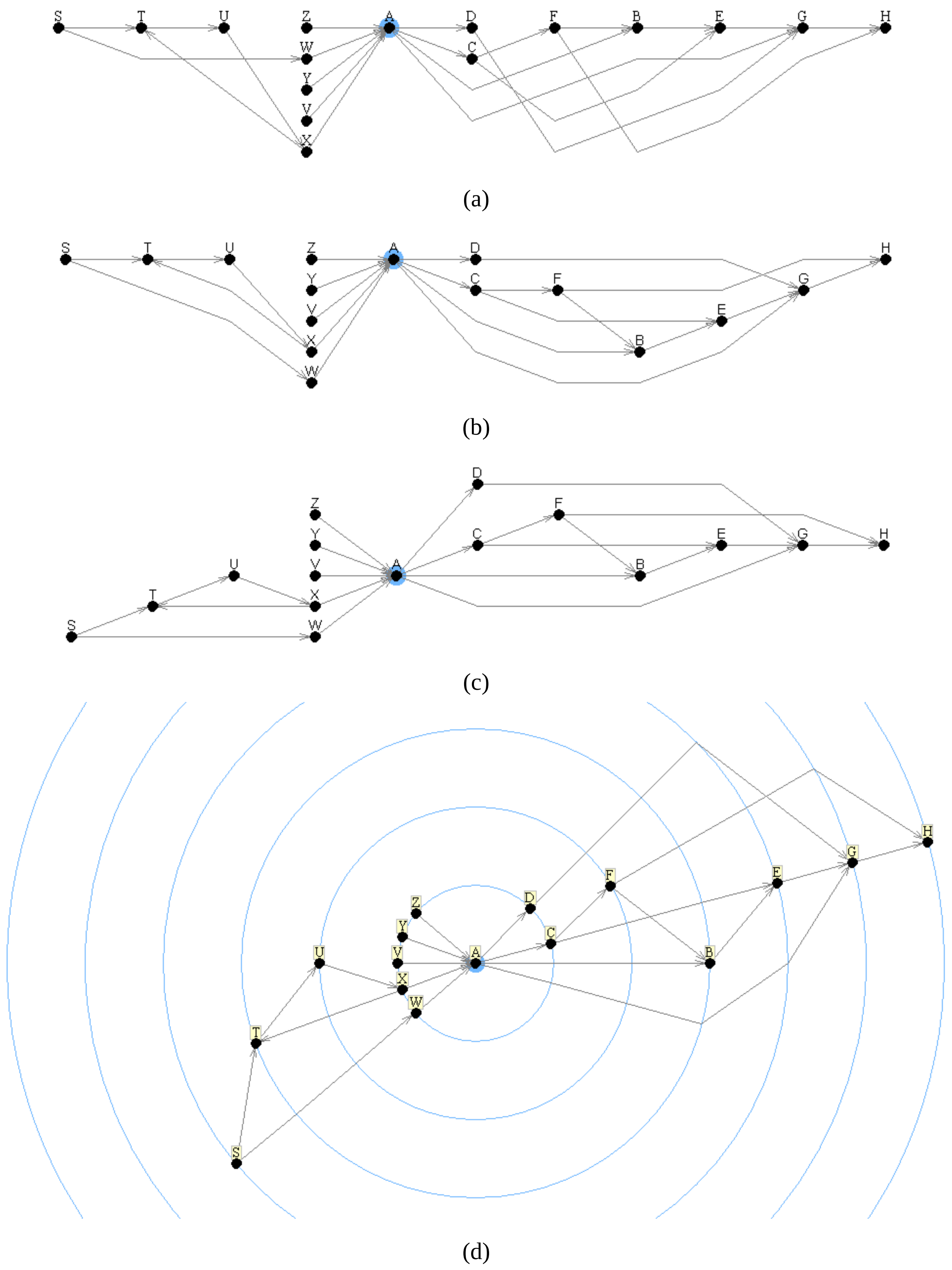
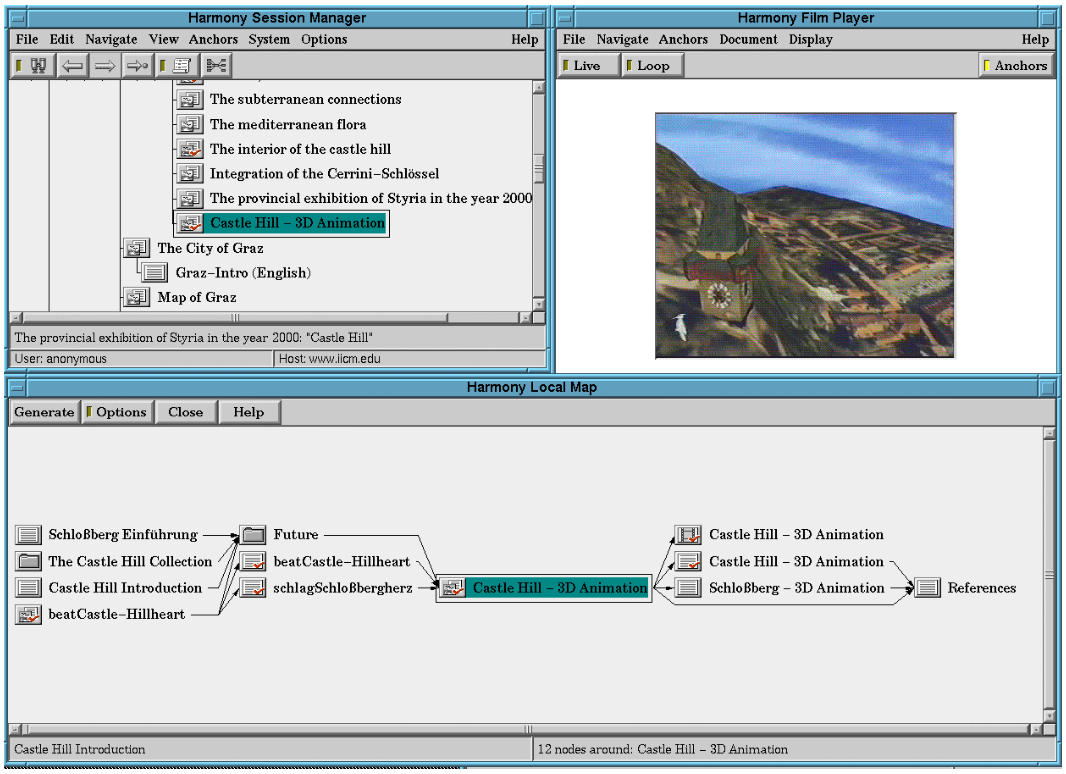


Figure 7.1: Steps of the concentric Sugiyama-style layout algorithm with hourglass longest path layering. The images show an example graph after every step of the layout algorithm. (a) layering (hourglass longest path), (b) crossing reduction (global sifting), (c) x-coordinate assignment (priority layout method), (d) final concentric Sugiyama-style layout.


Harmony Session Manager
File Edit Navigate View Anchors System Options Help
The subterranean connections
The mediterranean flora
The interior of the castle hill
Integration of the Cerrini–Schlössel
The provincial exhibition of Styria in the year 2000
Castle Hill – 3D Animation
The City of Graz
Graz–Intro (English)
Map of Graz
Harmony Film Player
File Navigate Anchors Document Display Help
Live
Loop
Anchors
Harmony Local Map
Generate
Options
Close
Help
Schloßberg Einführung
The Castle Hill Collection
Castle Hill Introduction
beatCastle–Hillheart
Future
beatCastle–Hillheart
schlagSchloßbergherz
Castle Hill – 3D Animation
Castle Hill – 3D Animation
Castle Hill – 3D Animation
Schloßberg – 3D Animation
References

The provincial exhibition of Styria in the year 2000: "Castle Hill"

User: anonymous Host: www.iicm.edu

Castle Hill Introduction

12 nodes around: Castle Hill – 3D Animation

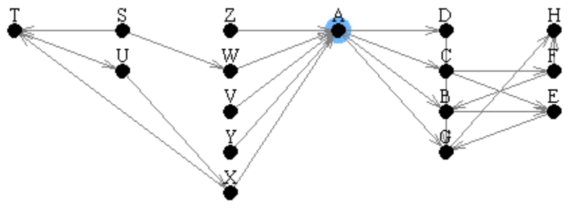
Figure 7.2: The *Harmony Local Map* introduced by Jürgen Schipflinger in his Master's thesis [Sch98]. An hourglass shaped Sugiyama-style layout is used to display incoming and outgoing links of a selected document.

Harmony Local Map, which is part of the graphical interface to the *Hyperwave* information server. Figure 7.2 shows the hourglass shaped layout.

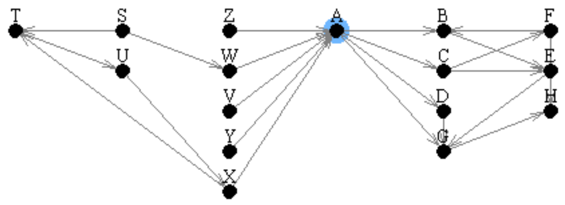
Hourglass Breath First Layering Algorithm

Alternatively to the longest path layering algorithm, the *hourglass breadth first layering algorithm* was implemented. It is based on the idea that every node should be on the circle corresponding to the depth of the node. For example, if a node has a distance of two from the focused node (depth two), then it would be assigned to the second circle. The number of circles needed is never higher than the drawing depth, which is not the case when using longest path layering.

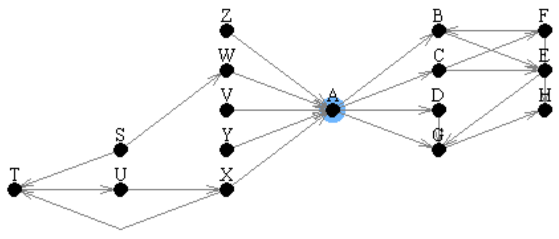
The algorithm uses again a breadth first searching algorithm. It starts at the focused node and proceeds in both directions (incoming and outgoing). The nodes are assigned to the layer with the number of the iteration step where the node was found for the first time. This can lead to edges within the same layer, as can be seen in Figure 7.3(a). When a "traditional" Sugiyama-style algorithm with straight line layers is used this should be avoided, because several edges within the same layer could cover each other and make unclear which edge connects which nodes. Using concentric layers this case causes no problems as can be seen in Figure 7.3(d), which shows the final layout. There are also more backward edges than in the layout generated using the longest path algorithm. However, it brings the nodes to the layers corresponding to their drawing depth, which corresponds better to users' expectations.



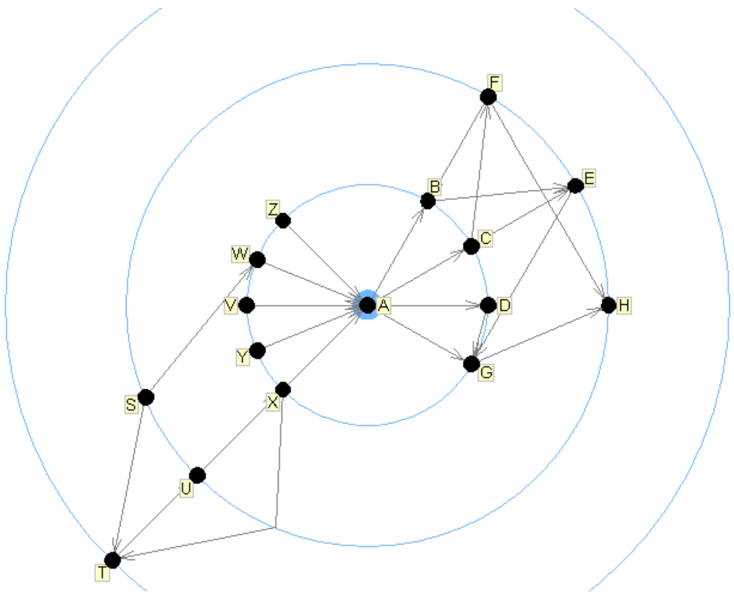
(a)



(b)



(c)



(d)

Figure 7.3: Steps of the concentric Sugiyama-style layout algorithm with hourglass breadth first layering. The images show an example graph after every step of the layout algorithm. (a) layering (hourglass breadth first), (b) crossing reduction (global sifting), (c) x-coordinate assignment (priority layout method), (d) final concentric Sugiyama-style layout.

7.2.2 Crossing Reduction

The crossing reduction step computes a permutation of the nodes in every layer which minimises the number of crossing edges. To achieve this *global sifting* as presented in [MSM99] is used. This algorithm was already implemented in the original version of JMFGraph and is also used for the other two Sugiyama-style algorithms.

For one node, sifting means moving the node within its layer, by repeatedly swapping it with its neighbours to the leftmost position then to the rightmost position. Storing the number of edge crossings at each position, the node can be set to this position with least crossings when sifting is finished. Globally, for all nodes, the algorithm works as follows. A list of all nodes $l = (l_1, ..., l_n)$ with $l_i \varepsilon V$ of the graph $G = (V, E)$ is made. The list is ordered descending by the node's indegree. Then all nodes l_i, i from 1 to n, are sifted in their layer. If this process does not reduce the number of edge crossings a fail counter is incremented and the list is reversed before the second trial. Otherwise the second trial is made with the same order. After the second loop the list is reversed in any case and the whole process is repeated until the number of fails exceeds a maximum number of fails [MSM99].

Figures 7.1(b) and 7.3(b) show the layout after the crossing reduction step has been applied. It can be seen that the algorithm can remove more crossings if it is applied after longest path layering. On the other hand, the layout generated using this layering needs more layers than the one obtained by breadth first layering.

7.2.3 X-Coordinate Assignment

As described in Section 2.3.3 the idea of the x-coordinate assignment step is to align the nodes in their layers in a way that bends are reduced and symmetry is enhanced, without changing the order computed in the crossing reduction step. Like in the layering step an algorithm invented for a "traditional" Sugiyama approach is applied here to both sides of the hourglass individually. The used algorithm has also already been implemented in the first version of JMFGraph. It is the method used by Sugiyama for the original version of his algorithm. The approach is called *priority layout method* and is introduced in [STT81].

The original version of the heuristic performs three sweeps through the layers. It starts with a DOWN sweep from layer L_2 to L_n, then goes UP again from L_n to L_1 and ends with another DOWN sweep from L_t to L_n with $(2 \leq t \leq n)$. At every layer the nodes are ordered by priorities. Dummy nodes are assigned the highest priority. All other nodes are assigned a priority corresponding to the number of neighbours the node has in the upper (for DOWN sweeps) or in the lower (for UP sweeps) layer. Then, starting from highest to lowest priority, the nodes are assigned x-coordinates moving the node as close as possible to the barycentre of the positions of its neighbours in the upper (lower) layer. This is done without changing the order of the nodes [STT81].

This approach is applied to the concentric layout method as follows. The first sweep process starts at layer L_{f-1} goes to L_1 and back to L_f. Since the hourglass layout is a tree structure having its root at the focused node, it is better not to sweep back to L_1. This would destroy the tree structure obtained by the last sweep. The same is done for the outgoing nodes by sweeping from L_{f-1} to L_n and going back to L_f. Since the layer of the focused node is changed by this last sweep, the whole right half has to be shifted so that the focused node returns to the position it was in before the approach was applied to the right half. It is clear that the x-coordinate assignment is in fact a y-coordinate assignment in this case because the concentric layout is aligned horizontally and not vertically like the original Sugiyama layout.

Figure 7.1(c) shows that the step really reduces the number of bends in the polylines. It also brings symmetry into the layout. This can also be seen in Figure 7.3(c) where it moves the focused node to a relatively central vertical position.

7.2.4 Final Concentric Layout

The drawing is finished by removing the dummy nodes and mapping the node positions obtained by the last step to corresponding coordinates in two dimensional space. In a horizontal drawing the node's layer becomes its x-coordinate and the position in the layer the y-coordinate, both with a certain scale and offset. For a concentric layout another mapping algorithm, computing the angles and distances from the focused node, is needed.

The algorithm maps the layers to concentric circles. This is done by putting the focused node, which is in layer L_f, in the middle. Then layer L_{f-1} is mapped to the left half of the smallest circle C_1 and layer L_{f+1} to its right half, L_{f-2} and L_{f+2} to C_2 and so on.

When using a traditional mapping of a layer to a straight vertical line space is infinite; that is not the case when mapping to a circle. A first approach to solve this would be to divide the 180^o by the number of nodes plus one in every layer. So if there would be two nodes in a layer, the first node would be at angle 60^o and the second at 120^o. The effect of doing this for every layer can be seen in Figure 7.4(b). Since the number of nodes in every layer can be different, nodes which have been assigned to the same positions in two different layers by the previous step (see Figure 7.4(a)) now lie at different angles. Symmetry and bend reduction are lost.

The solution can be seen in Figure 7.4(c). First, the highest (p_{max}) and the lowest (p_{min}) position of all layers $L_{i<f}$ is determined. Then the angle α is computed such that:

$$\alpha = \frac{180^o}{|p_{max} - p_{min}| + 2}$$

A node lying at position p_i is always drawn at angle $\alpha \cdot (p_{max} - p_i + 1)$ for all layers $L_{i<f}$. Nodes keeping the same positions in different layers lie on the same radial line. So symmetry and straight lines are preserved. The same is done for layers $L_{i>f}$ using angle β, which can be different than α. This is all true if the radius of every circle C_i is $r_{C_1} \cdot i$.

Sometimes the user wants to allocate more space to the incoming sector than to the outgoing sector or the other way around. In this case, the 180^o is simply replaced by the chosen angle. For example, if there are very few incoming nodes and many outgoing nodes, 90^o and 270^o could be used. Or if there are few nodes on both sides 90^o and 90^o might be chosen.

It happens often that edges extend from one half into the other. Then they are routed through the focus layer L_f and the focus node is not the only node in its layer. Dummy nodes are placed above and below the focus node in the focus layer to accomodate edges which cross from the incoming to outgoing sectors or vice versa. The maximum of the number of dummy nodes below and above the focus node has to be computed. Then the distance that should separate the dummy nodes is computed by dividing the radius r_{C_1} by this maximum plus one. When 180^o is used for the incoming and outgoing sector then the dummy nodes can be placed on two vertical lines below and above the focus node separated by the computed distance (see Figure 7.5(a)). If one side uses more than 180^o these lines have to be rotated. Figure 7.5(b) shows a layout where the sector for the incoming nodes has 280^o and the outgoing 80^o. The lines for dummy nodes are rotated by $\pm 50^o$ ($= (280^o - 180^o) \div$ 2).

7.3 User Interface

The concentric Sugiyama-style layout offers certain possibilities for user interaction. Two different layering algorithms available. The user can make a selection by using a special layout properties

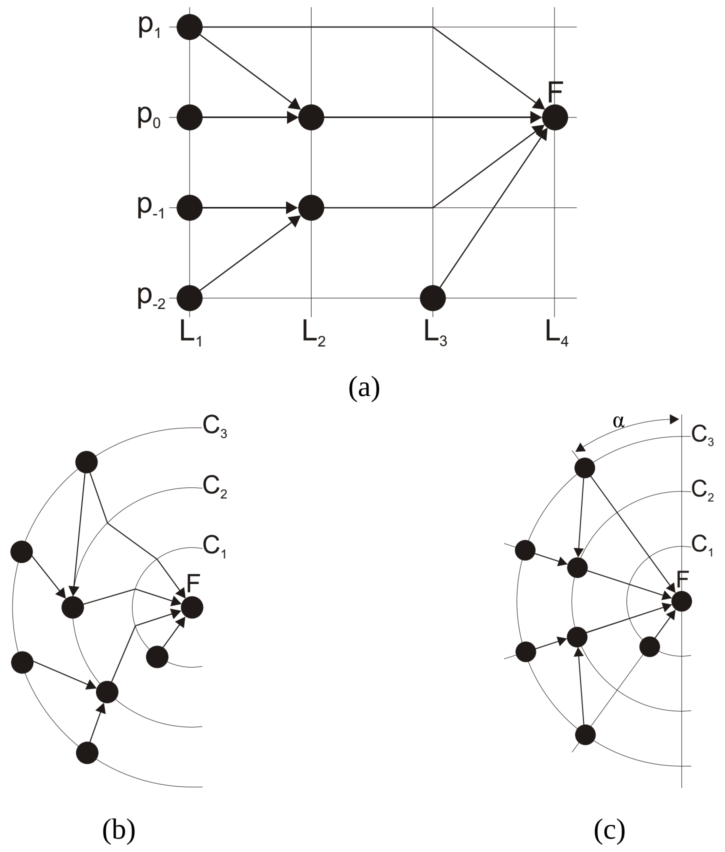


Figure 7.4: (a) shows a layout after x-coordinate assignment step. Nodes (including dummy nodes) have been assigned to layers L_i and positions p_i. (b) and (c) show a mapping of the layers from (a) to concentric circles. In (b) nodes are evenly distributed in every layer. Bend reduction and symmetry are lost. In (c) nodes keeping the same positions in different layers are assigned to the same angle which preserves bend reduction and symmetry.

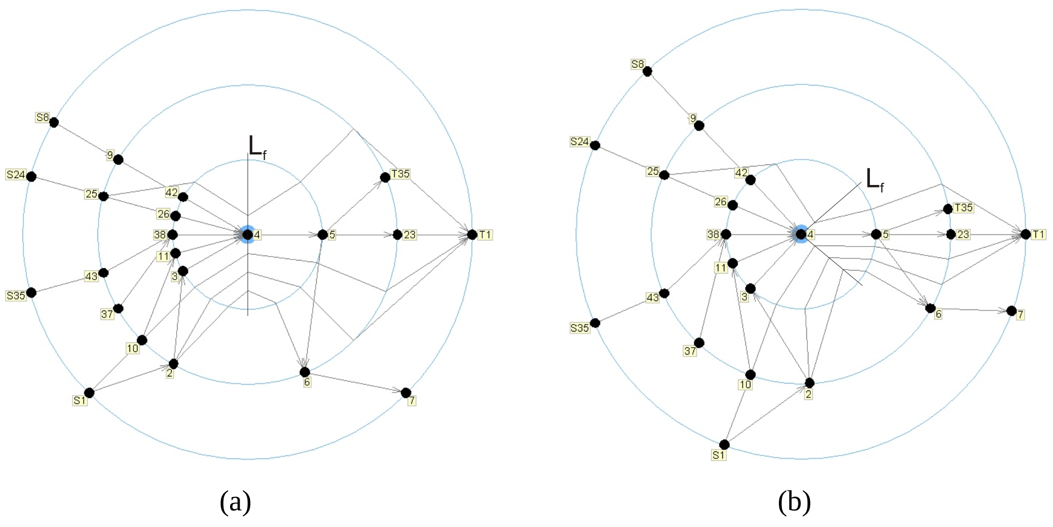
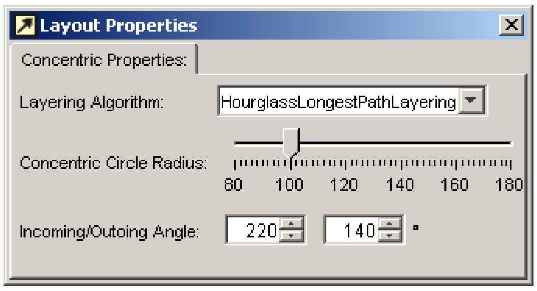


Figure 7.5: Edges between incoming and outgoing nodes are routed through the focus layer L_f. (a) shows a layout, where the incoming and outgoing sector both have angles of 180^o. In this case the dummy nodes in L_f lie on two vertical lines. In (b) the sector for the incoming nodes has 280^o and the outgoing 80^o. Here the lines for the dummy nodes are rotated by $\pm 50^o$.

dialogue (see Figure 7.6). The same interface can be used to define the radius for the concentric circles. The radius can also be modified using the left and right arrow keys. The incoming and outgoing angle can be adjusted by using the layout properties dialogue. The user can also adjust the angles by holding the *SHIFT*-key and turning the mouse wheel. All these modifications to the layout are updated immediately.


Layout Properties

Concentric Properties:

Layering Algorithm: HourglassLongestPathLayering

Concentric Circle Radius: 80 100 120 140 160 180

Incoming/Outoing Angle: 220 140 °

Figure 7.6: The User Interface Dialogue for Concentric Layout Properties can be used to select a layering algorithm, define the radius for the concentric circles and adjust the incoming and outgoing angles.

7.4 Comparison with Circular Tree Drawing

The layout of a traditional circular tree drawing algorithm is described in Section 2.3.4. Such an algorithm can not be applied to general graphs without modification. In [YFDH01] this is done by removing edges creating cycles and reinserting them later on. However, this allows no control over edge crossings. The concentric Sugiyama-style layout reduces the number of edge crossings in the crossing reduction step by considering all edges.

In a circular tree drawing a sector is assigned to every subtree which is proportional to the subtree's size. However, nodes can still not be distributed arbitrarily over the whole sector, because edges going from a node on one layer to a node on the next could leave the so called *annulus wedge* (see Figure 7.7). This can cause edge crossings [dBETT99]. When drawing trees this problem can be solved by computing the angle of the tangent and placing neighbouring nodes only within this angle. It is not possible to reserve an annulus wedge for a subgraph, because subgraphs can intersect each other in the plane. So when using the concentric Sugiyama-style layout such escaping edges can occur from time to time. Sometimes they can disturb the layout, not when crossing other edges but when two edges cover each other.

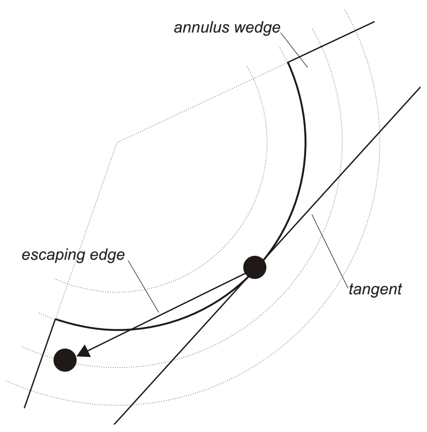


Figure 7.7: This shows the possibility of escaping edges when using a circular tree drawing. The same can happen in the concentric Sugiyama-style layout. This figure is based on a drawing in [dBETT99].

7.5 Example Concentric Layout

The concentric Sugiyama-style layout leads to very pleasing drawings. It strenghtens the impression of watching a focused node and its neighbourhood by using the circles. While a layout using node placement on parallel lines is limited by the area of the drawing plane, the concentric layout is limited by the circumference of the concentric circles. If two layers are to be inspected and the radius is adjusted such that the second concentric circle fits inside the plane, then about a hundred nodes can be aligned on the second circle using JMFGraph. A parallel line layout on the same size plane could never show a hundred nodes in one layer. When showing a focused node and its neighbourhood the innermost circles are the most important.

The layout algorithm also works well with the dynamic graph drawing algorithm described in Chapter 6. Figure 7.8 shows a transition using these two algorithms. To make the drawing even tidier Figure 7.8 shows also the use of curves instead of polylines. This technique will be described in the next chapter in more detail.

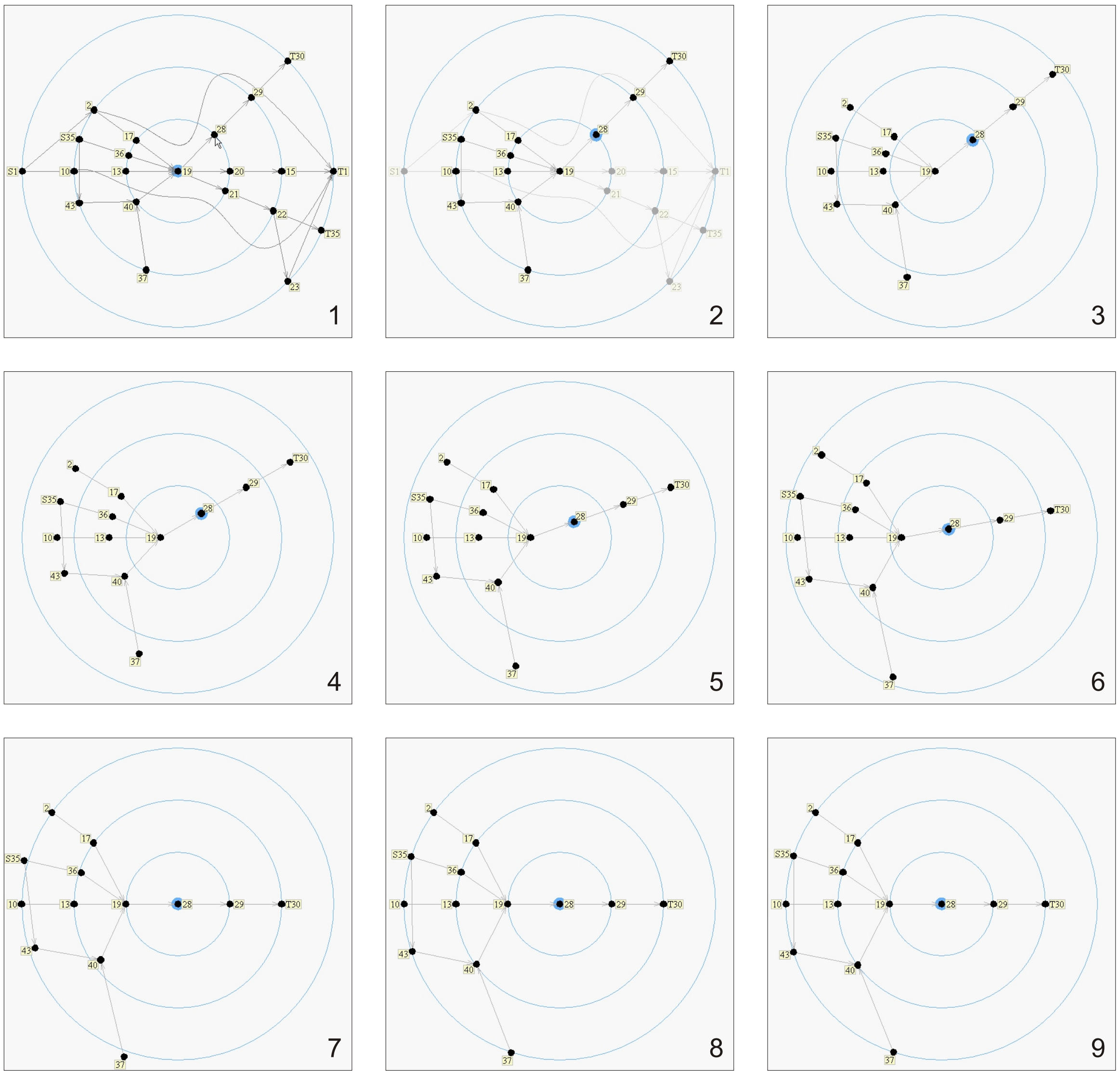


Figure 7.8: Using concentric Sugiyama-style layout together with the dynamic graph drawing algorithm described in Chapter 6. The use of curves instead of polylines can also be seen.

Chapter 8

Selected Details of the Implementation

Besides the two main inovations described in the last two chapters, there are also some other new features in the current version of JMFGraph. The most interesting will be described in this chapter. It includes a description of the parser for the GraphML file format, the approach for drawing polylines as Lagrange curves, and the export tool for saving the layout of the graph as an SVG file.

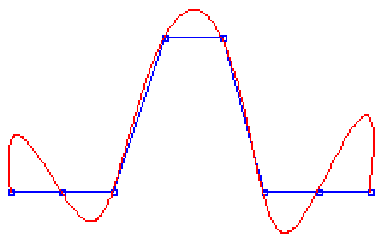
8.1 GraphML Parser

The GraphML [Gra03] (see Section 4.3.6) file parser is the second parser besides the dot file parser for reading graphs from files. GraphML is a very new standard and does not introduce any standard parameters for visual representation. Since it is expected that modular extensions will be presented in the future, it was decided to implement a parser reading only the `<node>` and `<edge>` tags for now.

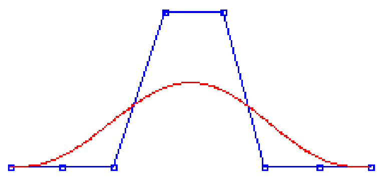
The responsible parser class is called `GraphMLFileParser` and is included in the package `jmfgraph.inputmode.file`. Since only two kinds of tags have to be processed, the parser uses *Java's SAX2 (Simple API for XML Parsing)* package. Instead of reading the whole XML hierarchy into a datastructure this API provides the possibility to process every element immediately. Every time an element is read, it is decided if it is a `<node>` or `<edge>` tag. Data tags are ignored for now. All edges are registered as directed edges, because JMFGraph provides only layout algorithms for directed graphs.

8.2 Lagrange Curves for Long Edges

Edges in a hierarchical layout of a graph that connect nodes in non-adjacent layers are controlled by dummy nodes in every layer they pass. Since the dummy nodes can have different x-coordinates, the edges can have bends. Such edges can be represented either by polylines or curves. Two important types of curves which use control points are *Bezier splines* and *B-splines*. A property of those types of curves is that the control points define the directions of the curved segments, but the control points do not lie on the curve. When thinking of Sugiyama-style algorithms, where the crossing reduction step determines the order of the nodes in each layer, it is important that the dummy nodes in the final layout preserve this order. To ensure that the curves actually go through the dummy nodes calculated in the layout, it is necessary to use another type of curve.



(a) Lagrange curve



(b) Bezier curve

Figure 8.1: Comparison of a Bezier spline and a Lagrange curve. (a) shows a Bezier spline. It can be seen that the control points are not part of the curve. (b) shows a Lagrange curve. All control nodes lie on the curve [Spl04].

In contrast, *Lagrange* curves really do go through their control nodes. The theory of the computation of Lagrange polynomials is introduced at [Spl04]. Generally, all these curves are not drawn by using arcs of circles, but by using chains of short straight line segments. The step size which determines the lengths of the line segments is chosen to give the impression of curves.

Figure 8.1 [Spl04] shows a comparison of a Bezier spline and a Lagrange curve. The curves are drawn as red lines, while the control nodes and straight line connections between them are drawn in blue. It can be seen that the Lagrange curve (a) includes its control nodes while the Bezier spline (b) does not. However, to achieve this the oscillation of the Lagrange curve has to be much higher. These oscillations can lead to edge crossings between layers.

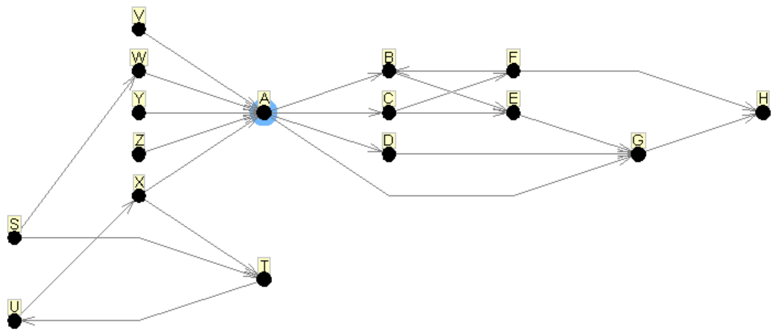
Figure 8.2 (a) shows a layout with several long edges represented as polylines with sharp corners. In (b) polylines are replaced by Lagrange curves. Using curves leads to more aesthetically pleasing layouts. Curved edges can be traced by the viewer more easily than polyline edges with bends [WPCM02].

8.3 Exporting the Graph Layout as SVG

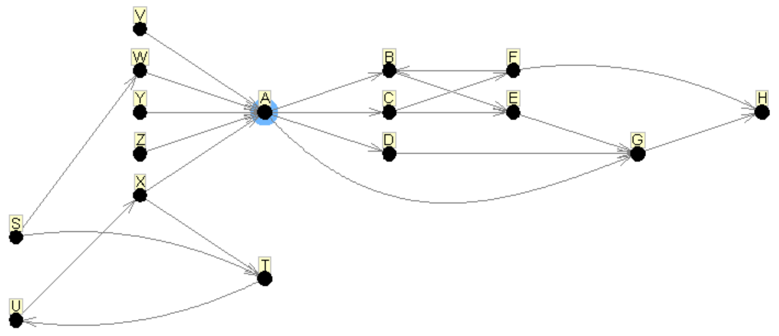
Another additional feature of the current version of JMFGraph is the possibility of exporting graph layouts to a file. The vector graphics format SVG (Scalable Vector Graphics) was chosen as the file format. SVG is an XML based file format for web graphics developed by the *World Wide Web Consortium (W3C)* [W3C04]. The specification of its current version 1.1 can be found at [SVG04]. SVG files can be viewed in web browsers by using the *Adobe SVG Viewer*. The current version 3.0 is available at [Ado04].

Since they are composed of vector graphics elements and shapes, SVG files are freely scalable without loss of quality and the introduction of artefacts.

The implementation of layout export is realised using the factory design pattern. The factory



(a) Polylines



(b) Lagrange curves

Figure 8.2: Comparison of a layout once using polylines (a) and once using Lagrange curves (b).

`jmfgraph.drawing.DrawingWriterFactory` can be used to create new instances of all registered drawing writers which extend the `jmfgraph.drawing.AbstractDrawingWriter` class. At the moment the SVG writer is the only one. Further writers for other file formats can be added by registering them in `jmfgraph.InstallReg` and adding the new class to the package `jmfgraph.drawing.writer`. Writers have to implement the following methods:

- `public void open(OutputStream output_stream)`
- `public void writeHeader(int width, int height)`

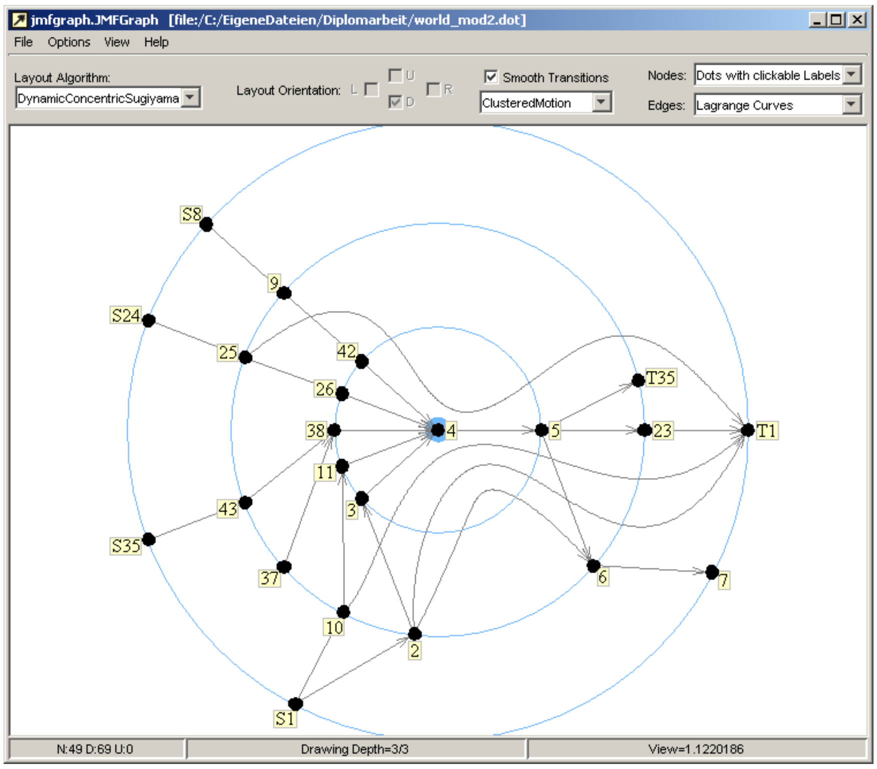
-
  ```
  public void writeLine(double x1, double y1, double x2, double y2,Color color,
                        double stroke_width, boolean dashed)
  ```

-
  ```
  public void writeCircle(int x, int y, int radius, Color color,
                          Color fill_color, int stroke_width)
  ```

-
  ```
  public void writeRectangle(double x, double y, double width, double height,
                             Color color, Color fill_color, int stroke_width)
  ```

-
  ```
  public void writeText(String string, int x, int y, String font_family,
                        int font_size, boolean italic, boolean bold,
                        Color color)
  ```
- `public void close()`

The SVG writer exports the layout exactly as it is displayed on screen and stores it to a file of the user's choice. Figure 8.3 shows a drawing in JMFGraph and the exported drawing opened with Adobe's SVG Viewer plugin. The marking of the focused node and the concentric circles of the concentric Sugiyama-style algorithm are not exported to the SVG file, because these things do not belong to the graph layout as such, but are displayed to help the user during browsing.



jmfgraph.JMFGraph [file:/C:/EigeneDateien/Diplomarbeit/world_mod2.dot]

Layout Algorithm: DynamicConcentricSugiyama

Layout Orientation: L ☐ U ☐ D ☑ R ☐

☑ Smooth Transitions

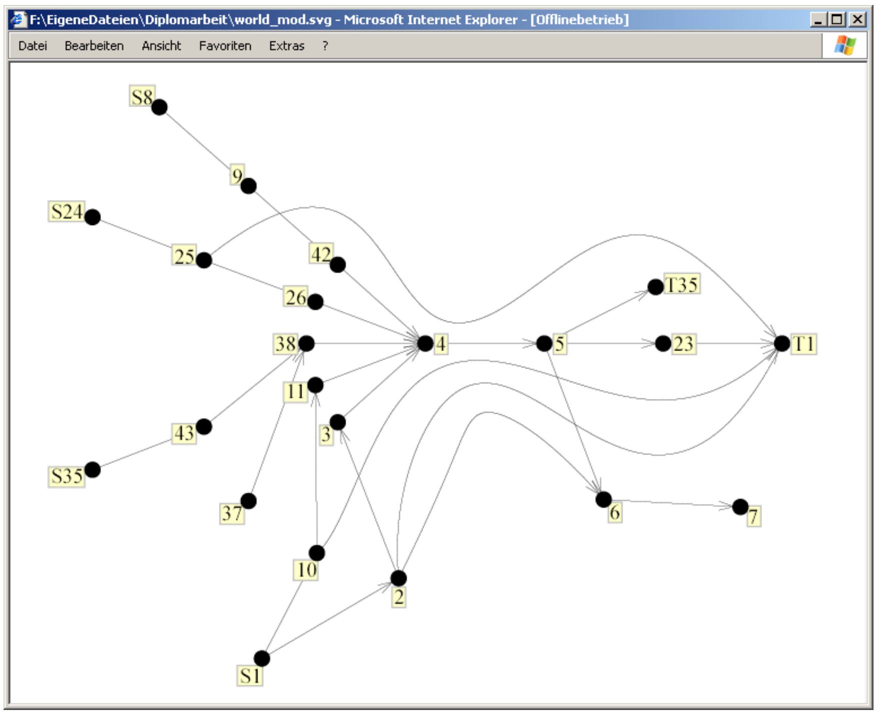
ClusteredMotion

Nodes: Dots with clickable Labels

Edges: Lagrange Curves

N:49 D:69 U:0 | Drawing Depth=3/3 | View=1.1220186

(a) JMFGraph



F:\EigeneDateien\Diplomarbeit\world_mod.svg - Microsoft Internet Explorer - [Offlinebetrieb]

(b) Browser

Figure 8.3: (a) shows a concentric graph layout as it is displayed in JMFGraph and (b) as it is displayed with Adobe's SVG Viewer plugin [Ado04] after exporting it to the vector graphics format SVG.

Chapter 9

Outlook

9.1 General Trends

Graph drawing packages including any kind of dynamic graph drawing approach are rather seldom. As summarised in Chapter 4, only a few tools currently include dynamic drawing methods. Most software are demonstration applications for new algorithms. One reason for this might be a lack of a well-founded theory behind the dynamic graph drawing approach, in contrast to static graph drawing. Sporadically, some algorithms have been published over the last few years. It will be seen in the next years if the approach presented in [FE02] will gain in popularity and be used in more applications.

User testing will have to be performed in future to find out which kind of transitions require the least effort on behalf of the user to track. It would also be important to test clustering algorithms for the dynamic graph drawing approach to find out how many individual movements can be tracked by the viewer. Also, the matter of choosing the right speed for transitions is a field of possible future investigation.

Applications using dynamic graph drawing to help the user navigate through the web, as done by tools described in [Per03], are quite rare. Many users have problems finding their way through websites, which are not well-structured. Using such maps in a second frame could improve the orientation of users.

There has been a lack of a real standard format, for storing and exchanging graphs between applications. The introduction of GraphML has possibly solved this problem. Since most of important graph drawing organisations support GraphML, it should become a real standard over the next few years.

9.2 Ideas for Future Work

Since “... all good things must come to an end” [Q94] (in the words of *Q* in the final episode of *Star Trek - The Next Generation*), there are still many possibilities for improvement of JMFGraph. The focus of this thesis has been on the introduction of the dynamic graph drawing approach into JMFGraph and the development of a suitable layout algorithm. Hence, not all of the suggestions for improvements made in [Ste01] were realised.

Sugiyama-style algorithms can have long running times when laying out very large graphs. Especially the computation time for the crossing reduction step can be very high. Therefore, it would be a good idea to start the layout algorithm in a separate thread. The user interface would then still respond while the layout is beeing computed and the user could cancel the drawing if it takes too long.

For now, the dynamic drawing approach is used to animate transitions from one layout to the next, if the focus changes. It would be nice to animate all modifications made to the layout. Since JMFGraph includes a mode where the user can jump between the steps of the layout algorithm, it would be a good idea to animate the transition when moving a step forward or backward. Especially for the crossing reduction step, this would have great eductional value if the viewer could see how the nodes change their positions in a layer. Also the switch to another layout algorithm or zooming could be smoothly animated.

There are also possibilities for improving the clustering algorithm currently used. The strategy for initial partitioning currently used just fills cluster by cluster. By changing this and using a preprocessing step like *Edge Elimination* described in [FH01] the results could be improved.

The layout export to SVG stores the static drawing as it is displayed on screen. Since SVG also supports animations it would be possible to export whole transitions to SVG. Then no screen capture tools would be needed anymore if a transition should be stored. Since SVG is vector oriented, file sizes would be small and the quality would be excellent.

The user interface of JMFGraph is a mixture of user and programer viewpoints. For example, the layout and motion algorithms can be choosen from a list box where their real classnames are displayed. This is done to pass the algorithm name directly to the factory which creates a new instance for it. In an end-user application, such things would have to be changed. When in browsing mode it would be quite a good idea to store the history of focused nodes, so the user could go back.

The GraphML parser in JMFGraph only processes the most necessary XML tags for nodes and edges. If there are standard extensions in future, the GraphML parser should be adapted to understand those tags as well.

Chapter 10

Concluding Remarks

In this thesis a new graph drawing approach was presented, which can be used to dynamically explore very large graphs. It consists of a new Sugiyama-style layout algorithm, which places nodes on concentric circles around a focused node and a dynamic graph drawing algorithm which smoothly animates transitions, when the user sets the focus to another node. This approach was integrated into an existing framework for graph drawing called *JMFGraph (Java Modular Framework for Graph Drawing)*, which is described in [Ste01].

Since the original version of JMFGraph changed the graph layout immediately when the user selected another focus, the use of dynamic graph drawing became necessary. When viewing a drawing with several objects, users create their own mental map and remember the relative positions of the objects. If the layout changes dramatically from one second to the next, the user looses this mental map and the orientation is lost. Using smooth transitions the mental map can be preserved. The new layout algorithm improves the orientation process for the user still more. With its concentric layout the viewer of the drawing can immediately recognise, which node is focused and which nodes are in its neighbourhood.

In Chapter 2, the most important existing static graph drawing approaches were presented. Traditional, Sugiyama-style algorithms and their steps were described. Also the necessary definitions concerning graph theory and graph drawing in general were introduced.

Chapter 3, introduced the dynamic graph drawing approach, which tries to preserve the mental map by calculating smooth transitions to a new layout, with as few changes as possible. The original rigid motion approach [FE02] and the k-means clustering technique for dynamic graph drawing [FH01] were described.

Chapter 4 gave a survey of graph drawing packages and applications. The relative rarity of dynamic graph drawing approaches in existing software packages was noted. Also the difficult path towards a common graph file format was summarised.

Chapter 5 explained the structure of JMFGraph by giving an overview over the responsibilities of each module. It was indicated which things were new and which were changed compared to the first version of JMFGraph.

In Chapter 6 the dynamic part of the new graph drawing approach was introduced. It was explained how the algorithm invented by *Friedrich and Eades* was adapted to be usable for a Sugiyama-style layout. Also the usage and results of the k-means clustering algorithm was shown. The advantages of browsing supported by smooth clustered transitions in contrast to immediate layout changes was shown.

The new static layout algorithm, which places nodes on concentric circles, was described in Chapter 7. It was explained why existing circular approaches are not applicable to directed graphs with a focused node and incoming and outgoing neighbours. Each step of the algorithm was then described in detail. Two different layering algorithms were presented, both creating hourglass shaped layouts. It was shown how the mapping to the concentric circles preserves straight lines. Finally, the new algorithm was compared to circular tree drawings and the combination of motion algorithm and concentric layout was shown in an example.

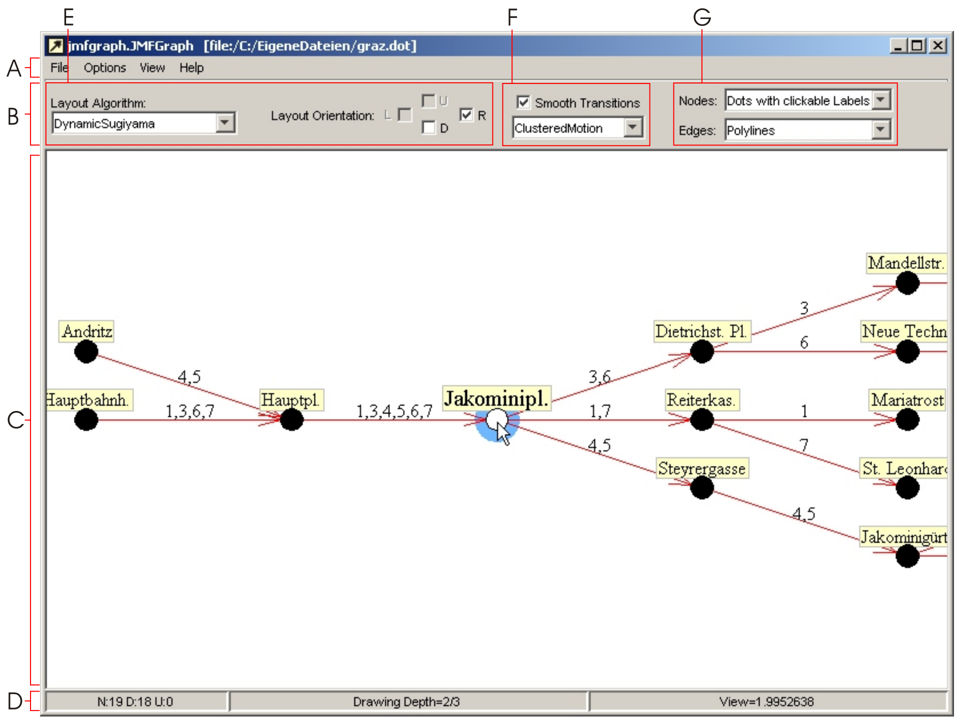
Chapter 8 covered some of the interesting details of the other new functionality in JMFGraph. The Java architecture of the GraphML parser was described. The choice of Lagrange curves instead of Bezier splines for representing edges including their control nodes was explained. Finally, the SVG export module for storing graph layouts and the simplicity of adding further file formats was explained.

Chapter 9 introduced personal ideas of future trends concerning graph drawing and suggestions for further improving JMFGraph in the future.

Appendix A

JMFGraph User Guide

A.1 The JMFGraph Work Area


jmfgraph.JMFGraph [file:/C:/EigeneDateien/graz.dot]

Layout Algorithm: DynamicSugiyama

Layout Orientation: L ☐ U ☐ D ☐ [✓ R]

[✓ Smooth Transitions] ClusteredMotion

Nodes: Dots with clickable Labels

Edges: Polylines

N:19 D:18 U:0 | Drawing Depth=2/3 | View=1.9952638

Figure A.1: This figure shows the work area of JMFGraph. Menu Bar (**A**), Tool Bar (**B**), Drawing Pane (**C**), Status Bar (**D**), Layout Tool Bar (**E**), Transition Tool Bar (**F**), Representation Tool Bar (**G**).

Figure A.1 shows the work area of JMFGraph. It consists of a drawing pane (C), where the graph is displayed, a menu bar (A), a tool bar (B) and a status bar (D). All functions can be accessed either by using the menu or the tool bar.

Using the *Layout Tool Bar* (E) the static layout algorithm used for computing the layout of the drawing can be chosen. The orientation of the layout can be modified by selecting one of the four checkboxes. The user can select **U**(bottom up) or **D**(top down) for creating a vertical layout or **L**(right left) or **R**(left right) to create a horizontal layout. For further information on layout algorithm properties see Section A.6.

If the chosen layout algorithm supports dynamic exploration (*Dynamic Sugiyama* or *Dynamic Concentric Sugiyama*) the mouse can be used to select the focused node. A new layout with the new focus in the centre is computed and displayed in the drawing pane (C). To enable smooth transitions between the old and new layout, the checkbox in the *transition tool bar* (F) has to be enabled. Using the list box in the transition tool bar a dynamic graph drawing algorithm can be selected, which is used to compute the transitions. Special properties for the motion algorithm can be defined by using the dialogue described in Section A.5.

The *Representation Tool Bar* (G) can be used to change the graphical representation of nodes and edges. When using the node representation *Dots with clickable Labels* it is possible to change the focused node by clicking on its label. If labels cover each other partially, the right mouse key can be used to bring a label to the foreground. To change the labels' fonts see Section A.3.

The leftmost display in the *Status Bar* (D) shows the number of nodes and the number of directed and undirected edges of the entire graph currently opened. In the centre the current incoming and outgoing drawing depths are displayed. On the right hand side, the current zoom factor can be seen.

A.2 Importing and Exporting Graphs

A graph can be opened from a file by selecting *File* > *Open File*. A dialogue opens where the user can first choose one of the supported file formats and then select a file. Another possibility is to choose one of the demonstration graphs, which are available by selecting *File* > *Open Demo*. The dialogues are also accessible via keyboard shortcuts by typing Cmd + O or Cmd + D.

If the user wants to export the layout of the graph as it is displayed on screen, this can be done by choosing *File* > *Export Drawing* or typing Cmd + E. A dialogue opens, where a supported file format and a file name can be chosen. Currently, it is only posssible to export graph layouts in SVG format.

A.3 Changing the View and Font Properties

To change the magnification of the displayed drawing the user can use the zoom tools in the *View* menu. To zoom in, either *View* > *Zoom In* or *View* > *Zoom In ++*, for larger steps, can be chosen. The same can be done by typing + or turning the mouse wheel up, for small zooming steps, and using Page Up for large zooming steps. For zooming out, *View* > *Zoom Out*, *View* > *Zoom Out ++*, -, turning the mouse wheel down or Page Down can be used. To alter the zoom factor Alt or Cmd can be additionally pressed. While Alt decreases the zoomfactor, Cmd increases it. To return to the default magnification either *View* > *View 1:1* from the menu can be chosen or Home can be typed. All shortcuts are summarised in Table A.2.

When zooming out of the drawing labels can become too small to read or too large when zooming in. To adjust the font properties, the font chooser tool (see Figure A.2) can be used. To open it either choose *View* > *Font Chooser* from the menu or type Cmd + F.

To change the font the user can select a font family, a font style, and a font size. A sample text, with the chosen font is displayed in the centre of the dialogue. To apply the current settings to the labels in the graph press the *APPLY* button. All node and edge labels are displayed with the selected font and the font properties defined in the graph's file are overruled. If the *OK* button is pressed, the font properties are also updated and the dialogue is closed. By pressing *CANCEL* the dialogue is

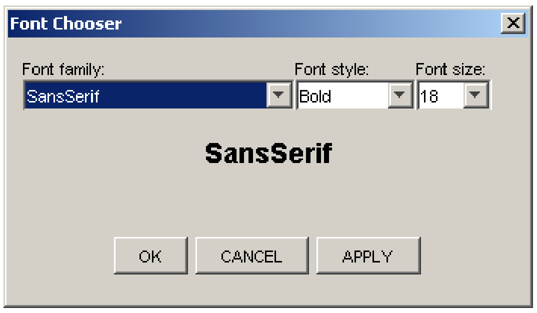


Figure A.2: To change the font properties of the graph's labels the font chooser can be used.

closed without applying the new font properties. When a new graph is opened the fonts defined in the file for the new graph are used.

A.4 Using the Step Mode

All layout algorithms used in JMFGraph consist of several subalgorithms. The Sugiyama-style algorithms for example have four steps. To demonstrate the functionality of every step, it is convenient to show the changes to the layout step by step. This can be done by opening the step mode dialogue from the *Options* menu or pressing Cmd + S.

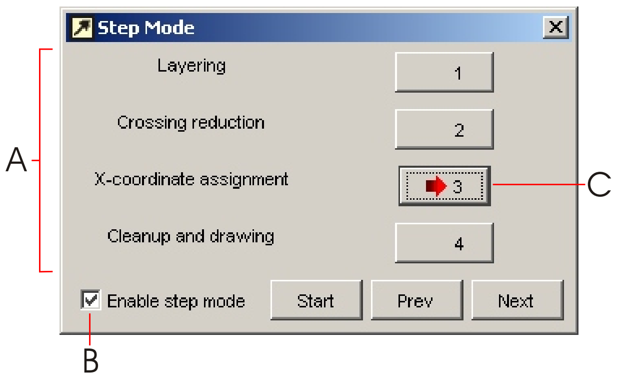
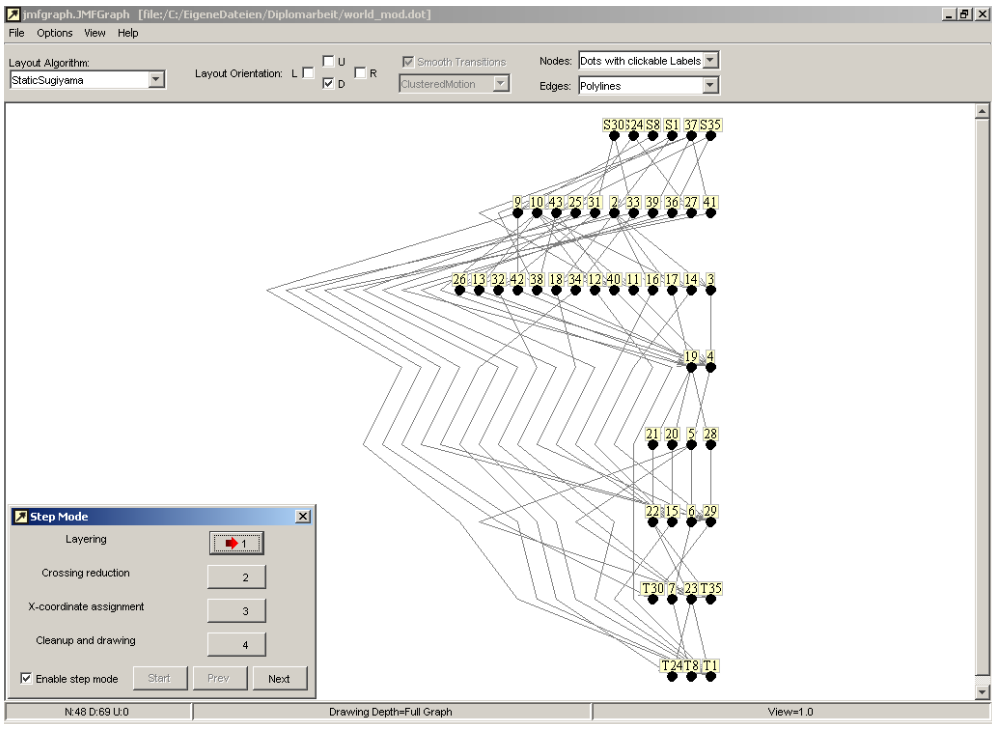


Figure A.3: To demonstrate the functionality of each step of the layout algorithm, the step mode dialogue can be used. It shows the names of all steps and provides buttons to select the desired step(A). If for example "X-coordinate assignment" is selected (C) all of "Layering", "Crossing reduction" and "X-coordinate assignment" are applied to the layout. Also the buttons *Start*, *Prev* and *Next* can be used to select the final step to be applied. To enable and disable the step mode, checkbox (B) can be used.

Figure A.3 shows the dialogue and marks its important areas. To enable step mode, the checkbox at the bottom left corner (B) has to be selected. Once step mode is enabled a certain step can be selected. This can be done by clicking the buttons to the right of the steps' names (A) or by using the


jmfgraph.JMFGraph [file:/C:/EigeneDateien/Diplomarbeit/world_mod.dot]

Layout Algorithm: StaticSugiyama

Layout Orientation: L ☐ U ☐ D ☑ R ☐

☑ Smooth Transitions ClusteredMotion

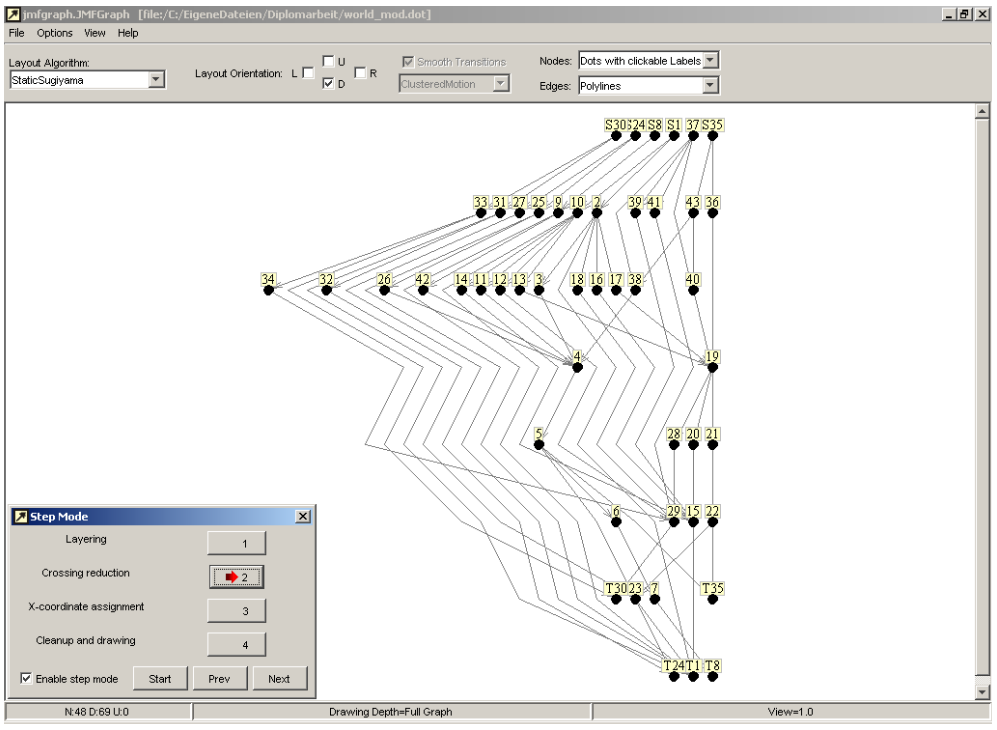
Nodes: Dots with clickable Labels

Edges: Polylines

☑ Enable step mode

N:48 D:69 U:0 | Drawing Depth=Full Graph | View=1.0

(a)


jmfgraph.JMFGraph [file:/C:/EigeneDateien/Diplomarbeit/world_mod.dot]

Layout Algorithm: StaticSugiyama

Layout Orientation: L ☐ U ☐ D ☑ R ☐

☑ Smooth Transitions ClusteredMotion

Nodes: Dots with clickable Labels

Edges: Polylines

☑ Enable step mode

N:48 D:69 U:0 | Drawing Depth=Full Graph | View=1.0

(b)

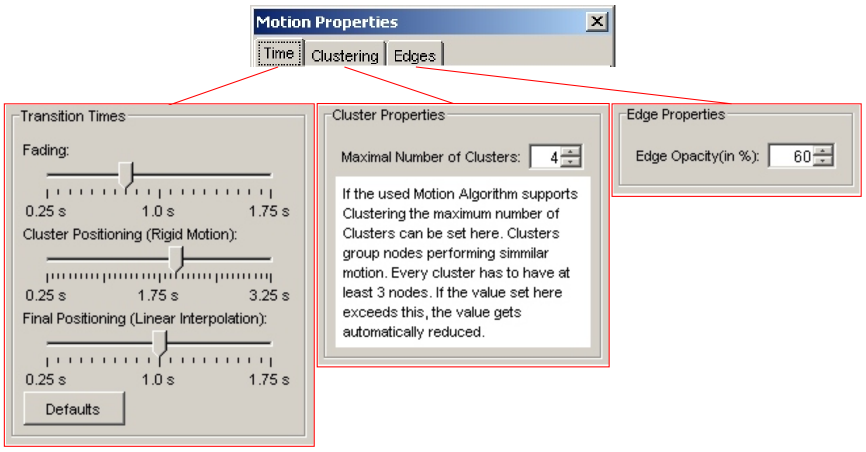
Figure A.4: This figure shows the use of the step mode dialogue. (a) shows the layout after the "Layering" step. (b) shows the layout after "Crossing reduction" has been applied.

buttons *Start*, *Prev* and *Next*. The currently selected step is marked by a red arrow. If a certain step is selected, all steps upto and including the selected step are performed.

Figure A.4 shows two screenshots of JMFGraph using the step mode dialogue. In Figure A.4(a) “Layering” is the currently selected step in the dialogue. The drawing shows the graph with its nodes assigned to layers, but without any crossing reduction. Looking at Figure A.4(b) it can be seen that crossing reduction is included in the computation of the graph layout by selecting step 2 in the step mode dialogue.

A.5 Dynamic Exploration of Graphs

To dynamically explore a graph, it is necessary to select a layout algorithm which supports dynamic focused drawing from the layout algorithms select box (see Figure A.1 (E)). At the moment two such algorithms are available: *DynamicSugiyama* and *DynamicConcentricSugiyama*. Once the algorithm is selected, the user can select a node to be the focus. A new layout, placing this node in the centre of the drawing, is then computed. The smooth transition, if enabled (see Figure A.1 (F)), is computed using the selected dynamic graph drawing algorithm. At the moment, a single algorithm called *ClusteredMotion* is available. To define its properties, the motion properties dialogue can be opened by selecting *Options* > *Motion Properties* or using the keyboard shortcut Cmd + M.



Maximal Number of Clusters: 4

If the used Motion Algorithm supports Clustering the maximum number of Clusters can be set here. Clusters group nodes performing simmilar motion. Every cluster has to have at least 3 nodes. If the value set here exceeds this, the value gets automatically reduced.

Edge Opacity(in %): 60

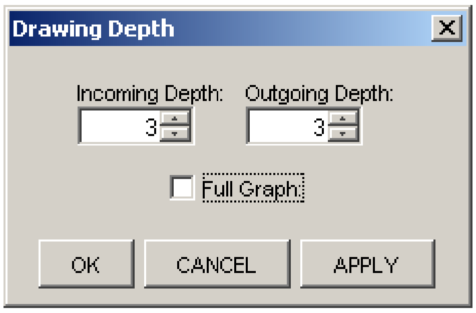
Figure A.5: The motion properties dialogue provides three tabbed palettes, which can be used to adjust transition time, clustering and fading properties of the dynamic graph drawing algorithm.

The motion properties dialogue contains three tabbed palettes: *Time*, *Clustering* and *Edges*. The Time palette includes adjustments concerning transition times. The user can adjust three different times by using sliders, as can be seen in Figure A.5. The slider labelled *Fading* defines the time spent on fading out old nodes and edges and fading in new nodes and edges during transition. The second slider labelled *Cluster Positioning* defines the duration of the transition process, whereby nodes are moved in groups as rigid objects. The last slider is responsible for setting the time of the *Final Positioning* which brings the nodes to their end positions by moving them along straight lines. All adjustments are made in seconds. To set the sliders back to their defaults, the button at the bottom left can be used.

The *Clustering* palette is available if the dynamic drawing algorithm supports clustering of nodes performing similar motion. A input box (spinner) is provided, that allows the user to set the maximum number of used clusters. If the algorithm decides that the number is too high, a smaller number of clusters is used. It never exceeds the number set by the user. The *Fading* palette provides the possibility to define the opacity of edges during transition. By default, the value is set to 60 percent.

A.6 Changing Graph Layout Properties

When using a layout algorithm supporting focused drawing, the user can select the subset of nodes around the focused node by setting the incoming and outgoing drawing depth. Since JMFGraph works with directed graphs, incoming and outgoing drawing depth can be adjusted independently using the drawing depth dialogue shown in Figure A.6. For example, an outgoing drawing depth of three means, that all nodes having a distance upto three from the focus node and beeing reachable by outgoing edges starting at the focus node are drawn. The dialogue can be opened by selecting *View > Drawing Depth* or by using the shortcut Alt + D.



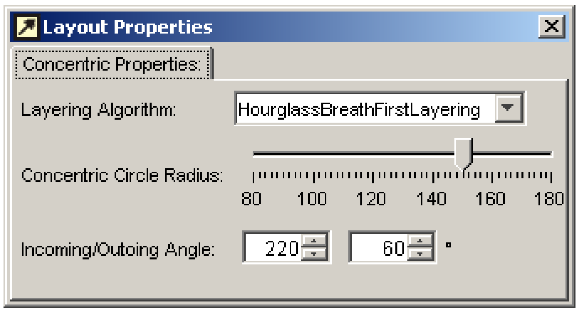
Incoming Depth: 3
Outgoing Depth: 3

Figure A.6: The drawing depth dialogue can be used to set the incoming and outgoing drawing depth around the focus node. To show the full graph the checkbox can be selected, if supported by the drawing algorithm.

To adjust the drawing depth the two input boxes (spinners) at the top of the dialogue can be used. To update the drawing the button *APPLY* has to be pressed. If the checkbox labelled *Full Graph* is enabled, which means that the drawing algorithm supports displaying the entire graph, it can be selected.

Special properties of layout algorithms can be adjusted using the layout properties dialogue available by selecting *Options > Layout Properties* or by using the keyboard shortcut Cmd + L. As can be seen in Figure A.7, only one palette for the concentric layout algorithm is available at the moment.

The select box at the top can be used to select the layering algorithm. Moving the slider in the middle changes the radius of the concentric circles. The same effect can be achieved using the left and right arrow key (see also Table A.3). To change the angles allocated to the sectors, which are used for drawing the incoming and outgoing nodes, the two input boxes (spinners) at the bottom can be used. Another possibility is to use the mousewheel in combination with holding Shift. By turning the wheel up the outgoing angle is increased. If the sum of outgoing angle plus incoming angle would exceed 360 degrees, the outgoing angle is automatically decreased. By turning the wheel down, the same is done for the incoming angle.



Layering Algorithm: HourglassBreathFirstLayering

Concentric Circle Radius: 80 100 120 140 160 180

Incoming/Outoing Angle: 220 60 °

Figure A.7: The layout properties dialogue can be used to adjust properties concerning the layout algorithm. At the moment adjustments are provided for the concentric layout algorithm described in.

A.7 Shortcuts and Mouse Controls

This section summarises all shortcuts introduced in the previous sections. Table A.1 list standard commands. Table A.2 summarises controls necessary for viewing the graph and Table A.3 lists shortcuts for commands affecting the concentric layout.

Command	**Shortcut**
Open File	Ctrl + O
Open Demo	Ctrl + D
Export Drawing	Ctrl + D
Quit	Alt + F4
Layout Properties	Alt + L
Motion Properties	Alt + M
Drawing Depth	Alt + D
Font Properties	Ctrl + F

Table A.1: Standard Commands.

Command	**Shortcut**
View 1:1/Fit in Window	Home
Zoom In	+ or Mousewheel Up
Zoom Out	- or Mousewheel Down
Zoom In ++	Page Up
Zoom Out ++	Page Down
Increase Zoom Factor	Ctrl
Decrease Zoom Factor	Alt
Move Node Label to Foreground	Right Mouse Key
Refresh	F5

Table A.2: View controls.

Command	**Shortcut**
Increase Outgoing Angle	Shift + Mousewheel Up
Decrease Outgoing Angle	Shift + Mousewheel Down
Increase Circle Radius	Right Arrow
Decrease Circle Radius	Left Arrow

Table A.3: Shortcuts for commands affecting the concentric layout.

Bibliography

[Ado04] Adobe SVG Zone. Adobe, 2004. http://www.adobe.com/svg/main.html.

[AGD03] AGD - A library of algorithms for graph drawing. Vienna University of Technology, 2003. http://www.ads.tuwien.ac.at/AGD/.

[aiS03] aiSee Graph Visualization. AbsInt, 2003. http://www.absint.com/aisee/.

[BEH$^+$01] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. GraphML Progress Report. In *Proceedings of the Symposium on Graph Drawing* [GD'01], pages 501–512.

[BLV98] Giuseppe Di Battista, Renato Lillo, and Fabio Vernacotola. Ptolomaeus: The web cartographer. In *Proceedings of the Symposium on Graph Drawing* [GD'98], pages 444–445.

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

[Car80] Marie-José Carpano. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-10(11):705–715, November 1980.

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

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

[cri03] SiteBrain Demo. cristella.com, 2003. http://www.cristalla.com/SiteBrain/SiteBrainDemo/SiteBrain%20Demo/plexfloating.htm.

[CT94] Isabel F. Cruz and Roberto Tamassia. How to visualize a graph: Specification and algorithms. Tufts University / Brown University, 1994. Tutorial on Graph Drawing. http://www.cs.brown.edu/calendar/gd94/tutorial.html.

[DAT03] DA-TU storyboard. Mao Lin Huang, 2003. http://www-staff.mcs.uts.edu.au/˜maolin/jgaa_demo/jgaa_demo.html.

[daV03] The Graph Visualization System daVinci V2.1. University of Bremen, 2003. http://www.tzi.de/˜davinci/.

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

[DFM$^+$02] Ugur Dogrusoz, Qingwen Feng, Brendan Madden, Michael Doorley, and Arne Frick. Graph visualization toolkits. *IEEE Computer Graphics and Applications*, 22(1):30–37, February 2002.

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

[dVP03] daVinci Presenter. b-novative, 2003. `http://www.daVinci-Presenter.de/`.

[Ead84] Peter Eades. A heuristic for graph drawing. In *Congressus Numerantium 42*, pages 149–160, 1984. `http://www.cs.usyd.edu.au/~peter/old_spring_paper.pdf`.

[ELMS91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.

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

[FE02] Carsten Friedrich and Peter Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications*, 6(3):353–370, 2002. ISSN 1526-1719 `http://jgaa.info`.

[FH01] Carsten Friedrich and Michael E. Houle. Graph drawing in motion II. In *Proceedings of the Symposium on Graph Drawing* [GD'01], pages 220–231.

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

[GD'94] GD'94. *Proceedings of the Symposium on Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, Princeton, New Jersey, USA, September 1994. Springer. ISBN 3-540-58950-3.

[GD'98] GD'98. *Proceedings of the Symposium on Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, Montréal, Canada, August 1998. Springer. ISBN 3-540-65473-9.

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

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

[GD'01] GD'01. *Proceedings of the Symposium on Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, Vienna, Austria, September 2001. Springer. ISBN 3-540-43309-0, `http://www.ads.tuwien.ac.at/gd2001/`.

[GD'03] GD'03. *Proceedings of the Symposium on Graph Drawing*, Lecture Notes in Computer Science, Perugia, Italy, September 2003. Springer. `http://www.gd2003.org/`.

[GDT03] GDToolkit. University of Rome, 2003. `http://www.dia.uniroma3.it/~gdt/editablePages/main_index.htm`.

[GJK$^+$01] Carsten Gutwenger, Michael Jnger, Gunnar W. Klau, Sebastian Leipert, Petra Mutzel, and René Weiskircher. AGD: A library of algorithms for graph drawing. In *Proceedings of the Symposium on Graph Drawing* [GD'01], pages 473–474.

[GM03] Intersection-Free Morphing of Planar Graphs. Cesim Erten and Stephen G. Kobourov and Chandan Pitta, 2003. `http://gmorph.cs.arizona.edu/`.

[GML03] The GML File Format. University of Passau, 2003. `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/`.

[Gra03] The GraphML File Format. graphdrawing.org, 2003. `http://graphml.graphdrawing.org/index.html`.

[GTV03] gnuTellaVision: Real Time Visualization of a Peer to Peer Network. Rachna Dhamija, Danyel Fisher, Ka-Ping Yee, 2003. `http://www.sims.berkeley.edu/~rachna/gtv/`.

[GV03] Graphviz. AT&T Labs - Research, 2003. `http://www.research.att.com/sw/tools/graphviz/`.

[GVT03] Graph Visualization Toolkits. Tom Sawyer Software, 2003. `http://www.tomsawyer.com/products.html`.

[GXL03] Graph eXchange Language. University of Koblenz-Landau, 2003. `http://www.gupro.de/GXL/`.

[H303] H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space. Tamara Munzner, 2003. `http://graphics.stanford.edu/papers/h3/`.

[Hah02] Philipp Hahn. Visualisierung von Graphenalgorithmen. Master's thesis, Darmstadt University of Technology, 2002.

[Har72] Frank Harary. *Graph Theory*. Addison-Wesley, 1972. ISBN 0-201-02787-9.

[HE98] Mao Lin Huang and Peter Eades. A heuristic for graph drawing. In *Proceedings of the Symposium on Graph Drawing* [GD'98], pages 374–383. ISBN 3-540-65473-9.

[HEW98] Mao Lin Huang, Peter Eades, and Junhu Wang. Online animated graph drawing using a modified spring algorithm. *Australian Computer Science Comm.: Proc. 21st Australasian Computer Science Conf., ACSC*, 20(1):17–28, 4–6 1998. ISBN 981-3083-90-5.

[Him96] Michael Himsolt. GML: A portable graph file format. Technical report, Universität Passau, 1996.

[HM00] Ivan Herman and M. Scott Marshall. GraphXML - An XML-Based Graph Description Format. In *Proceedings of the Symposium on Graph Drawing* [GD'00], pages 52–62.

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

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. Technical Report 1–2000, University of Koblenz-Landau, Institute for Software Technology, 2000.

[HZ02] Mao Lin Huang and Kang Zhang. Navigating Product Catalogs Through OFDAV Graph Visualization. In *Proc. of International Conference on Distributed Multimedia Systems*, pages 555–561, September 2002. ISBN 1-891706-11-x.

[INF97] INFOVIS'97. *IEEE Symposium on Information Visualization*, Phoenix, AZ, USA, October 1997. IEEE Computer Society. ISBN 0-8186-8189-6.

[INF01] INFOVIS'01. *IEEE Symposium on Information Visualization*, San Diego, CA, USA, October 2001. IEEE Computer Society. ISBN 0-7695-1342-5, `http://infovis.org/infovis2001/`.

[INF02] INFOVIS'02. *IEEE Symposium on Information Visualization*, Boston, MA, USA, October 2002. IEEE Computer Society. ISBN 0-7695-1751-X, `http://infovis.org/infovis2002/`.

[INF03] INFOVIS'03. *IEEE Symposium on Information Visualization*, Seattle, Washington, USA, October 2003. IEEE Computer Society. `http://infovis.org/infovis2003/`.

[KCH02] Yehuda Koren, Liran Carmel, and David Harel. Ace: A fast multiscale eigenvectors computation for drawing huge graphs. In *IEEE Symposium on Information Visualization* [INF02], pages 137–144.

[KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graph. *Information Processing Letters*, 31(1):7–15, April 1989. ISSN 0020-0190.

[KN93] Eleftherios Koutsofios and Stephen C. North. Drawing graphs with dot. AT&T Bell Laboratories, 1993. `http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`.

[KP03] S. G. Kobourov and C. Pitta. Intersection-free morphing of planar graphs. In *Proceedings of the Symposium on Graph Drawing* [GD'03].

[KW01] Michael Kaufmann and Dorothea Wagner. *Drawing Graphs: Methods and Models*. Springer, 2001. ISBN 3-540-42062-2.

[Lin03] Mao Lin's home page. Mao Lin Huang, 2003. `http://www-staff.mcs.uts.edu.au/~maolin/SiteView.html`.

[Lon04] Tube Guru enlightening London. Transport for London, 2004. `http://tube.tfl.gov.uk/guru/index.asp`.

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

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

[Mun97] Tamara Munzner. H3, laying out large directed graphs in 3D hyperbolic space. In *IEEE Symposium on Information Visualization* [INF97], pages 2–10.

[Per03] PersonalBrain. TheBrain Technologies Corporation, 2003. `http://www.thebrain.com/Default.htm`.

[Pto03] Ptolomaeus - the Web cartographer. University of Roma3, 2003. `http://www.dia.uniroma3.it/~ptolemy/`.

[Q94] Q. All Good Things. Star Trek - The Next Generation, 1994.

[RCM89] G. Robertson, S. K. Card, and J. D. Mackinlay. The cognitive coprocessor architecture for interactive user interfaces. In *Proceedings of the 2nd annual ACM SIGGRAPH symposium on User interface software and technology*, pages 10–18. ACM Press, 1989. ISBN 0-89791-335-3.

[RM88] Marcello G. Reggiani and Franco E. Marchetti. A proposed method for representing hierarchies. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-18(1):2–8, February 1988.

[San94] Georg Sander. Graph layout through the VCG tool. In *Proceedings of the Symposium on Graph Drawing* [GD'94], pages 194–205.

[Sch98] Jürgen Schipflinger. The Design and Implementation of the Harmony Session Manager. Master's thesis, Graz University of Technology, 1998. `ftp://ftp.iicm.edu/pub/theses/jschipf.pdf`.

[SD92] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface '92*, pages 258–264, 1992.

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

[Spl04] An Interactive Introduction to Splines. Evgeny Demidov, 2004. `http://www.ibiblio.org/e-notes/Splines/Intro.htm`.

[Ste01] Alexander Stedile. JMFGraph: A Modular Framework for Drawing Graphs in Java. Master's thesis, Graz University of Technology, 2001. `ftp://ftp.iicm.edu/pub/theses/astedile.pdf`.

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

[Sug02] Kozo Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*. Series on Software Engineering and Knowledge Engineering; Vol.11. World Scientific, 2002. ISBN 981-02-4879-2.

[SVG04] Scalable Vector Graphics (SVG) 1.1 Specification. W3C, 2004. `http://www.w3.org/TR/2002/CR-SVG11-20020430/`.

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

[VCG04] VCG Overview. Georg Sander, 2004. `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`.

[Vis03] The Visual Thesaurus, a Dictionary of the English Language. plumbdesign, 2003. `http://www.visualthesaurus.com/online/index.html`.

[vWN03] Jarke J. van Wijk and Wim A. A. Nuij. Smooth and efficient zooming and panning. In *IEEE Symposium on Information Visualization* [INF03], pages 15–22.

[W3C04] World Wide Web Consortium. W3C, 2004. `http://www.w3.org/`.

[Wor03] WordNet a lexical database for the english language. Princeton University, 2003. `http://www.cogsci.princeton.edu/˜wn/`.

[WPCM02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(1):103–110, June 2002. ISSN 1473-8716.

[XGM03] eXtensible Graph Markup and Modeling Language. Rensselaer Polytechnic Institute, 2003. `http://www.cs.rpi.edu/˜puninj/XGMML/`.

[YFDH01] Ka-Ping Yee, Danyel Fisher, Rachna Dhamija, and Marti Hearst. Animated exploration of dynamic graphs with radial layout. In *IEEE Symposium on Information Visualization* [INF01], pages 43–50.