Offline Drawing of Dynamic Graphs

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Diplom-Informatiker

Carsten Görg

Saarbrücken
April 2005

Tag des Kolloquiums:	16.06.2005
Dekan:	Prof. Dr. Jörg Eschmeier
Prüfungsausschuss:	
Gutachter:	Prof. Dr. Reinhard Wilhelm Prof. Dr. Stephan Diehl, Katholische Universität Eichstätt
Vorsitzender:	Prof. Dr. Philipp Slusallek
Akademische Mitarbeiterin:	Dr. Nicola Wolpert

Abstract

In this work new approaches to the problem of offline drawing dynamic graphs are presented: how to draw a given sequence of graphs which evolve over time by adding and/or deleting nodes and/or edges, such that the local quality of each drawing as well as the dynamic stability of all drawings is maximized. Unfortunately, these two goals conflict with each other in general. The first goal, namely maximizing the quality of a drawing for one single graph, is widely researched in the graph drawing community. The second goal, maximizing the stability of all drawings, also known by the term "preserving the mental map", has been identified to be crucial for the usability of systems for drawing of dynamic graphs.

To deal with the problem of conflicting optimization goals, new algorithms are developed which are parameterized such that they are able to trade stability for local quality and vice versa. Three different categories of graph drawing methods are investigated: force-directed, hierarchic, and orthogonal. All of these algorithms are integrated into a generic framework.

Metrics are used to check the stability of two successive drawings. So far, only metrics working on complete drawings of graphs have been known. Because the orthogonal and hierarchical algorithms use several phases, new metrics are introduced which work on intermediate results, that is on partial drawings of graphs and structures of drawings respectively.

To show the usefulness of this approach, several case studies in different application domains like algorithm animation and software visualization are presented.

Zusammenfassung

Diese Arbeit befasst sich mit dem Offline-Problem des Zeichnens dynamischer Graphen: wie kann eine Sequenz gegebener Graphen, die sich über die Zeit durch Einfügen bzw. Löschen von Knoten bzw. Kanten verändern, gezeichnet werden, dass sowohl die lokale Qualität jeder einzelnen Zeichnung als auch die dynamische Stabilität aller Zeichnungen maximiert wird. Diese beiden Ziele stehen im Allgemeinen leider in Konflikt zueinander. Das erste Ziel, die Qualität einer Graphzeichnung zu maximieren, ist von den Graphzeichnern schon weit erforscht. Das zweite Ziel, die Stabilität von Zeichnungen einer Graphsequenz, auch bekannt als "Erhaltung der Mental Map", wurde als entscheidend für die Nutzbarkeit von Systemen für dynamisches Graphzeichnen identifiziert.

Um das Problem der zueinander in Konflikt stehenden Optimierungsziele zu lösen, werden neue Algorithmen entwickelt, die parametrisiert sind, so dass sie Stabilität gegen lokale Qualität eintauschen können und umgekehrt. Graphzeichenmethoden drei verschiedener Kategorien werden untersucht: kräftebasierte, hierarchische und orthogonale. Diese Algorithmen werden in ein generisches Framework integriert.

Um die Stabilität zweier aufeinanderfolgender Zeichnungen zu bewerten, werden Metriken benutzt. Bis jetzt waren allerdings nur Metriken bekannt, die auf vollständigen Zeichnungen von Graphen arbeiten. Da der orthogonale und der hierarchische Algorithmus aus mehreren Phasen bestehen, werden neue Metriken entwickelt, die auf den Zwischenergebnissen dieser Phasen arbeiten, das heißt auf unvollständigen Zeichnungen von Graphen bzw. auf Strukturen dieser Zeichnungen.

Anhand von Fallstudien in verschiedenen Anwendungsgebieten wie Algorithmenanimation und Softwarevisualisierung wird der praktische Nutzen dieser Methode gezeigt.

Extended Abstract

Dynamic graph drawing has many applications and got more and more into the focus of the researchers. The basic difference between static and dynamic graph drawing algorithms is that in the dynamic case, in addition to producing an aesthetic drawing, the algorithm has to minimize changes to the user's mental map.

This thesis introduces a generic framework for the problem of offline drawing dynamic graphs: how to draw a given sequence of graphs which evolve over time by adding and/or deleting nodes and/or edges, such that the local quality of each drawing as well as the dynamic stability of all drawings is maximized.

First, we present the motivation and theory behind foresighted graph drawing. This approach projects the graphs of the sequence to the so called super graph, which represents an approximation of the whole sequence, then computes a drawing of the super graph using a standard static drawing algorithm. This drawing is used as a template, that is the drawings of the individual graphs of the sequence are induced by the drawing of the super graph. Thus neither nodes nor edges change their positions in the drawing and a high global stability is guaranteed. The disadvantage is that the local quality of individual drawings is sometimes poor because the quality is restricted by the drawing of the super graph.

To counteract this effect we extend the foresighted graph drawing approach by allowing local optimization of the individual drawings up to a given threshold. We use difference metrics to measure if the drawing resulting from the local adaptation is in the allowed range to guarantee the preserving of the mental map.

Thus, it is possible to trade local quality (of the drawing) for global stability (of all drawings of the sequence) by using a low or a high threshold respectively: a low threshold ensures global stability, but we have to pay for this with a reduced quality of the drawing and a high threshold allows drawings of better quality, but the stability is reduced.

We present the concept of a backbone as generalization of the super graph. The backbone does not contain all graphs of the sequence like the super graph, but only nodes of a high importance for the sequence of graphs. The importance of the nodes can be arbitrarily defined. Is is possible to use statistical information, for example nodes contained in many graphs of the sequence are more important than nodes only contained in few graphs, or it is possible as well to use semantical information if they are available.

The advantage of the tolerant version of foresighted graph drawing is that it is possible to decide what is more important for a specific graph sequence: emphasizing the quality of the individual drawings or preserving the mental map. The disadvantage is that it is no longer possible to use a standard static drawing algorithm because this approach requires an algorithm which allows to compute a drawing by adjusting an existing one.

Therefore, we show how to adapt three different approaches of drawing algorithms (the force-directed, the hierarchical, and the orthogonal approach) such that they fit into our framework.

The force-directed approach fits well in our framework because the possibility to adjust a given drawing is already built in. The drawing to be adjusted is taken as initialization for the algorithm and the amount of allowed changes can be modeled using simulated annealing.

Hierarchical and orthogonal approaches work both in phases, and we have to introduce new metrics which work on the intermediate results of these phases instead of on the final drawings. If we would realize after the last phase that an adaptation is not within the predefined limits, we could not reconstruct any more which phase was responsible for that.

For the hierarchical approach we compute a global ranking for the backbone and introduce rank metrics to adjust the rankings of the individual graphs up to a given threshold. For the crossing reduction phase we introduce an adjusted sorting approach called smooth sorting which allows to restrict the number of changes resulting from the sorting. For the orthogonal approach we adjust the computation of the quasi orthogonal shape by using a parameterized version of the network flow computation.

Afterward, we present the DGD-system which implements the framework and the adjusted algorithms in Java. It is available as a stand-alone application as well as a web application. Finally, we discuss the influence of different parameters and the usefulness of our approach by mean of case studies of applications from different domains. Some of the results presented in this thesis have been published at international conferences [DGK01, DG02, GBPD04].

Ausführliche Zusammenfassung

Dynamisches Graphzeichnen hat viele Anwendungen und etabliert sich langsam als eigenständiges Teilgebiet des Graphzeichnens. Der grundlegende Unterschied zwischen statischem und dynamischem Graphzeichnen besteht darin, dass im dynamischen Fall nicht nur die ästhetischen Anforderungen an eine gute Zeichnung des Graphen zu erfüllen, sondern gleichzeitig Änderungen der "Mental Map" des Betrachters zu minimieren sind.

Diese Arbeit präsentiert ein generisches Framework für das Offline Problem des Zeichnens dynamischer Graphen: wie kann eine Sequenz gegebener Graphen, die sich über die Zeit durch Einfügen bzw. Löschen von Knoten bzw. Kanten verändert, so gezeichnet werden, dass sowohl die lokale Qualität jeder einzelnen Zeichnung als auch die dynamische Stabilität aller Zeichnungen maximiert wird.

Zuerst führen wir die Theorie und Motivation des vorausschauenden Zeichnens von Graphen ein. Dabei wird eine Projektion der Graphsequenz auf einen Graphen, den sogenannten Supergraphen, berechnet und für diesen dann mit einem beliebigen statischen Graphzeichenalgorithmus eine Zeichnung erstellt. Diese dient als Schablone, um für die Einzelgraphen der Sequenz eine induzierte Zeichnung abzuleiten. Das führt dazu, dass weder Knoten noch Kanten ihre Position in der Zeichnung ändern und eine hohe globale Stabilität gegeben ist. Allerdings muss man dafür eine Einschränkung der lokalen Qualität der einzelnen Zeichnungen in Kauf nehmen.

Um dem entgegenzuwirken, erlauben wir in unserem zweiten Ansatz – vorausschauendes Zeichnen von Graphen mit Toleranz – lokale Optimierungen der induzierten Zeichnungen bis zu einem gegebenen Grenzwert. Um zu messen, ob eine Änderung noch innerhalb der gegebenen Grenzen liegt und somit die Mental Map des Betrachters bewahrt werden kann, wenden wir Differenzmetriken an.

Durch die Wahl des Grenzwertes ist es jetzt möglich, lokale Qualität gegen globale Stabilität einzutauschen: ein niedriger Grenzwert erhöht die globale Stabilität und schränkt die lokale Qualität ein, bei einem hohen Grenzwert verhält es sich umgekehrt.

Als Verallgemeinerung des Supergraphen stellen wir das Konzept des sogenannten Backbone einer Graphsequenz vor. Dieser ist eine partielle Projektion der Sequenz von Graphen und enthält nur diejenigen Knoten, die für die Graphsequenz von großer Bedeutung sind. Die Bedeutung von Knoten wird mit Hilfe einer frei wählbaren Funktion definiert. Diese kann zum Beispiel auf statistischen Informationen beruhen (wie häufig kommt ein Knoten in der Sequenz vor) oder auch die Semantik einer Graphsequenz nutzen.

Der Vorteil des toleranzbasierten Ansatzes liegt darin, dass festgelegt werden kann, ob für eine Sequenz die Qualität der einzelnen Zeichnungen oder die dynamische Stabilität im Vordergrund steht. Der Nachteil besteht darin, dass ein Algorithmus vorausgesetzt wird, der eine gegebene Zeichnung anpassen kann. Das trifft für die meisten Standardalgorithmen des statischen Graphzeichnens nicht zu.

Deshalb wird gezeigt, wie drei verschiedene Typen von Zeichenalgorithmen so angepasst werden können, dass sie in das Framework passen. Es handelt sich dabei um den kräftebasierten, den hierarchischen und den orthogonalen Ansatz.

Der kräftebasierte Ansatz passt sehr gut in das Framework, da er iterativ arbeitet und es einfach möglich ist, eine gegebene Zeichnung anzupassen. Um zu gewährleisten, dass die Anpassung innerhalb der vorgegebenen Grenze bleibt, wird er mit dem Ansatz des Simulated Annealing kombiniert.

Der hierarchische und orthogonale Ansatz arbeiten beide in Phasen. Deshalb werden neue Metriken eingeführt, die auf den Zwischenergebnissen der Phasen (partielle Zeichnungen oder eine Gestalt) arbeiten können. Würde erst nach der letzten Phase festgestellt, dass eine Anpassung nicht innerhalb der vorgegebenen Grenzen läge, könnte nicht mehr nachvollzogen werden, welche Phase dafür verantwortlich war.

Beim hierarchischen Ansatz wird eine globale Hierarchie für den Backbone berechnet und anschließend für die einzelnen Graphen mit Hilfe einer Rangmetrik bis zu einem bestimmten Grad angepasst. Bei der Kreuzungsminimierung wird eine "weiche" Sortierung angewendet, bei der ein Parameter festlegt, wie viele Änderungen in Bezug auf die Initialisierung erlaubt sind. Beim orthogonalen Ansatz wird eine parametrisierte Netzwerkflussberechnung mit Constraints benutzt.

Anschließend stellen wir das DGD-System vor, welches das Framework und die angepassten Algorithmen in Java implementiert. Es ist sowohl als standalone Applikation als auch als Webanwendung verfügbar. Abschließend werden der Einfluss der verschiedenen Parameter und der praktische Nutzen anhand von Fallstudien aus verschiedenen Anwendungsbereichen diskutiert. Teile dieser Arbeit wurden bereits auf verschiedenen internationalen Konferenzen veröffentlicht [DGK01, DG02, GBPD04].

Acknowledgments

First of all, I would like to thank my adviser, Professor Stephan Diehl, for his strong support throughout my degree. His scientific expertise, his encouragement, and his generous funding of my research have been essential for its success. Throughout his time at Saarbrücken he was a pleasant and really humorous officemate.

Also, I would like to thank Professor Reinhard Wilhelm for the opportunity of working in his group and for providing me the possibility of attending various international workshops and conferences to meet other researchers and to present my work to them. He created a warm, and optimistic working atmosphere in his group that inspired me. Further, I would like to thank him for his strong support of my DAAD postdoctoral scholarship application.

I enjoyed working with my colleagues, but not only working also barbecuing in summer times and eating cake all over the year :)

I want to thank the members of the Graph Animation project Peter Birke, Stephan Zimmer, and especially Mathias Pohl for helpful discussions and insights. Many of the implementation work has been done by them.

My thanks go to Stephan Diehl, Mathias Pohl, Stephan Thesing, and Deanna Tinnin for proof-reading parts of this thesis.

Finally, I am indebted to my family for their affection, support, and constant encouragement throughout the years.

Contents

1 Introduction **1**
1.1 Dynamic Graph Drawing 3
1.1.1 Dynamic Drawing of a Graph 4
1.1.2 Drawing of Dynamic Graphs 4
1.2 Overview of this Thesis 5

2 Fundamental Principles **7**
2.1 Sets and Functions 7
2.2 Graphs and Drawings of Graphs 8
2.2.1 Graph Drawing Paradigms 10
2.3 Mental Map 13
2.4 Difference Metrics 14
2.4.1 Drawing Alignment 14
2.4.2 Metrics 15

3 A Framework for Offline Drawing of Dynamic Graphs **23**
3.1 Foresighted Graph Drawing 25
3.1.1 Super Graph 27
3.1.2 Graph Sequence Partition 28
3.1.3 Strategies for Computing a GSP 30
3.1.4 Reduced Graph Sequence Partition 31
3.1.5 Optimization of the Edge Routing 33
3.1.6 Algorithm 34
3.1.7 Interactively Drawing of Dynamic Graphs 35
3.2 Foresighted Graph Drawing with Tolerance 36
3.2.1 Drawing Adjustment Strategies 38
3.2.2 Requirements for the drawing algorithms 39
3.2.3 Backbone 39

4 Force-Directed Approach **43**
4.1 Force-Directed Graph Drawing 43
4.1.1 Springs and Electrical Forces 44
4.1.2 Magnetic Fields 46
4.1.3 Simulated Annealing 46
4.2 Force-Directed Drawing of Dynamic Graphs 48

5 Hierarchical Approach **51**
5.1 Hierarchical Graph Drawing 51
5.1.1 Layer Assignment 52
5.1.2 Crossing Reduction 56
5.1.3 Horizontal Coordinate Assignment 57
5.2 Hierarchical Drawing of Dynamic Graphs 57
5.2.1 Rank Assignment 58
5.2.2 Crossing Reduction 65
5.2.3 Simultaneous Drawing Adjustment 66

6 Orthogonal Approach **69**
6.1 Orthogonal Graph Drawing 69
6.1.1 The Topology-Shape-Metrics Approach 69
6.1.2 Sketch Driven Orthogonal Graph Drawing 70
6.2 Orthogonal Drawing of Dynamic Graphs 72
6.2.1 Shape Adjustment 73
6.2.2 Metrics Adjustment 74
6.2.3 Simultaneous Drawing Adjustment 75

7 The Dynamic Graph Drawing System DGD **77**
7.1 The Viewer DGDView 79
7.2 Web Application 82

8 Case Studies **85**
8.1 Foresighted Graph Drawing 85
8.1.1 Algorithm Animation 85
8.1.2 Resource Allocation 87
8.1.3 Buffered IO 88
8.2 Foresighted Graph Drawing with Tolerance 90
8.2.1 Force-directed Approach 90
8.2.2 Hierarchical Approach 92
8.2.3 Orthogonal Approach 96
8.3 Evaluation 98

9 Related Work **101**
9.1 Dynamic Drawing of a Graph 101
9.1.1 Quality Measures for Graph Animation 102
9.1.2 Naive Methods for Graph Animation 103
9.1.3 Linear Regression Analysis Method 104
9.1.4 Motion Cluster Analysis Method 104
9.2 Drawing of Dynamic Graphs 105
9.2.1 General Frameworks 105
9.2.2 Force-Directed Drawings 106
9.2.3 Hierarchical Drawings 106
9.2.4 Orthogonal Drawings 107
9.2.5 Offline Problem 108

10 Conclusions **109**
10.1 Achievements 109
10.2 Future Work 111
10.2.1 Evaluate the Effectiveness 112
10.2.2 Graph Animation 112
10.2.3 The Framework 112

Bibliography **114**

Chapter 1

Introduction

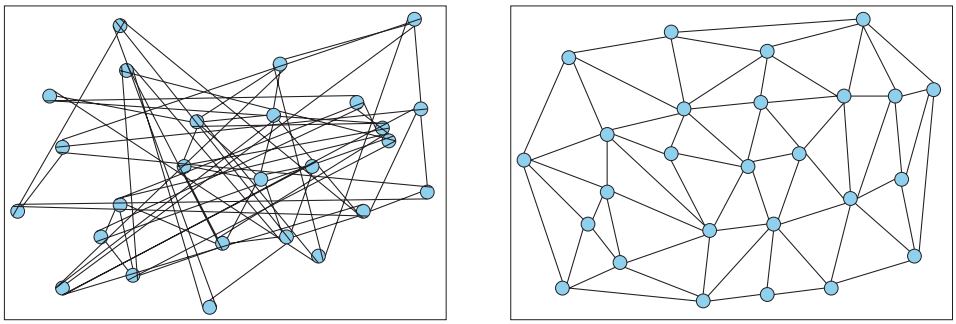
> *Optimizing the layout adjustment for a sequence of graph changes, e.g. for an off-line animation, is still an open yet very challenging area of research.*
>
> *– Jürgen Branke [Bra01b]*

Relational data is most commonly encoded in the form of a graph. A graph consists of a set of nodes representing the data items and a set of edges describing the relationships between the data items. Graphs have many applications and are, for example, used to model software systems, represent knowledge maps, or to describe communication networks.

Graph drawing deals with the visualization of relations between objects encoded in graphs: each node is usually displayed as a graphical object such as a circle, rectangle, polygon, or an image. Edges are usually drawn as straight or curved line segments connecting the nodes of the graph. The aim of graph drawing is to produce "good" visualizations of graphs, that is the drawing should reflect the information encoded in the graph as clear as possible. Thus, graph drawing is the art of conveying structured abstract data and belongs to the research area of information visualization.

In many applications a graph does not contain specific geometric information. Theoretically, when drawing such a graph, the positions of the nodes in the drawing could be chosen arbitrarily. However, for information visualization the choice of the positions of the nodes can make a large difference regarding the quality of the visualization. Figure 1.1(a) shows an example of a graph drawing – the nodes are displayed as circles and the edges are displayed as lines connecting two edges. But the drawing is of poor quality and it is hard to figure out the relationships between the nodes. Figure 1.1(b) displays the same graph as Figure 1.1(a), but this time the nodes are posi-

tioned such that no edges cross each other - in contrast to the drawing in Figure 1.1(a). This makes it easier to perceive the structure of the graph.



(a) Drawing of poor quality (b) Drawing of good quality

Figure 1.1: Two different drawings of the same graph.

Since the handmade design of appropriate drawings is a complex and expensive task, the automatic generation of graph drawings is becoming increasingly important to a variety of information visualization applications in science (especially computer science) and engineering. Examples include compilers, data bases, VLSI and network design, and software engineering.

Many general and specific algorithms for automatically drawing graphs have been developed – an overview can be found in [BETT99]. Graph drawing is a very lively and dynamic area and the results of the latest research are presented at the annual International Graph Drawing Symposium.

Most work on graph drawing addresses the problem of drawing a single, static graph. Algorithms have been developed for different classes of graphs, like undirected graphs, planar graphs, directed graphs, directed acyclic graphs, and trees. There also exist different drawing conventions. These are basic rules that the drawing must fulfill, for example a grid drawing where all nodes, edge crossings, and edge bends have to be integer coordinates, or an orthogonal drawing where all edge segments have to be drawn horizontally or vertically. Finally, there are different aesthetic criteria, like minimizing crossings and bends or maximizing symmetries, which a drawing should preserve as much as possible to achieve readability [DETT94, HMM00].

Different algorithmic approaches to draw a graph have been developed over the last decades. The most popular ones are force-directed, hierarchical, and orthogonal drawing methods. Force-directed approaches are very popular because they produce good drawings for many graphs, emphasize symmetric structures in a graph, and are easy to implement. Unfortunately they tend to be rather slow and to run into local minima. Hierarchical approaches are the most widely used algorithms for drawing layered graphs.

They produce drawings by placing nodes on layers (usually horizontal or vertical layers, but drawings with circular layers exist as well) while trying to minimize the number of edge crossings. Orthogonal approaches are popular in real life applications. For example, entity-relationship and data flow diagrams, information systems, data bases, and software engineering areas are quite often represented by orthogonal drawings.

1.1 Dynamic Graph Drawing

In many applications graphs are not static but change their structure and drawing conventions according to user and application actions. Thus, a graph is not drawn once and for all, but its drawing changes over time. There are two sources of dynamics:

- nodes or edges are dynamically added to or removed from the graph and therefore the drawing of the graph has to be updated,

- or it has to be updated due to changes of the requirements of the drawing, for example using a hierarchical approach instead of an orthogonal one, or collapsing and expanding clusters in large graphs.

Both cases have in common that the dynamics results in an additional aesthetic criterion known as "preserving the mental map" [MELS95] or dynamic stability: The human brain builds a mental map of its environment in order to be able to navigate without memorizing each detail. The mental map is a distorted and abstracted representation of the real environment. Unimportant areas tend to be collapsed to a single entity while important landmarks are overemphasized. When working with visualizations of information users build a mental map of the data which is closely linked to the particular visualization. If the visualization changes significantly due to changes in the data or the way it is represented they loose the mental map and have to rebuild it from scratch.

"Preserving the mental map" when changing the drawings of graphs has been identified to be crucial for the usability of systems for drawing of dynamic graphs [DETT99]. There are two possible approaches to this problem: either use graph drawing algorithms that try to minimize changes, or to communicate the changes in form of an animation, that is, a smooth transition from the old to the new drawing.

1.1.1 Dynamic Drawing of a Graph

Often, it is desirable to generate different drawings of the same graph, which emphasize particular aspects of the same underlying graph, for example a drawing representing the dataflow information in a telecommunication network could either emphasize the hierarchical structure of the dataflow or could emphasize which parts of the network communicate with which other parts of the network by clustering the graph appropriately.

In such a case there is the following challenge: given two different drawings of the same graph – preserve the mental map by communicating the changes. The common approach for this problem is to create a smooth transformation (called animation) between the two drawings which helps the users to maintain or quickly update their mental map. An animation is a sequence of images that is characterized by subtle but highly structured changes between consecutive frames over space and time. In the human brain these changes create the illusion of a smooth movement of the corresponding objects.

However, the standard method of simply using linear interpolation to create animations often yields animations of poor quality which are more confusing than useful to preserve the user's mental map (as an example see Figure 9.1). A superior method was introduced by Friedrich [Fri02, FE02, FH01]. He proposes to use a combination of affine transformations to build the animation between the two drawings and achieves much better results.

Dynamic drawing of a graph is not covered in this thesis, but is discussed as part of the related work in more detail in Section 9.1.

1.1.2 Drawing of Dynamic Graphs

The world of computer science is full of dynamic graphs, for example animations of graph algorithms or algorithms which work on linked data structures, dynamic visualizations of resource allocation in operating systems and project management, network connectivity and the constantly changing hyperlink structure of the world wide web.

Drawing of dynamic graphs addresses the problem of computing drawings of graphs which evolve over time because nodes or edges are dynamically added or removed. Here the challenge is to compute a drawing for each graph such that each drawing is of good quality and the successive drawings do not change too much such that the mental map can be preserved.

The ad-hoc approach to solve this problem is to compute a new drawing for the whole graph after each update using algorithms developed for static graph drawing. This approach produces drawings of good quality for each single graph, but in most cases these drawings do not preserve the mental

map at all.

To preserve the mental map between two drawings nodes which are present in both graphs should preserve some common relationships (for example proximity or orthogonality). Therefore, better approaches try to compute drawings which preserve invariant parts of the graphs also in the drawings. In this way changes in two successive drawings are reduced to a minimum and the mental map can be preserved. But, in most cases the quality of the single drawings is also reduced, because the restriction to preserve invariant parts limits the possible drawings.

In some cases all changes of the graphs are even known beforehand, for example if we want to visualize the evolution of a social network based on an email archive, or the evolution of program structures stored in software archives.

Thus, there are two variations of the problem of drawing of dynamic graphs: the online problem and the offline problem. The online problem considers only predecessor graphs of the current graph, whereas the offline problem considers predecessor as well as successor graphs, that is it considers a whole sequence of graphs, which is known beforehand.

In the latter case each graph can be drawn being fully aware of what graphs will follow – the knowledge about the future can be used to improve the quality and stability of the drawings.

Off course, it is also possible and reasonable to combine the drawing of dynamic graphs approach and the dynamic drawing of a graph approach to preserve the mental map: try to minimize the changes in two successive drawings and communicate the unavoidable changes by using an animation between the drawings.

This thesis deals only with the offline problem. So far, there exists only one other approach that takes advantage of the knowledge about the future, namely TGRIP [CKN+03]. This approach which is restricted to spring embedding (a force-directed graph drawing approach) and some approaches for the online problem are discussed in Chapter 9.

1.2 Overview of this Thesis

This thesis is organized as follows: Chapter 2 introduces the basic principles and notations of graph drawing. It contains the mathematical definitions of a graph and a drawing of a graph, followed by a description of drawing conventions and aesthetic criteria of drawings. Then, it presents the concept of the mental map and difference metrics for comparing drawings of a graph, that is to measure how much the drawings change.

Chapter 3 presents a framework for drawing sequences of graphs – where the whole sequence is known beforehand – while preserving the mental map and trading layout quality for dynamic stability (tolerance). The framework is generic in the sense that it works with different graph drawing algorithms with related metrics and adjustment strategies.

Chapter 4 treats the force-directed approach, Chapter 5 deals with the hierarchical approach and Chapter 6 with the orthogonal approach. Each of these chapters first introduces a static graph drawing approach and then shows how to adapt this approach such that it fits in the framework presented in Chapter 3.

Chapter 7 describes the implementation of our framework. A stand-alone application as well as a web service is available. Chapter 8 presents case studies in different application domains and also discusses the influence of different drawing strategies and parameters of the algorithms presented in Chapters 3 to 6.

After an overview of related work in Chapter 9, this thesis concludes with a summary and a short discussion of possible directions of future research in Chapter 10.

Some of the results presented in this thesis have been published in [DGK01, DG02, GBPD04].

Chapter 2

Fundamental Principles

The main purpose of this chapter is to introduce the mathematical principles and to define the basic concepts of graphs and graph drawings. Some basic concepts are assumed to be known (when in doubt see [KW01, BETT99, DETT94]).

Related topics are covered in many textbooks: graph theory is described in [BM76, Har72], graph algorithms are presented in [Eve79, Gib80, Meh84, NC88], and computational geometry, which also provides a good background for graph drawing methods, is described in [PS85].

Furthermore, the concept of the *mental map* and a large variety of difference metrics are introduced, which are essential to compare two graph drawings and such to decide if they are "close" enough so that the users can preserve their mental maps.

2.1 Sets and Functions

This section provides basic definitions of sets and functions, which are used in the rest of this thesis.

Definition 2.1 (Set Notations) Let X be an arbitrary set.

- X^n is the set of all n-ary tuples $(x_1, \ldots, x_n)$ with $x_i \in X$.
- X^* is an ordered list with elements of X.
- $\mathcal{P}(X)$ is the power set of X: $\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$.
- $\mathcal{P}_n(X)$ is the set of all subsets of X with cardinality n: $\mathcal{P}_n(X) = \{Y \in \mathcal{P}(X) \mid |Y| = n\}$.

- A *multiset* is a pair (X, f), where f is a function mapping X to the cardinal numbers greater than zero. For any $x \in X$, $f(x)$ is called the multiplicity of x. In other words a multiset is a set for which repeated elements are considered.

Definition 2.2 (Function Notations) Let X and Y be two arbitrary sets.

- $f : X \to Y$ is a mapping of elements of the set X to elements of the set Y.
- $X \to Y$ is the set of all mappings from X to Y.
- Let $f : X \to Y$ be a mapping. The set

$$\mathfrak{D}(f) := \{x \in X \mid \exists y \in Y : y = f(x)\}$$

  is the domain of the mapping f. If $\mathfrak{D}(f) \neq X$ then f is a partial mapping.

2.2 Graphs and Drawings of Graphs

Relational structures, consisting of a set of entities and relationships between those entities, are widely spread in computer science. Such structures are usually modeled as graphs: the entities are nodes, and the relationships are edges.

Definition 2.3 (Graph) Let V be a finite set. A *directed graph* is a tuple $G = (V, E)$ with $E \subseteq V \times V$, where V is the set of *nodes* and E the multiset of *directed edges*, that is, ordered pairs (v, w) of nodes. The directed edge (v, w) is an *outgoing edge* of v and an *incoming edge* of w. Nodes without outgoing edges are called *sinks*, nodes without incoming edges are called *sources*.
$pred(v) = \{w \in V \mid (w, v) \in E\}$ is the set of predecessor nodes of v and $succ(v) = \{w \in V \mid (v, w) \in E\}$ the set of successor nodes of v.
The cardinalities of these sets are $indeg(v) = |pred(v)|$ and $outdeg(v) = |succ(v)|$. The degree of a node v is $degree(v) = indeg(v) + outdeg(v)$.
An *undirected graph* is defined similarly to a directed graph, except that the elements of E are *undirected edges*, that is, unordered pairs $\{v, w\}$ of nodes.

The *end-nodes* of an undirected edge $e = \{v, w\}$ are v and w. The nodes v and w are *adjacent* to each other and e is *incident* to v and w. The *neighbors* of v are its adjacent nodes.

An edge (v, v) is called *self-loop*. An edge which occurs more than once in E is a *multiple edge*. A *simple graph* has no self-loops and no multiple edges. Unless otherwise specified, we assume in the following that graphs are simple.

Further, a (directed) *path* in a (directed) graph $G = (V, E)$ is a sequence $(v_1, v_2, \ldots, v_n)$ of distinct nodes of G, such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$. A (directed) path is a (directed) *cycle* if $(v_n, v_1) \in E$. A directed graph is *acyclic* if it has no directed cycles.

A graph $G' = (V', E')$, such that $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$, is a *subgraph* of graph $G = (V, E)$. If $E' = E \cap (V' \times V')$ then G' is the *induced* subgraph by V'.

An undirected graph is *connected* if for each pair $\{v, w\}$ of nodes there is a path between v and w. A directed graph $G = (V, E)$ is connected if for each pair (v, w) of nodes there is a sequence of nodes $v_0, \ldots, v_n$ with $v_0 = v, v_n = w$ and for $0 \leq i < n$ holds: $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$. A maximally connected subgraph of a graph G is a *connected component* of G.

Definition 2.4 (Drawing of a Graph) A *drawing* of a graph $G = (V, E)$ is a pair $\Gamma = (\Gamma^V, \Gamma^E)$ with $\Gamma^V : V \to \mathbb{R}^2$ and $\Gamma^E : E \to (\mathbb{R}^2)^*$ with $\Gamma^E(v, w) = \{(x_1, y_1), \ldots, (x_n, y_n)\} \Rightarrow \Gamma^V(v) = (x_1, y_1) \wedge \Gamma^V(w) = (x_n, y_n)$, where Γ^V is the mapping of the nodes and Γ^E the mapping of the edges. $\Gamma(G)$ is the set of all drawings of a graph G.

The function Γ^V assigns a point in the plane to each node and the function Γ^E maps each edge to a sequence of points. The final drawing of a graph results from the functions Γ^V, Γ^E, and the graphical interpretation of the nodes and edges. A node could be drawn as a circle or a rectangle for example, and an edge could be a polygonal chain, a straight line segment, or a curve. A directed edge is usually drawn as an arrow head at the target node.

At this point, it is important to note that a graph and its drawing are quite different objects – in general, a graph has many different drawings. Nevertheless it is common to use the same terminology for an edge (v, w) and the drawing $\Gamma^E(v, w)$.

A drawing Γ is called *planar* if no two distinct edges intersect. A graph is called *planar* if it admits a planar drawing.

Planar graphs are important in graph drawing for several reasons. First, edge crossings reduce readability (see [BPCJ95, PCJ96, Pur97]) and thus, planar graphs are really appropriate to produce nice drawings. Second, the theory of planar graphs has a long history in graph theory (see [NC88]) which can be used to simplify topological concepts.

A planar drawing partitions the plane into topologically connected regions called *faces*. The unbounded face is usually called the *external face*. A planar drawing determines a circular ordering on the neighbors of each node v according to the clockwise sequence of the incident edges around v. Two planar drawings of the same graph G are *equivalent* if they determine the same circular orderings of the neighbor sets. A *(planar) embedding* is an equivalence class of planar drawings and is described by the circular order of the neighbors of each node. An *embedded graph* is a graph with a specified embedding. A planar graph may have an exponential number of embeddings.

The *dual graph* G^* of an embedding of a planar graph G has a node for each face of G, and an edge $\{f, g\}$ between two faces f and g for each edge that is shared by f and g. In a sense, the dual graph G^* catches the combinatorial information in the embedding. If two graphs have the same embedding, then they have the same dual graph.

Figure 2.1 shows a planar graph and its dual graph.

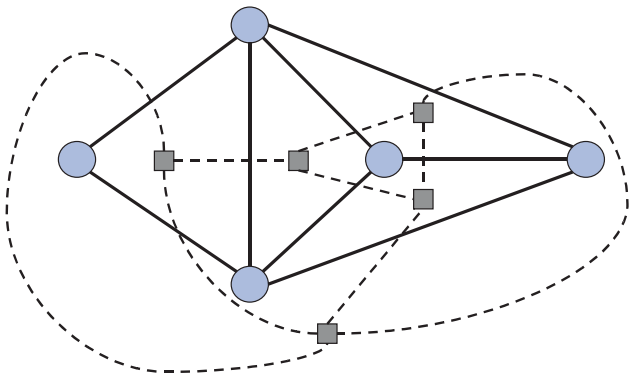


Figure 2.1: Dual graph (drawn with boxes for nodes and dashed lines for edges) for an embedding of a planar graph.

2.2.1 Graph Drawing Paradigms

This section provides an overview of two important graph drawing concepts: the drawing conventions used when drawing a graph, and the aesthetic criteria for a readable drawing.

Drawing Conventions

Drawing conventions are certain properties that the drawing must satisfy. These properties deal with the representation and placement of nodes and edges. Widely used drawing conventions are given in the list below:

- **Grid Drawing**: Nodes, crossings, and edge bends have integer coordinates.
- **Planar Drawing**: No two edges cross.
- **Polyline Drawing**: Each edge is drawn as a sequence of one or more straight lines.
- **Straight-line Drawing**: Each edge is drawn as a straight line segment.
- **Orthogonal Drawing**: Each edge is drawn as a sequence of one or more straight lines where horizontal and vertical segments alternate.
- **Upward/Downward Drawing**: For acyclic directed graphs, each edge is drawn as a curve monotonically nondecreasing/nonincreasing in the vertical direction.

Straight-line and orthogonal drawings are special cases of polyline drawings. Drawings with curved edges can be approximated using polyline drawings.

Aesthetic Criteria

Aesthetic criteria specify graphic properties of the drawing that we would like to apply, as much as possible, to achieve readability. Several criteria have been identified to be essential (see [BFN85, PCJ96, STT81]):

- **Crossings**: Minimization of the total number of crossings between edges. If too many edges cross each other, the human eye cannot easily find out which nodes are connected by an edge. Ideally, we would like to have planar drawings, but not every graph admits one.
- **Bends**: Minimization of bends is an important criterion because the human eye can more easily follow an edge with none or only few bends than an edge wildly zig-zagging through the drawing. In VLSI production, bends in wires are potential spots of trouble. We distinguish three sub criteria:
  - **Total Bends**: Minimization of the total number of bends along edges. This is especially important for orthogonal drawings.
  - **Maximum Bends**: Minimization of the maximum number of bends on an edge.

  - **Uniform Bends**: Minimization of the variance of the number of bends on the edges.

- **Edge Length**: In VLSI schematics, edges correspond to wires which carry information from one point on the chip to another. To do this fast, wires should be short. We distinguish three sub criteria:

  - **Total Edge Length**: Minimization of the sum of the lengths of the edges.
  - **Maximum Edge Length**: Minimization of the maximum length of an edge.
  - **Uniform Edge Length**: Minimization of the variance of the lengths of the edges.

- **Area**: Minimization of the area of the drawing. The area of a drawing can be defined in different ways. For example, we can define it as the bounding box, that is, the smallest rectangle with horizontal and vertical sides covering the drawing, or as the convex hull, that is, the area of the smallest convex polygon covering the drawing. The ability to construct area-efficient drawings is important in practical visualization applications, where saving screen space is of high importance. Further, it is also crucial for VLSI schematics.

- **Angular Resolution**: Maximization of the smallest angle between two edges incident on the same node. This aesthetic is especially relevant for straight-line drawings.

- **Aspect ratio**: Minimization of the aspect ratio of the drawing, that is the lengths of the sides of the bounding box of the drawing should be balanced.

- **Symmetry**: If a graph contains symmetrical information then it is important to reflect this symmetry in its drawing. Unfortunately, displaying symmetries is not an easy task.

- **Clustering**: When drawing large graphs it is necessary to cluster the nodes to reveal some of the structure of the graph.

In most cases not all of these criteria can be met in one drawing. Individual criteria might conflict or the computational complexity of optimizing all criteria at once can be too high. Most graph drawing algorithms therefore try to optimize a subset of these criteria. Attempts have been made to

classify these criteria according to how important they are for creating good graph drawings [Pur97, Pur00].

In some cases further restrictions may apply to the drawing. Edges may be allowed to only point into a specified range of directions, edges may not be allowed to cross each other, or node positions might be restricted to points on a grid. The above aesthetic criteria are naturally associated with optimization problems. However, most of these problems are computationally hard. Thus, many approximation strategies and heuristics have been devised.

Precedence Among Aesthetics Criteria

Unfortunately, aesthetic criteria often conflict with each other. Thus, tradeoffs are unavoidable. And even if they do not conflict, it is often algorithmically difficult to deal with all of them at the same time. Therefore, most graph drawing methods establish a precedence relation among aesthetics.

2.3 Mental Map

The phenomenon of "mental maps" is well studied in the areas of geography [GW86] and psychology [Den91]. The term *mental map* is commonly credited to Tolman [Tol48]. It refers to the observation that humans tend to build map-like cognitive representations of their environment including metric properties and topological relationships between landmarks. These mental maps are abstract and distorted representations of the underlying real structures. Unimportant areas tend to be collapsed to a single entity while important landmarks are overemphasized.

When working with a graph, the user builds a mental map of that graph, that is he will learn about the structure of the drawing, will learn to navigate in the drawing, and try to understand its meaning [ELMS91, MELS95].

In many applications graphs are not constant but change according to user and program actions. These changes can be visual such as changes of positions or appearance of nodes and edges, or structural such as addition or removal of nodes or edges. When the graph changes beyond a certain threshold the mental map of the user gets destroyed and has to be rebuilt.

Figure 2.2 shows an example. The diagram (a) shows the current drawing of the graph. The diagram (b) shows a drawing which might result after the insertion of the red edge and the rerun of the drawing algorithm. Diagram (c) shows a drawing after the insertion of the red edge which preserves the mental map.

To increase efficiency in graph drawing applications the number of events

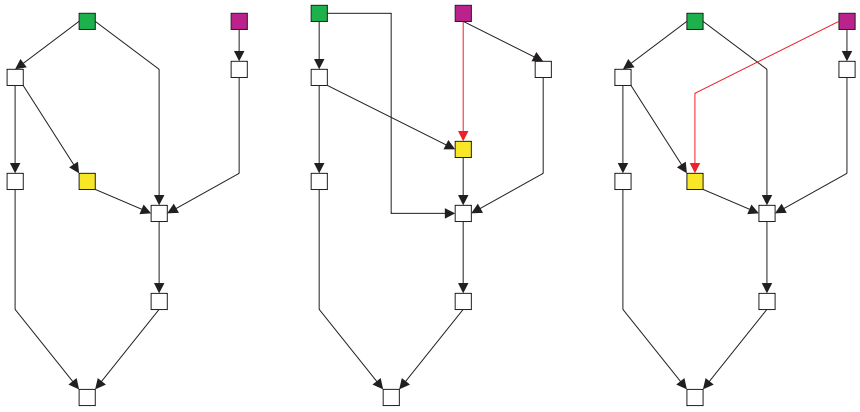


Figure 2.2: (a) Current drawing, (b) New drawing after inserting the red edge and rerunning a static drawing algorithm, (c) New drawing which preserves the mental map.

where the mental map is not preserved should be minimized. To achieve this we try to minimize the changes when changing the graph structure or computing a new layout. To be able to measure the changes of two subsequent drawings, we introduce difference metrics in the following section.

2.4 Difference Metrics

Bridgeman and Tamassia [BT98, BT01] propose a number of difference metrics for orthogonal drawings (most of them can be applied to other drawing paradigms as well). The proposed metrics fall into five categories: distance, proximity, orthogonal ordering, shape, and topology. The distance category could be considered a subset of proximity, but it is kept separate to distinguish between metrics using the Euclidean distance between points and those using relative orderings based on distance.

2.4.1 Drawing Alignment

Most of the metrics compare coordinates between drawings. Therefore, they are sensitive to the particular values of the coordinates, as illustrated in Figure 2.3. The Euclidean distance metric (see Section 2.4.2) would compute a distance of 4.25. However, translating the dark points one unit to the left and then scaling by $1/2$ in the x direction allows the points to be matched exactly, that is the distance would be 0.

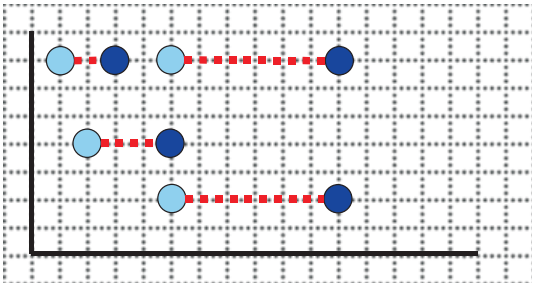


Figure 2.3: Two point sets superimposed in one drawing (corresponding points in the two sets are connected with dotted lines).

To eliminate this effect, the drawings are aligned before the metric is computed. This can be done by extracting a (matched) set of points from the drawings and then applying a point set matching algorithm to obtain the best fit – exact matches are not possible in general. In most cases the matching algorithm should take into account scaling and translation.

2.4.2 Metrics

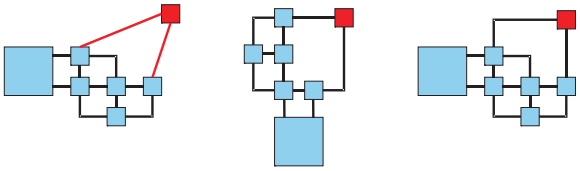
Every metric compares two drawings Γ and Γ' of the same graph G. Each object of the graph G is associated with two sets of coordinates, one describing the position in Γ, called P, and the other the position in Γ', called P'. Thus $p' \in P'$ is the corresponding point for $p \in P$ and vice versa. A matched set of objects is a set of the pairs describing the position of the object in the two drawings.

Usually, only the nodes – considered as single points without physical extension – are taken into account as objects of the graph. But, it is also possible to consider the bounding boxes of the nodes instead to achieve that the node size and shape is taken into account as well. Further, the bends of the edges could also be considered as objects of the graph.

Let Δ be a difference metric defined so that the value of the measure is 0 if the drawings are identical. To be useful, Δ should satisfy the following properties:

- Rotation: Given drawings Γ and Γ', $\Delta(\Gamma, \Gamma'_\Theta)$ should have the minimum value for the angle a user would report as giving the best match, where Γ'_Θ is Γ' rotated by an angle of Θ with respect to its original orientation (see Figure 2.4).

- Ordering: Given drawings Γ, Γ', and Γ'', $\Delta(\Gamma, \Gamma') < \Delta(\Gamma, \Gamma'')$ if and only if a user would say that Γ' is more like Γ than Γ''.



(a) (b) (c)

Figure 2.4: The rotation problem: (a) is the current graph with a modification shown in red; (b) and (c) show possible drawings of the successive graph, where (b) is mirrored and rotated. While (b) and (c) are both clearly drawings of the graph shown in (a), the similarity is more clearly seen in the properly rotated drawing (c).

- Magnitude: Given drawings Γ, Γ', and Γ'', $\Delta(\Gamma, \Gamma') = \frac{1}{c}\Delta(\Gamma, \Gamma'')$ if and only if a user would say that Γ' is c times more like Γ than Γ''.

In order to facilitate comparisons the metrics are normalized by dividing by the maximum possible value, or the lowest known upper bound if the maximum value is not known, such that all values are between 0 and 1. In the following, let $d(p, q)$ be the Euclidean distance between points p and q.

Distance Metrics

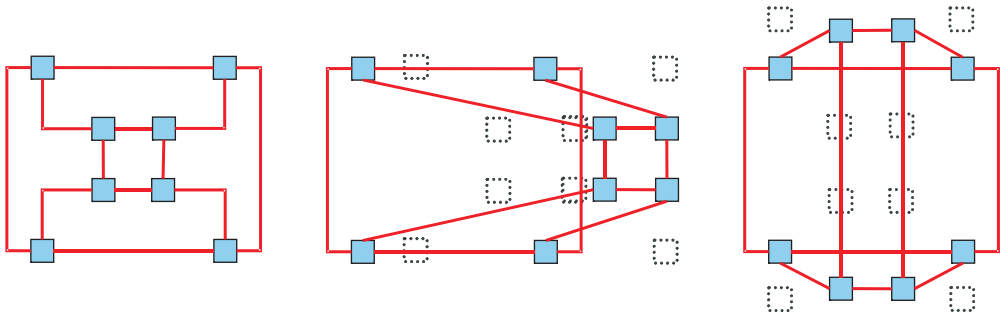
The distance metrics reflect the simple observation that the location of the points should not move too far between drawings.

Hausdorff-Distance The *Hausdorff distance* is a standard metric for determining the distance between two point sets and measures the largest distance between a point in one set and its nearest neighbor in the other. The undirected Hausdorff distance for two point sets P and P' does not take into account the fact that the point sets may be matched:

$$\text{hausdorff}(P, P') = \max\{\max_{p \in P} \min_{q' \in P'} d(p, q'), \max_{p' \in P'} \min_{q \in P} d(p', q)\}$$

The paired Hausdorff distance is an adaptation of the undirected Hausdorff distance for matched point sets, and is defined as the maximum distance between two corresponding points:

$$\text{phausdorff}(P, P') = \max_{p \in P} d(p, p')$$



(a) (b) (c)

Figure 2.5: Proximity: (b) is more similar to (a) than (c) because the relative shape of both the inner and outer squares are preserved even though the distance (using the Euclidean distance metrics) between (a) and (c) is smaller. An aligned version of the nodes of (a), used in the computation of the distance metric, is shown in dotted lines in (b) and (c).

Euclidean Distance The *Euclidean distance* is a simple metric measuring the average distance moved by each point from the first drawing to the second.

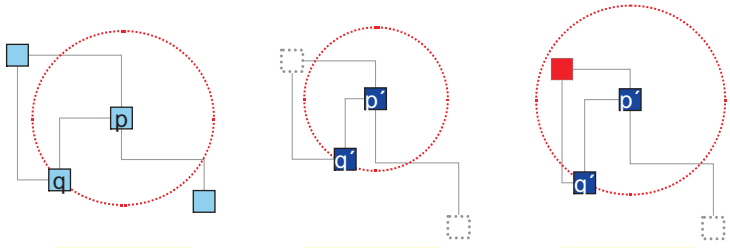
$$\text{dist}(P, P') = \frac{1}{|P|} \sum_{p \in P} d(p, p')$$

Neighborhood Metrics

The proximity metrics reflect the idea that points near each other in the first drawing should remain near each other in the second drawing. This is stronger than the distance metrics because it captures the idea that if a subgraph moves relative to another without any changes within either subgraph the distance should be less than if each point in one of the subgraph moves in a different direction (see Figure 2.5).

Nearest Neighbor Within *Nearest neighbor within* is based on the idea that if q is the closest point to p in Γ, then q' should be closest point to p' in Γ' (see Figure 2.6). Considering only distances within a single drawing means that nearest neighbor within is alignment-independent.

This metric has two versions, weighted and unweighted. In the weighted version the number of points closer to p' than q' is considered, whereas in the unweighted version it only matters if q' is or is not the closest point. The reasoning behind the weighted version is that if there are more points



(a) Drawing Γ (b) Drawing Γ' (c) Drawing Γ''

Figure 2.6: Nearest neighbor within metric.

between p' and q', the visual linking between p' and q' has been disrupted to a greater degree.

In both cases the distance is scaled by the number of points being considered. Let $nn(p)$ be the nearest neighbor of p in the point set of p and $nn(p)'$ be the corresponding point in P' to $nn(p)$.

$$\text{nnw}(P, P') = \frac{1}{\text{UB}} \sum_{p \in P} \text{weight}(\text{nearer}(p))$$

where

$$\text{nearer}(p) = \{q \mid d(p', q') < d(p', nn(p)'), q \in P, q \neq p, q \neq nn(p)\}$$

Unweighted

$$\begin{aligned} \text{weight}(S) &= \begin{cases} 0 & \text{if } |S| = 0 \\ 1 & \text{otherwise} \end{cases} \\ \text{UB}(n) &= |P| \end{aligned}$$

Weighted

$$\begin{aligned} \text{weight}(S) &= |S| \\ \text{UB}(n) &= |P|(|P| - 1) \end{aligned}$$

Nearest Neighbor Between *Nearest neighbor between* is similar to nearest neighbor within but instead measures whether or not p' is the closest of the points in Γ' to p when the two drawings have been aligned (see Figure 2.7).

$$\text{nnb}(P, P') = \frac{1}{\text{UB}} \sum_{p \in P} \text{weight}(\text{nearer}(p))$$

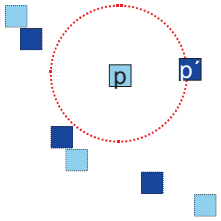


Figure 2.7: Nearest neighbor between metric.

where

$$\text{nearer}(p) = \{q \mid d(p,q') < d(p,p'), q \in P, q \neq p\}$$

Unweighted

$$\begin{aligned} \text{weight}(S) &= \begin{cases} 0 & \text{if } |S| = 0 \\ 1 & \text{otherwise} \end{cases} \\ \text{UB}(n) &= |P| \end{aligned}$$

Weighted

$$\begin{aligned} \text{weight}(S) &= |S| \\ \text{UB}(n) &= |P|(|P|-1) \end{aligned}$$

Orthogonal Metrics

The *orthogonal ordering* metrics reflects the desire to preserve the relative ordering of every pair of points – if p is northeast of q in Γ, p' should remain to the northeast of q' in Γ'. The simplest measurement of difference in the orthogonal ordering is to take the angle between the vectors $\overrightarrow{qp}$ and $\overrightarrow{q'p'}$ (constant weighted orthogonal ordering). This has the nice feature that if q is far away from p, $\overrightarrow{q'q}$ must be larger to result in the same angular move, which reflects the intuition that the relative position of points close to each other is more important than the relative position of points far away from each other (see Figure 2.9).

In the linear-weighted version changes in the north, south, east, and west relationships are weighted more heavily than changes in angle which do not affect this relationship (see Figure 2.8).

Let θ_{pq} be the counterclockwise angle between the positive x-axis and the vector $q - p$.

$$\text{order}(P, P') = \frac{1}{W} \min\{\int_{\theta_{pq}}^{\theta_{p'q'}} \text{weight}(\theta)\, d\theta, \int_{\theta_{p'q'}}^{\theta_{pq}} \text{weight}(\theta)\, d\theta\}$$

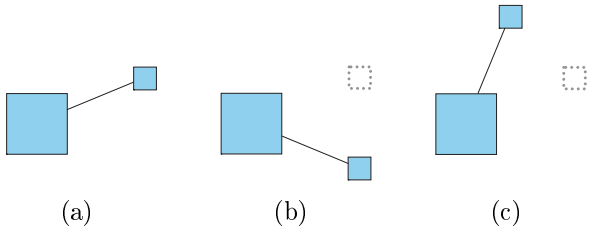


Figure 2.8: Orthogonal ordering: Even though the angle the node moves relative to the center of the large node is the same from (a) to (b) and from (a) to (c), the perceptual difference between (a) and (c) is much greater. The original location of the node is shown with a dotted box in (b) and (c) for purpose of comparison.

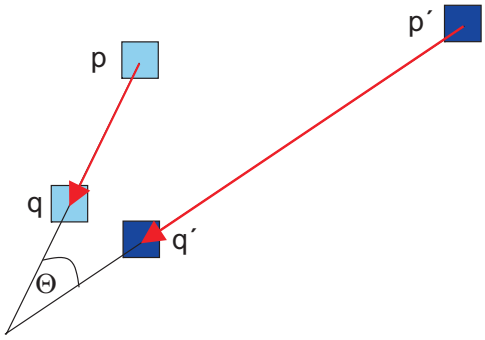


Figure 2.9: Orthogonal relationships.

Constant-weighted

$$\begin{aligned} \text{weight}(\theta) &= 1 \\ W &= 2\pi \end{aligned}$$

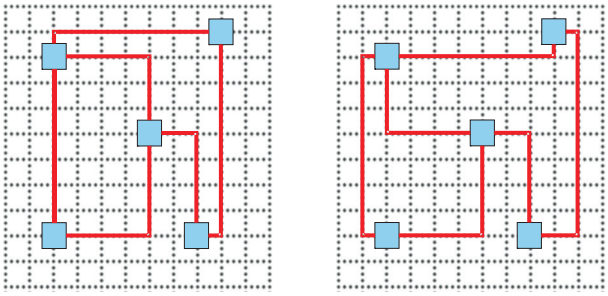
Linear-Weighted

$$\begin{aligned} \text{weight}(\theta) &= \begin{cases} \frac{(\theta \bmod \pi/2)}{\pi/4} & \text{if } (\theta \bmod \pi/2) < \pi/4 \\ \frac{\pi/2-(\theta \bmod \pi/2)}{\pi/4} & \text{otherwise} \end{cases} \\ W &= \pi/2 \end{aligned}$$

Shape Metrics

The metrics introduced so far only considered node positions. The *shape* metrics is motivated by the reasoning that edge routing may have an effect on the overall look of the graph (see Figure 2.10). The shape of an edge is the sequence of directions (north, south, east and west) traveled when

traversing the edge from source to sink. The *shape string* is the sequence of the characters N, S, E, and W. The edge of Figure 2.11, for example, has the shape string NENW.



(a) (b)

Figure 2.10: Shape metric: (a) and (b) look different even though the graphs are the same and the nodes have the same positions.

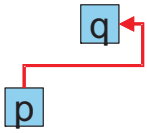


Figure 2.11: Example for a shape string.

For non-orthogonal edges the predominant direction is chosen. For each edge the minimum number of edits to transform the shape string in one drawing to the string in the other is computed, where an edit consists of inserting, deleting, or replacing a character in the shape string. The metric is the average number of edits per edge.

$$\text{shape} = \frac{1}{\text{UB}} \sum_{e \in E} \text{edits}(\text{shapestring}(e), \text{shapestring}(e'))$$

The edit distance is not normalized for the length of the sequence, and the upper bound is as follows:

$$\text{UB} = \sum_{e \in E} |\text{length}(e) - \text{length}(e')| + \min\{\text{length}(e), \text{length}(e')\}$$

where $\text{length}(e) = |\text{shapestring}(e)|$.

Topology

The topology metric reflects the idea that preserving the order of edges around a node is important in preserving the mental map. The topology of a graph is represented by its dual graph. Thus, maintaining topology just means maintaining the dual graph.

Further Comments

So far, the metrics compare two drawings of the same graph. However we also would like to compare drawings of different graphs. To this end we take only the nodes (and edges if appropriate for the metric) of the intersection of both graphs into account, because nodes only contained in one of the graphs are not comparable.

Chapter 3

A Framework for Offline Drawing of Dynamic Graphs

In this chapter we introduce a generic framework for drawing a sequence of evolving graphs which is known beforehand. The framework is generic because it works with different graph drawing algorithms. The adaptation of force-directed, hierarchical, and orthogonal drawing algorithms to fit in this framework is given in Chapters 4 to 6.

First we present the concept of the mental distance between two drawings of two graphs. This distance indicates how difficult it is for users to adopt their mental map from one drawing to the other one, that is a small mental distance means that users can easily preserve or adopt their mental maps whereas a large mental distance means that the mental map of the users cannot be preserved. Or in other words, the mental distance is a metric that indicates how close the mental maps of two given drawings are.

Definition 3.1 (Mental Distance) Let $\Gamma_1 \in \Gamma(G_1)$ be a drawing of graph G_1 and $\Gamma_2 \in \Gamma(G_2)$ a drawing of graph G_2. Then the function $\Delta : \Gamma(G_1) \times \Gamma(G_2) \rightarrow \mathbb{R}_0^+$ is a metric for how good Γ_2 preserves the mental map of Γ_1 and is called *mental distance* between Γ_1 and Γ_2. In particular $\Delta(\Gamma_1, \Gamma_2) = 0$ means that Γ_1 and Γ_2 have the same mental map.

The metrics presented in Section 2.4.2 can be used to compute mental distances between two drawings.

Assume that the quality of a drawing could be measured by a function Ψ regarding aesthetic goals like compactness, even distribution of nodes, minimal number of edge crossings, etc [PCJ96]. Such formal criteria are the computational crutches to substitute real models of human cognition or simply taste.

Definition 3.2 (Quality of a Drawing) Let $\Gamma \in \Gamma(G)$ be a drawing of graph G. Then the function $\Psi : \Gamma(G) \to \mathbb{R}_0^+$ is a metric for the quality of a drawing. In particular $\Psi(\Gamma) = 0$ means that Γ has minimal quality.

Now we can state the problem of offline drawing of dynamic graphs. In most applications the graphs are given as a sequence, in which the graphs will result from changes to the preceding graph and thus will share some nodes and edges.

Definition 3.3 (The Offline Drawing of Dynamic Graphs Problem) Given a sequence of n graphs $G_1, \ldots, G_n$. Compute drawings $\Gamma_1, \ldots, \Gamma_n$ for these graphs such that

1. $\overline{\Delta} = \sum_{1 \leq i < n} \Delta(\Gamma_i, \Gamma_{i+1})$ is minimal
2. $\overline{\Psi} = \sum_{1 \leq i \leq n} \Psi(\Gamma_i)$ is maximal

Thus, drawing of dynamic graphs is an optimization problem with two objective functions. The first one pursues the global goal to preserve the mental map by minimizing the mental distances between two successive drawings over the whole sequence. The second objective function pursues a local goal, namely to maximize the quality of each single drawing.

Just as a side note: the online drawing of dynamic graphs problem is stated as follows: Given $\Gamma_1, \ldots \Gamma_{n-1}$ and G_n compute Γ_n. In Chapter 9 some approaches to the online problem are discussed.

Unfortunately, the two optimization goals of the offline drawing problem conflict with each other and in general cannot be achieved at the same time. Figure 3.1 illustrates this fact: if we want to achieve a high stability, that is a small mental distance $\overline{\Delta}$, the quality of the drawings is decreased, because there are too many restrictions. In the other case, if we want to achieve a high quality $\overline{\Psi}$ of the drawings, the mental map cannot be preserved, because the high quality causes too many changes in successive drawings.

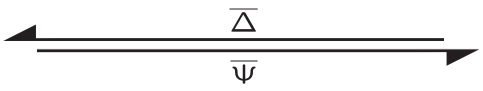


Figure 3.1: Conflicting optimization goals.

In the remainder of this chapter we present a framework for drawing algorithms for dynamic graphs which provides the possibility to trade local quality for global stability.

But first, we discuss the extreme approaches by only taking one of the two optimization functions into account.

- The first, called the ad-hoc approach, ignores the dynamic stability and computes drawings of highest quality for each graph of the sequence by using an algorithm for static graph drawing. In most cases this approach produces drawings which do not preserve the mental map at all.

  In the worst case it is even possible that two successive graphs, which are equal, have completely different drawings. For example, some spring embedder algorithms assign random initial positions to the nodes and therefore could produce different drawings of the same graph.

  Thus, this approach is not suitable for drawing of dynamic graphs.

- The second approach, called foresighted graph drawing, achieves to preserve the mental map by impossing strong restrictions to the drawings: nodes do not change their positions in successive drawings. This approach is presented in Section 3.1 in detail.

In Section 3.2 the foresighted drawing approach is extended such that it is controlled by a tolerance threshold: trade quality for stability.

3.1 Foresighted Graph Drawing

In this section we present the foresighted drawing approach. This approach computes a global drawing of a whole sequence of graphs using an arbitrary static graph drawing algorithm. Thus, the algorithm is generic with respect to the static graph drawing algorithm. A global drawing induces a drawing for each graph of the sequence. A unique feature of this approach is that once they are drawn neither nodes nor the bends of edges change their positions in graphs subsequently drawn. Using static graph drawing algorithms, which accept fixed node positions as an additional input, it is also possible that only the bends change their positions. The algorithm is called *foresighted graph drawing* because it knows the future of the current graph, that is the modifications that are applied to transform the graph to the subsequent ones.

Figure 3.2 displays the construction of a finite automaton as a sequence of drawings computed with the foresighted drawing approach: the nodes are immediately placed where they will be in the final drawing. Therefore they do not have to change their positions in the drawings.

In the following we consider graphs with multi-edges. For this we extend the definition of a graph by adding an unique identifier to each edge.

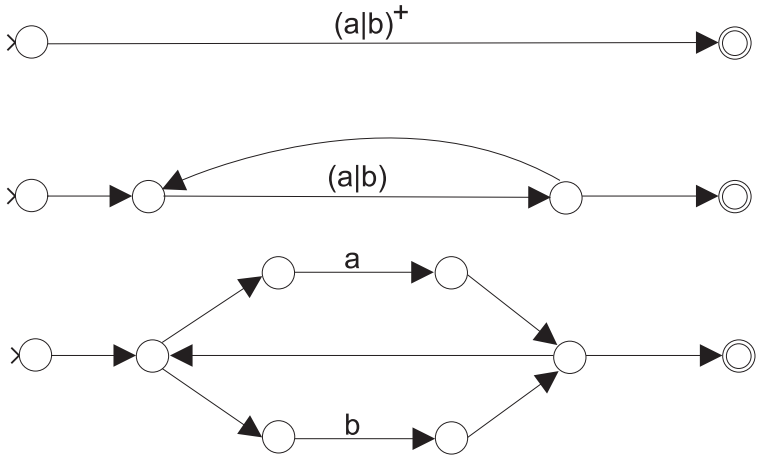


Figure 3.2: Foresighted Graph Drawing.

Definition 3.4 (Multi-graph) A *multi-graph* is a tuple $G = (V, E)$, where V is a set of nodes. Elements of the set E consist of a pair of nodes and an identifier. If the pairs are ordered, G is a *directed multi-graph* and $E \subseteq V \times V \times \mathrm{Id}$, where Id is a set of identifiers and the elements $e = (v, w, id)$ are called *directed edges*. If the pairs are unordered, G is an *undirected multi-graph* and $E \subseteq \mathcal{P}_2(V) \times Id$, where the elements $e = (\{v, w\}, id) \in E$ are called *undirected edges*. Furthermore, for directed multi-graphs the following holds: $\forall (v, w, id_1), (v', w', id_2) \in E : id_1 = id_2 \Rightarrow v = v'$ and $w = w'$. And for undirected multi-graphs it analogously holds: $\forall (\{v, w\}, id_1), (\{v', w'\}, id_2) \in E : id_1 = id_2 \Rightarrow \{v, w\} = \{v', w'\}$.

The last condition of Definition 3.4 assures that two different edges have different identifiers. All other notations concerning a graph introduced in Chapter 2 apply analogously.

In the following we consider sequences of graphs, in which a graph results from modifications (adding or deleting nodes and edges) of its preceding graph. Usually subsequent graphs share some nodes and edges. But in the worst case each graph can consist of totally different nodes and edges.

Definition 3.5 (Consistent Graph Sequence) A *consistent graph sequence* $S = [G_1, \ldots, G_n]$ is a sequence of graphs with $G_i = (V_i, E_i)$ and $\forall (v, w, id_1) \in E_p, (v', w', id_2) \in E_r$ with $1 \leq p \leq n$ and $1 \leq r \leq n : id_1 = id_2 \Rightarrow v = v'$ and $w = w'$.

The restriction in this definition ensures that edge identifiers are used consistently in all graphs, that is an edge is adjacent to the same nodes in all graphs it occurs in.

Now, we state a generic algorithm which computes a foresighted drawing of a consistent graph sequence.

Algorithm 1 Generic Foresighted Graph Drawing
foresightedDrawing($S = [G_1, \ldots, G_n]$,staticDrawing(),projectGraphs())

```
(G = (V, E), map_V : ⋃_i V_i → V, map_E : ⋃_i E_i → E) := projectGraphs(S)
drawing:= staticDrawing(G)
for i := 1 to n do
  drawGraph(G_i,drawing,map_V,map_E)
end for
```

The input of Algorithm 1 consists of a consistent graph sequence $S = [G_1, \ldots, G_n]$, with $G_i = (V_i, E_i)$, a function staticDrawing() which computes a static drawing of a graph, and a function projectGraphs() which projects a sequence of graphs to one single graph and additionally provides corresponding mappings for the nodes and edges. Using this function the algorithm computes the projection G of S as well as the mapping of the nodes map_V and the mapping of the edges map_E. Then, the function staticDrawing() computes a static drawing of the projection G, that is a data structure drawing that contains the positions of the nodes of V and the curves of the edges of E. The function drawGraph() draws each graph of the sequence by adopting the corresponding drawing information out of the data structure drawing using the mapping map_V for the nodes and the mapping map_E for the edges.

In the following sections we will have a look at some concrete functions to compute the projection G as well as the mapping functions map_V and map_E.

3.1.1 Super Graph

Now, we define the super graph of a consistent sequence of graphs and use it to compute a projection.

Definition 3.6 (Super Graph) Let $S = [G_1, \ldots, G_n]$ be a consistent graph sequence with $G_i = (V_i, E_i)$, then the graph $\widehat{G} = (\widehat{V}, \widehat{E})$ with

$$\widehat{V} = \bigcup_{i=1}^{n} V_i \quad \text{and} \quad \widehat{E} = \bigcup_{i=1}^{n} E_i$$

is called *super graph* of the sequence S.

The super graph is the union of all graphs of the sequence. Using the super graph to compute a foresighted drawing with Algorithm 1, the function projectGraphs() is defined as follows:

- $G = \widehat{G}$
- map_V is the identity
- map_E is the identity

The Super Graph and the Mental Map

Using the super graph as a projection preserves the mental map very well, as all nodes and edges occurring in the graph sequence have their own place in the drawing which never changes. If a node occurs in a graph of the sequence, it is drawn at "its own" place, if it does not occur in a graph of the sequence, "its" place remains empty. The same analogously holds for the edges. Unfortunately, the aesthetical criteria of a static drawing are not well satisfied, as the super graph of a long sequence can become really large although all single graphs of the sequence are small. This is the case if the single graphs of the sequence contain many different nodes and edges. The result is that the drawing of each individual graph of the sequence contains much unused space. This reduces the clarity and the aesthetic criterion of compactness is not satisfied.

This problem is avoidable by the following observation: nodes never occurring in a same graph of the sequence can share the same position in the drawing. To achieve this goal we construct a smaller graph on the basis of the super graph by taking into account the life times of nodes and edges.

Definition 3.7 (Life Time) Let $S = [G_1, \ldots, G_n]$ with $G_i = (V_i, E_i)$ be a consistent graph sequence and $\widehat{G} = (\widehat{V}, \widehat{E})$ the corresponding super graph. Then $T(v) = \{i \mid v \in V_i\}$ are the *life times* of the nodes $v \in \widehat{V}$ and $T(id) = \{i \mid (v, w, id) \in E_i\}$ are the life times of the edge identified by id.

The life time of a node is the set of points in time at which the node is contained in the sequence of graphs, that is the set of indices of the graphs of the sequence which contain the node. The same analogously holds for the life time of an edge. The life time has not to consist of successive points in time, but can contain breaks or can even consist only of single, not continuous points in time.

3.1.2 Graph Sequence Partition

This section describes how the super graph can be transformed into a more compact graph by bundling nodes with disjoint life times together. First we introduce the term partition.

Definition 3.8 (Partition) A *partition* of a finite set S is defined as a system $\{S_1, \ldots, S_r\}$ of non empty, pairwise disjoint subsets of S, whose union is S: $S_i \cap S_j = \emptyset \; \forall 1 \leq i < j \leq r$ and $\bigcup_{i=1}^{r} S_i = S$. A *trivial partition* is the system $\{\{x\} |\, x \in S\}$ that consist of all singletons of S.

Now, we build a graph partition out of the set of nodes of the super graph.

Definition 3.9 (Graph Partition) Let $G = (V, E)$ be a graph and $\widetilde{V} \subseteq \mathcal{P}(V)$ and $\widetilde{E} \subseteq \widetilde{V} \times \widetilde{V} \times \mathrm{Id}$. A graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ is a *graph partition* of G if and only if $\widetilde{V}$ is a partition of V and $(\widetilde{v}, \widetilde{w}, id) \in \widetilde{E} \Leftrightarrow \exists v \in \widetilde{v}$ and $w \in \widetilde{w} : (v, w, id) \in E$. We call $\widetilde{E}$ the set of edges induced by $\widetilde{V}$.

In other words, each node in $\widetilde{V}$ represents one or more nodes of V and all edges between two nodes in V are converted into edges between the representatives of the two nodes in $\widetilde{V}$.

If a graph partition of a super graph fulfills the condition that each set of the partition contains nodes with disjoint life times, it is named a graph sequence partition.

Definition 3.10 (Graph Sequence Partition GSP)
Let $S = [G_1, \ldots, G_n]$ with $G_i = (V_i, E_i)$ be a consistent graph sequence and $\widehat{G} = (\widehat{V}, \widehat{E})$ be the super graph of S. A graph partition $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ of $\widehat{G}$ where $\widetilde{V} = \{P_1, \ldots, P_k\}$ is a *graph sequence partition* of S if and only if $\forall v, w \in P_i$ with $v \neq w \Rightarrow T(v) \cap T(w) = \emptyset$. We call $\tilde{G}$ a minimal GSP of S, if there exists no GSP of S with less nodes.

In a GSP nodes with disjoint life times are bundled together. Unfortunately, the problem of computing a minimal GSP (hence minimal graph sequence partition problem mGSPP) is $\mathcal{NP}$-complete. The proof of the $\mathcal{NP}$-completeness of mGSPP and mRGSPP (see Section 3.1.4) by reduction on the minimal graph coloring problem [GJ79] is given in [DGK00].

Now we present an algorithm which computes a GSP.

Algorithm 2 Computing a GSP

$W := \widehat{V}, P := \emptyset, p := 0$
while $v \in W$ **do**
  if $\exists j : T(v) \cap T(P_j) = \emptyset$ **then**
    $P_j := P_j \cup \{v\}, T(P_j) := T(P_j) \cup T(v)$
  else
    $p := p + 1, P_p := \{v\}, T(P_p) := T(v)$
  end if
  $W := W \setminus \{v\}$
end while

Algorithm 2 intentionally leaves open how the set P_j is chosen. We will introduce different strategies for this in Section 3.1.3. In general, Algorithm 2 does not compute a minimal GSP, but we will see in the next section that minimal GSPs are not always desirable depending on which aesthetic criteria should be optimized. If a minimal partition has to be computed this can be done by using an algorithm solving the exact graph coloring problem.

Using a GSP to compute a foresighted drawing with Algorithm 1, the function projectGraphs() is defined as follows:

- $G = \widetilde{G}$
- map_V maps each node $v \in \bigcup_i V_i$ to the node of $\widetilde{V}$ that represents v
- map_E maps each edge $e \in \bigcup_i E_i$ to the edge of $\widetilde{E}$ that represents e

GSP and the Mental Map

The drawings resulting from using a GSP as a projection instead of the super graph better fulfill the aesthetical criterion of compactness, as the drawings are more compact because of the bundling of the nodes and they do not contain as much unused space. But the mental map is not preserved as well, because now several nodes share the same position in different drawings and this could confuse the viewer of the sequence of drawings.

3.1.3 Strategies for Computing a GSP

This section introduces different strategies to compute a GSP. In general Algorithm 2 does not compute a minimal GSP. From an aesthetical point of view it is not too bad that we do not compute minimal GSP's. A minimal

GSP is often not the best choice as we pay for the minimal number of nodes by an increased number of edge crossings in most cases. A GSP can be computed using one of the following strategies:

1. Search the list from P_1 to P_p or from P_p to P_1. In this case the GSP is built randomly.

2. Add v to the set with the smallest cardinality.
   This strategy tries to compute partitions of equal size and such to obtain an even distribution of nodes.

3. Add v to the set, whose life time has the smallest cardinality.
   This strategy tries to achieve that the positions of all nodes are occupied the same amount of time.

4. Only allow a limited number of nodes in a partition. If there is no partition with less nodes, then create a new set.
   This strategy prevents that too many nodes share the same position.

5. Only allow a limited number of edges in a partition. If there is no partition with less edges, then create a new set. This strategy prevents that too many edge crossings arise.

6. Give priority to nodes with adjacent edges to the same (already computed) partitions. This strategy prevents also that too many edge crossings arise.

These strategies can also be combined.

3.1.4 Reduced Graph Sequence Partition

In a GSP the number of nodes of the super graph of a consistent graph sequence is reduced. In a similar way, the number of edges in a GSP can be reduced.

Definition 3.11 (Reduced Graph Sequence Partition RGSP)
Let $S = [G_1, \ldots, G_n]$ with $G_i = (V_i, E_i)$ be a consistent graph sequence, $\widehat{G} = (\widehat{V}, \widehat{E})$ be the super graph of S and $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ be a GSP of $\widehat{G}$. The graph $\overline{G} = (\widetilde{V}, \overline{E})$, where $\overline{E} \subseteq \widetilde{V} \times \widetilde{V} \times \mathcal{P}(\mathrm{Id})$, is a *reduced GSP*, if and only if $\overline{E} = \{(\widetilde{s}, \widetilde{u}, P_1), \ldots, (\widetilde{v}, \widetilde{w}, P_k)\}$ with $\widetilde{s}, \widetilde{u}, \widetilde{v}, \widetilde{w} \in \widetilde{V} \Rightarrow \{P_1, \ldots P_k\}$ is a partition of the identifiers of the edges of $\widetilde{E}$ and $\forall(\widetilde{v}, \widetilde{w}, \{m_1, \ldots, m_k\}) \in \overline{E}$ the following holds: $(\widetilde{v}, \widetilde{w}, m_i), (\widetilde{v}, \widetilde{w}, m_j) \in \widetilde{E} : T(m_i) \cap T(m_j) = \emptyset$ for $1 \leq i < j \leq k$.

We call $\overline{G}$ a minimal RGSP of S, if there exists no RGSP of S with less edges.

A RGSP bundles multi-edges of the GSP together, that is edges of the GSP which connect the same nodes, if they have disjoint life times. Therefore, only multi-edges which result from building the super graph or the GSP can be bundled together, because other multi-edges which existed beforehand have the same life times. An edge $(\widetilde{v_1}, \widetilde{v_2}, \{m_1, \ldots, m_k\})$ of the RGSP represents k edges which exist at different times, that is in different graphs of the graph sequence, between a node in $\widetilde{v}$ and $\widetilde{w}$. Also the problem of computing a minimal RGSP (hence minimal reduced graph sequence partition problem mRGSPP) is $\mathcal{NP}$-complete.

As computing minimal RGSPs is $\mathcal{NP}$-complete, we present an algorithm which does not compute minimal RGSPs, but yields good results in practice, that is RGSPs with small numbers of edges. The algorithm actually computes a partition of identifiers of the edges of a RGSP.

Algorithm 3 Computing a RGSP

$W := \{m_1, \ldots, m_k\}$, i.e. the set of all identifiers occurring in $\widetilde{E}$
$P := \emptyset, p := 0$
while $n \in W$ **do**
  Let $(\widetilde{v}, \widetilde{w}, n)$ be the edge identified by n
  $p := p + 1, P_p := \{n\}, T(P_p) := T(n)$
  while $\exists m \in W : (\widetilde{v}, \widetilde{w}, m) \in \widetilde{E}$ and $T(P_p) \cap T(m) = \emptyset$ **do**
    $P_p := P_p \cup \{m\}, T(P_p) := T(P_p) \cup T(m), W := W \setminus \{m\}$
  end while
  $W := W \setminus \{n\}$
end while

If only the edges but not the nodes of a super graph are to be bundled together it is possible to compute a RGSP 'directly' out of the super graph. For that purpose a trivial GSP is computed, at which the partition consists of nothing but singletons - the nodes of the super graph - and this GSP is reduced afterward to a RGSP.

Using a RGSP to compute a foresighted drawing with Algorithm 1, the function projectGraphs() is defined as follows:

- $G = \overline{G}$
- map_V maps each node $v \in \bigcup_i V_i$ to the node of $\widetilde{V}$ that represents v

- map_E maps each edge $e \in \bigcup_i E_i$ to the edge of $\overline{E}$ that represents e

RGSP and the Mental Map

The drawings resulting from using a RGSP as a projection instead of a GSP again better fulfill the aesthetical criterion of compactness, as the drawings are even more compact due to the bundling of the edges. But the mental map is not preserved as well, because now several edges share a common curve in the drawing and this could confuse the viewers of the drawing.

3.1.5 Optimization of the Edge Routing

Since the drawing of each individual graph of the sequence is derived from the super graph, the GSP, or the RGSP it is possible that the edge routing is not optimal. An edge could for example be drawn as a curve around a node that is not contained in the current graph of the sequence, but will be contained in a following graph. This can be avoided by optimizing the edge routing of the individual graphs.

For this purpose the static drawing algorithm has to provide a function which takes a graph with fixed node positions as input and computes a drawing for the edges. Then, it is possible to compute the drawing of the super graph, of the GSP, or of the RGSP, where only the node positions matter. Now, the positions of the nodes are derived from this drawing for each graph of the sequence and an optimal drawing for the edges is computed.

Optimized Edge Routing and the Mental Map

By optimizing the edge routing the aesthetical criteria of the drawings of the graphs of the sequence are better fulfilled, but at the same time the mental map is not preserved as well, because the drawing of the edges changes over time. Therefore the viewer of the drawing might find it more difficult to identify the edges in the individual graphs.

3.1.6 Algorithm

The following generic algorithm computes a drawing of a sequence of graphs. It uses the super graph, the GSP, and the RGSP in combination with a static graph drawing algorithm to draw a sequence of graphs while preserving the mental map, mode $\subseteq \{GSP, RGSP, \text{optEdges}\}$.

Algorithm 4 Compact Foresighted Graph Drawing
foresightedDrawing($S = [G_1, \ldots, G_n]$,staticDrawing(),mode)

```
Ĝ := buildSuperGraph(G_1, ..., G_n)
if GSP ∈ mode then
  G̃ := computeGSP(Ĝ)
else
  G̃ := trivialGSP(Ĝ)
end if
if RGSP ∈ mode then
  Ḡ := computeRGSP(G̃)
else
  Ḡ := trivialRGSP(G̃)
end if
globalDrawing:= staticDrawing(Ḡ)
for i := 1 to n do
  currentDrawing:= extractDrawing(G_i,globalDrawing)
  if optEdges ∈ mode then
    drawing := optimizeEdges(currentDrawing)
  else
    drawing := currentDrawing
  end if
  drawGraph(G_i,drawing)
end for
```

The function extractDrawing() computes the corresponding part of the drawing of the current graph G_i of the sequence out of the global drawing. The function optimizeEdges() optimizes the drawing of the edges of a given drawing, where the position of the nodes are unchanged.

Short Summary and Remark

In this section we introduced the theory of foresighted graph drawing and investigated the influence of each single step on the mental map and the aesthetic criteria. Figure 3.3 gives again an overview.

Foresighted graph drawing does not work for all classes of graphs equally well, as it requires that the super graph, GSP or RGSP of the graphs belongs to the same class as the individual graphs. For example, in general the super graph of trees is not a tree, and the super graph of planar graphs is not a planar graph.

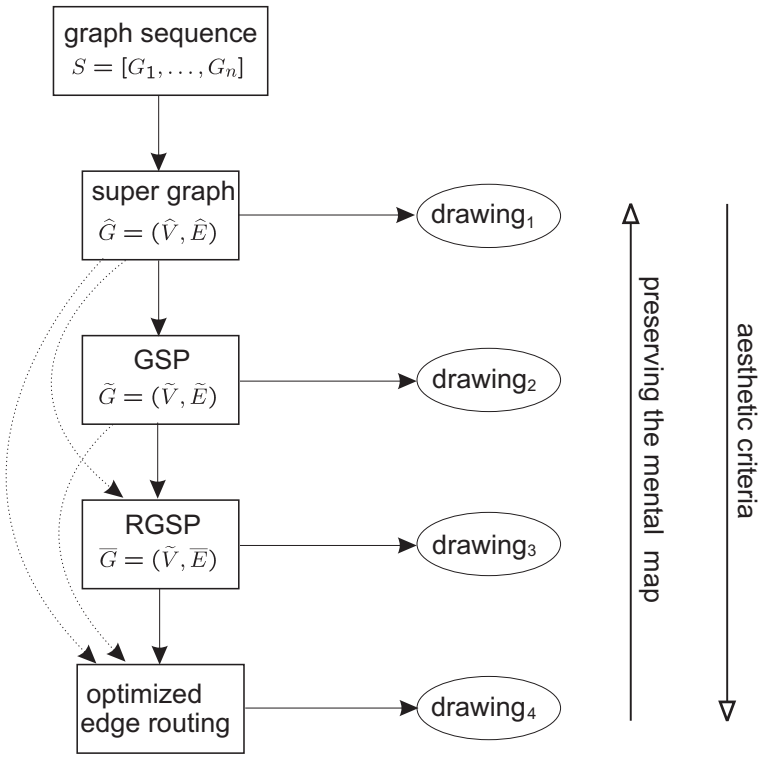


Figure 3.3: Correlation between foresighted drawing, mental map, and aesthetic criteria.

3.1.7 Interactively Drawing of Dynamic Graphs

In some applications the future of a graph depends on user input. Nevertheless between such points in time when the user interacts with the application, the program can perform several "foreseeable" changes of the graph. Thus the execution of such an interactive application can be modeled as a series of graph sequences. When we draw a graph sequence of such a series on the screen, we do not know the next graph in the series but we know the one before. As a "smooth" transition between the previous and the current graph sequence we can use the traditional animation approach. More precisely: Let $S = [G_1, \ldots, G_n]$ be the previously drawn graph sequence.

Then graph G_n was drawn on the screen using the foresighted drawing for a RGSP $\overline{G}$ of S. Now the user does some input and triggers the graph sequence $S' = [G'_1, \ldots, G'_m]$. To draw this sequence the application computes a RGSP $\overline{G'}$ of S' and uses morphing between the graph G_n with node and edge positions as in $\overline{G}$ and G'_1 with node and edge positions as in $\overline{G'}$.

3.2 Foresighted Graph Drawing with Tolerance

The approach presented in the last section preserves the mental map in a trivial way by using the global drawing, but does so at the cost of other aesthetic criteria.

Now, we extend the foresighted drawing approach such that it can trade aesthetic quality for dynamic stability and vice versa. Again, we compute first a global drawing of the whole sequence, but we allow to make local adjustments to the drawings of the induced graphs up to some predefined threshold while preserving the mental map of the whole sequence. To this end we use a tolerance value δ and allow such drawings for individual graphs of the sequence for which the mental distance to certain other graphs[1] is smaller than δ. As a result we can formulate a weaker problem.

Definition 3.12 (The Tolerant Offline Drawing of Dynamic Graphs Problem) Given a sequence of n graphs $G_1, \ldots, G_n$ and a tolerance value δ. Compute drawings $\Gamma_1, \ldots, \Gamma_n$ for these graphs such that

1. $\Delta(\Gamma_i, \Gamma_{i+1}) < \delta$ for all $1 \leq i < n$
2. $\overline{\Psi} = \sum_{1 \leq i \leq n} \Psi(\Gamma_i)$ is maximal

As a simple corollary we get that $0 \leq \overline{\Delta} < n * \delta$. In general we can expect that $\overline{\Psi}$ increases for larger values of δ. In other words a small δ enforces dynamic stability, while larger values increase local quality (see Figure 3.4).

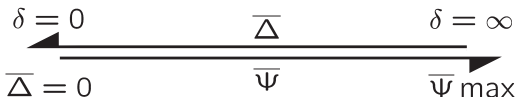


Figure 3.4: Threshold δ: Trading quality for stability.

[1] The strategies that we will present differ in particular with regard to what other graphs are chosen for comparison.

For efficiency reasons we do not try to compute an optimal solution, but we compute approximations. Algorithm 5 is generic in the sense that it works with different static drawing algorithms and related metrics and adjustment strategies. We use the notation $\Gamma|_G$ to denote the drawing which results from restricting Γ to the nodes and edges in G. In other words $\Gamma|_G$ is the drawing for G induced by the drawing Γ.

Algorithm 5 Foresighted Graph Drawing with Tolerance δ

Compute global drawing $\widehat{\Gamma}$ for super graph (resp. GSP or RGSP) of $G_1, \ldots, G_n$
for $i := 1$ **to** n **do**
  $\widehat{\Gamma}_i := \widehat{\Gamma}|_{G_i}$
  $\Gamma_i :=$ adjust$(\ldots)$ // Compute Γ_i by adjusting $\widehat{\Gamma}_i$
  // using one of the strategies discussed in Section 3.2.1
end for
animate drawings $\Gamma_1, \ldots, \Gamma_n$ of graph sequence $G_1, \ldots, G_n$

Algorithm 5 first computes a global drawing $\widehat{\Gamma}$ of a graph which is a projection of the whole graph sequence. Then, for each graph the induced drawing is taken from the global drawing and is adjusted afterward. Section 3.2.1 presents different strategies that can be used to adjust the drawing. Finally the whole sequence of computed drawings is displayed using a graph animation which is described in Algorithm 6.

Algorithm 6 Animate Drawings $\Gamma_1, \ldots, \Gamma_n$ of Graph Sequence $G_1, \ldots, G_n$ with $G_i = (V_i, E_i)$

for $i := 1$ **to** n **do**
  if i=1 **then**
    Display drawing Γ_1
  else
    Fade out all nodes $\in \Gamma_{i-1}^V \setminus \Gamma_i^V$ and edges $\in \Gamma_{i-1}^E \setminus \Gamma_i^E$
    Show animation transforming $\Gamma_{i-1}|_{G_{i-1} \cap G_i}$ to $\Gamma_i|_{G_{i-1} \cap G_i}$
    Fade in all nodes $\in \Gamma_i^V \setminus \Gamma_{i-1}^V$ and edges $\in \Gamma_i^E \setminus \Gamma_{i-1}^E$
  end if
end for

The graph animation algorithm works as follows: The drawing of the first graph of the sequence is displayed. Each following drawing is shown using the following technique:

- fade out all nodes that are removed from the previous graph.

- show an animation moving the nodes that are contained in the previous and the current graph from their old to their new positions.
- fade in all nodes that are inserted in the current graph.

Section 9.1 introduces several methods to compute animations between two drawings of graphs.

3.2.1 Drawing Adjustment Strategies

Classical drawing adjustment methods $\mathsf{adjust}(G_i, \Gamma_{i-1})$ draw a graph G_i by adapting the drawing Γ_{i-1} of the preceding graph. The adjustment strategies presented in this section also take the global drawing into account. We use the super graph (resp. GSP or RGSP) as a rough abstraction of the whole sequence of graphs. In a sense it contains information about the future of the whole sequence. In addition the strategies might consider the previous, next or all graphs of the sequence. Instead of the graph G_i these strategies get the drawing $\widehat{\Gamma}_i$ for the graph induced by the global drawing and try to adjust it while regarding constraints on the mental distance to other drawings. If they cannot fulfill the constraints these strategies yield the induced drawing.

Strategy 1 (Independent Adjustment)
Usage in algorithm 5: $\Gamma_i = \mathsf{adjust}(\widehat{\Gamma}_i, \delta)$
This strategy tries to preserve the mental map by ensuring that $\Delta(\widehat{\Gamma}_i, \Gamma_i) < \delta$. *As a result all graphs in the animation stay close to the global drawing.*

Strategy 2 (Predecessor Dependent Adjustment)
Usage in algorithm 5: $\Gamma_i = \mathsf{adjust}(\widehat{\Gamma}_i, \Gamma_{i-1}, \delta)$
This strategy differs from the above by requiring that the drawing stays close to that of the preceding one, that is $\Delta(\Gamma_{i-1}, \Gamma_i) < \delta$. *Note, that there is no constraint for adjusting* $\widehat{\Gamma}_1$. *As a result* Γ_1 *can be very far from the induced drawing and this can have undesirable effects. The value of* $\Delta(\Gamma_1, \widehat{\Gamma}_2)$ *might get greater than* δ *and all adjustments to* $\widehat{\Gamma}_2$ *might not sufficiently reduce the mental distance. In this case the adjustment will fail and return the induced drawing.*

Strategy 3 (Context Dependent Adjustment)
Usage in algorithm 5: $\Gamma_i = \mathsf{adjust}(\widehat{\Gamma}_i, \Gamma_{i-1}, \widehat{\Gamma}_{i+1}, \delta)$
This strategy extends the previous one by enforcing that the drawing stays close to both the preceding drawing as well as the induced drawing for the subsequent graph, that is $\Delta(\Gamma_{i-1}, \Gamma_i) < \delta$ *and* $\Delta(\Gamma_i, \widehat{\Gamma}_{i+1}) < \delta$. *In particular, this strategy makes sure that* Γ_1 *stays close to* $\widehat{\Gamma}_2$ *and thus we do not run into the problem discussed above when adjusting* $\widehat{\Gamma}_2$.

The above strategies try to adjust a drawing as much as possible, before proceeding to the next one. The previous drawing can thus impose too many restrictions on the next one and render adjustments impossible. The following strategy strives to evenly adjust all drawings in the sequence.

Strategy 4 (Simultaneous Adjustment)
Usage in algorithm 5: $(\Gamma_1, \ldots, \Gamma_n) = \mathsf{adjust}(\widehat{\Gamma}_1, \ldots, \widehat{\Gamma}_n, \delta)$
This strategy simultaneously adjusts the induced drawings $\widehat{\Gamma}_1, \ldots, \widehat{\Gamma}_n$ *such that* $\Delta(\Gamma_i, \Gamma_{i+1}) < \delta$ *for all* $1 \leq i < n$. *A variant of the simultaneous adjustment strategy could also try to preserve the inertia of movements.*

3.2.2 Requirements for the drawing algorithms

In Algorithm 4 (Foresighted Graph Drawing) an arbitrary static drawing algorithm could be used to compute the global drawing. In Algorithm 5 (Foresighted Graph Drawing with Tolerance), however, it is not possible to use an arbitrary static drawing algorithm, because the algorithm has to provide the ability to compute a drawing of a graph by adjusting another drawing, and not just to compute a drawing from scratch. And even more: the degree of the adjustment is not free, but limited by a given threshold.

In Chapter 4 we show how a force-directed algorithm that works iteratively can be modified to meet these requirements. In Chapter 5 and Chapter 6 we discuss how algorithms based on several computing phases, namely hierarchical respectively orthogonal drawing algorithms, can be modified to meet these requirements.

3.2.3 Backbone

In this chapter the super graph played a crucial role. The reason for using the super graph was that it provided an approximation about the graph sequence and that its drawing could be used as a sketch for all graphs of the sequence. However, the super graph is restrictive, as it induces a drawing for all nodes without taking into account that they are of different relevance for the sequence.

To improve that model we now introduce the concept of a backbone of a sequence. Therefore we need a function that defines the importance of a node in the sequence $G_1, \ldots, G_n$.

Definition 3.13 (Importance and Backbone) Given a sequence of graphs $G_1, \ldots, G_n$ with $G_i = (V_i, E_i)$ and $V = \bigcup_{i=1}^{n} V_i$ and $E = \bigcup_{i=1}^{n} E_i$.

- The function $I : V \to [0, 1]$, mapping each node of the sequence to a a value between zero and one, is called *importance*.
- The graph
$$B_i^{\delta_I} = (V_B, E_B)$$
with
$$\begin{aligned} V_B &= \{v \in V \mid I(v) \geq \delta_I\} \subseteq V \text{ and} \\ E_B &= \{(u, v) \in E \mid u, v \in V_B\} \end{aligned}$$
is called *backbone* of the sequence with respect to I and δ_B. If I and δ_B is clear from context B denotes the the backbone.

This concept of a backbone is a generalization of the concept of a super graph: The backbone is less restrictive and is adjusted to the given graph sequence. For $\delta_I = 0$ the backbone is equal to the super graph.

Depending on the choice of the importance function, the backbone represents different base models. There are several possibilities for choosing an importance function. We can define the function depending on the structure of the sequence. For example the number of occurrences of a node in the sequence: $I(v) = |\{i \mid v \in V_i\}|$ for a graph sequence $G_1, \ldots, G_n$, or the number of occurrences of a node in subsequent graphs, that is the longest life time of the node. Another possibility to define the importance function is the node size. Large nodes often act as landmarks in drawings and should not change their position, because the loss of the landmark makes orientation more difficult. Therefore large nodes could be seen more important than small nodes. If we know enough about the semantics of the graphs, we can instead choose an importance function that takes this information into account, that is we can use application-domain specific importance functions.

The improved algorithm for foresighted drawing that uses the backbone instead of the super graph now looks as follows:

Algorithm 7 Improved Foresighted Drawing with Tolerance

compute global drawing $\widehat{\Gamma}$ for the backbone B of $G_1, \ldots, G_n$
for $i := 1$ **to** n **do**
  $\widehat{\Gamma}_i := \widehat{\Gamma}|_{G_i}$
  $\Gamma_i :=$ adjust$(\ldots)$
end for
animate drawings $\Gamma_1, \ldots, \Gamma_n$ of graph sequence $G_1, \ldots, G_n$

In this improved version the global drawing does not provide initial drawing information for nodes $v \in V_i \setminus V_B$, that is those that are not part of the

backbone. So the adjustment functions have to assign initial positions to these nodes.

Chapter 4

Force-Directed Approach

This chapter is divided into two parts. The first part introduces the principles of force-directed graph drawing. The second part shows, how to adapt force-directed drawing algorithms such that they fit into the framework for drawing dynamic graphs as presented in Chapter 3.

4.1 Force-Directed Graph Drawing

Force-directed graph drawing algorithms are intuitive methods to create straight-line drawings of (in most cases undirected) graphs. They compute node positions by simulating forces in a physical system. The nodes and edges of the graph are interpreted as physical components exerting forces on each other. Force-directed methods minimize the energy of the simulated system by moving nodes along their force vectors. Force-directed approaches consist of two parts:

- A force model consisting of physical objects (representing the elements of the graph). For example, a "spring" of "natural length" l_{uv} can be assigned to each pair (u, v) of nodes connected by an edge. The spring follows Hooke's law, that is, it induces a force of magnitude proportional to $d_{uv} - l_{uv}$ on u, where d_{uv} is the Euclidean distance between u and v.
- An algorithm that (approximately) computes an equilibrium configuration of the system. Often, simple iterative methods are used.

The specifications of a model fully represent the intuition behind what is considered a good layout. Its associated algorithm merely serves as an optimization routine for the objective function expressed in the model.

In spite of their disadvantages to be rather slow and to tend to run into local minima, force-directed methods are very popular because they produce

good drawings for many graphs, emphasize symmetric structures in a graph, and are easy to implement.

Many force-directed algorithms have been proposed and tested [Ead84, KK89, FLM95, Bra96, EK97], which differ both in the force or energy model used, and in the method applied to find an equilibrium or minimal energy configuration. In the following sections we introduce some basic models for simulating physical system.

4.1.1 Springs and Electrical Forces

The simplest force-directed method uses a combination of *spring* and *electrical* forces. While nodes represent equally charged particles which repel each other, edges act as springs attracting connected nodes to each other.

For example, Figure 4.1(a) shows a graph modeled with this system. An equilibrium configuration, where the sum of forces on each particle is zero, is illustrated in Figure 4.1(b). This configuration can now be interpreted as a straight-line drawing of the graph, shown in Figure 4.1(c).

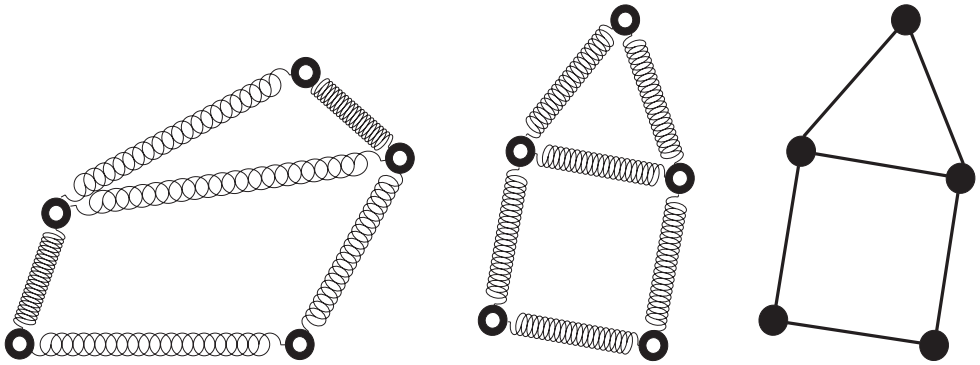


Figure 4.1: A spring algorithm.

The force acting on each node $v \in V$ is given by

$$F(v) = \sum_{u:\{u,v\}\notin E} f_{\text{rep}}(\Gamma^V(u), \Gamma^V(v)) + \sum_{u:\{u,v\}\in E} f_{\text{spring}}(\Gamma^V(u), \Gamma^V(v))$$

where f_{rep} is the electrical repulsion exerted on v by node u, and f_{spring} is the force acting on v by the spring between u and v. The force f_{spring} follows Hooke's law, that is f_{spring} is proportional to the difference between the distance between u and v and the natural length of the spring.

Thus, the forces are defined as follows:

$$f_{\text{rep}}(\Gamma^V(u), \Gamma^V(v)) = \frac{c_1}{d(u,v)^2} 1_{v,u}$$

and

$$f_{\text{spring}}(\Gamma^V(u), \Gamma^V(v)) = |d(u,v) - l_{uv}| c_2 1_{v,u}$$

where $d(u,v)$ is the distance between the node positions $\Gamma^V(v)$ and $\Gamma^V(u)$, $1_{u,v}$ is the unit vector from v in direction of u, l_{uv} is the natural length of the spring between u and v, c_1 is a repulsion constant, and c_2 is a constant controlling the strength of the spring.

The first sum represents the repelling force between every pair of non-adjacent nodes $u, v \in V$. The second sum represents the spring forces between adjacent nodes $u, v \in V$. The direction of this force depends on whether the current distance between the nodes u and v is less or greater than the natural length l of the spring.

To compute an equilibrium of this system nodes are iteratively moved according to a net force vector $F(v)$, which is the sum of all repulsion and spring forces acting on v. After computing the net forces for all nodes, each node is moved a constant δ times this vector. This constant is used to prevent excessive movement due to synchronous update. The system approaches a stable state in which no local improvement is possible any more by iteratively computing the forces on all nodes and updating the node positions accordingly (see Algorithm 8).

Algorithm 8 Spring embedder

for all $v \in V$ **do**
  assign initial position to $\Gamma^V(v)$
end for
for $t := 1$ **to** #Iterations **do**
  for all $v \in V$ **do**
    $F(v) = \sum_{u:\{u,v\} \notin E} f_{\text{rep}}(\Gamma^V(u), \Gamma^V(v)) + \sum_{u:\{u,v\} \in E} f_{\text{spring}}(\Gamma^V(u), \Gamma^V(v))$
  end for
  for all $v \in V$ **do**
    $\Gamma^V(v) := \Gamma^V(v) + \delta \cdot F(v)$
  end for
end for

The initial positions of the nodes can either be chosen randomly, or the nodes can be placed on a circle which is a widely used technique.

4.1.2 Magnetic Fields

For directed graphs it is desirable to have the directed edges point into roughly the same direction, which cannot be achieved by the presented spring embedder model presented above. Sugiyama and Misue [SM95a, SM95b] proposed a model in which some or all of the springs are magnetized, and there is a global magnetic field that acts on the springs. The magnetic field can be used to rotate the edges to point in any given direction. Let Θ be the angle between the prescribed and the current direction of an edge, and $1^{\perp}_{u,v}$ be the unit length vector perpendicular to $1_{u,v}$ and pointing toward a decrease of Θ, then the rotation forces

$$f_{\text{rot}}(\Gamma^V(u), \Gamma^V(v)) = b \cdot d(u,v)^{c_1} \cdot \Theta^{c_2} \cdot 1^{\perp}_{u,v}$$

can be combined with the spring and electrical forces and thus rotates the edge to reduce Θ (see Figure 4.2). The constant b controls the strength of the magnetic field acting on an edge between u and v, c_1 and c_2 are parameters controlling the relative dependency of rotative forces on node distances and angle deviation, respectively.

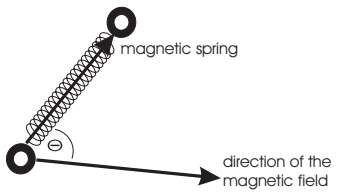


Figure 4.2: Magnetic spring.

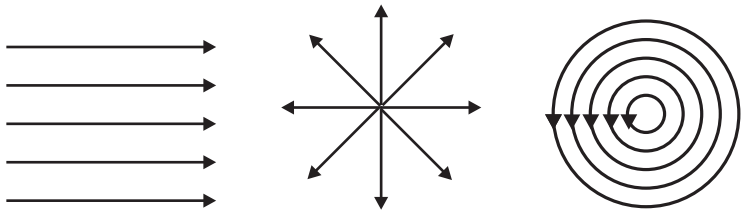
There are three basic types of magnetic fields (see Figure 4.3):

- Parallel: All magnetic forces operate in the same direction.
- Radial: The forces operate radially outward from a point.
- Concentric: The forces operate in concentric circles.

It is also possible to combine these basic fields.

4.1.3 Simulated Annealing

Davidson and Harel [DH96] introduced a general method for minimizing objective functions of combinatorial problems. Given a candidate solution a new solution is proposed by minor modification of the current one. If the new solution reduces the value of the objective function, it becomes the new



(a) Parallel (b) Radial (c) Concentric

Figure 4.3: Types of magnetic fields.

candidate solution. Otherwise, it becomes the new candidate solution only with probability $e^{\frac{-\Delta E}{T}}$, where ΔE is the increase of the objective function, and $T > 0$ is the temperature parameter. Convergence is enforced by slowly lowering T to zero (see Algorithm 9).

Algorithm 9 Simulated Annealing

```
while T > THRESHOLD do
  for all v ∈ V do
    Γ^V_old := Γ^V
    Γ^V(v) := Γ^V(v) + Δ_random
    if E(Γ^V_old) < E(Γ^V) then
      with probability 1 − e^{(E(Γ^V_old) − E(Γ^V))/T} reset Γ^V := Γ^V_old
    end if
  end for
  anneal T
end while
```

Fruchterman and Reingold [FR91] combine this approach with a spring model: the repulsion and spring forces determine the direction of the movement of a node, but the global temperature determines how far the node is moved. The global temperature anneals using a temperature scheme depending on the number of the nodes and edges.

Frick et al. [FLM95] expand this concept by applying local temperature for every single node. The total temperature of the system is the average of the temperatures of all nodes. The amount of movement of every node depends of its own temperature. Local temperatures have the advantage that not all parts of a graph have to get to their final positions at the same time. Further, the temperature is more sensitive about the behavior pattern of the movements of the nodes and therefore, the algorithm terminates faster in general.

4.2 Force-Directed Drawing of Dynamic Graphs

In this section we show how a simple spring embedder [Ead84, Bra01a] can be modified to perform drawing adjustment according to the strategies presented in Chapter 3.

The spring model fits quite well into our framework because the adjustment of a given drawing is implicitly provided by the model: just initialize the algorithm with the given drawing and then let the forces act.

Unfortunately once this spring embedder has computed a drawing that does not preserve the mental map further iterations do not resolve the problem. As a result we extended the spring embedder by simulated annealing. We assume a global temperature T which cools off after each iteration.

Algorithm 10 adjust$(G_i, \widehat{\Gamma}_i, \Gamma_{i-1}, \delta)$ predecessor dependent

```
for all v ∈ V_i do
  if v ∈ Γ̂_i^V then
    Γ_i^V(v) := Γ̂_i^V(v)
  else
    assign initial position to Γ_i^V(v)
  end if
end for
for t := 1 to #Iterations do
  Compute forces for each node in Γ_i^V with global temperature T
  Compute new drawing Γ'_i by applying forces to nodes in Γ_i^V
  if Δ(Γ_{i-1}, Γ'_i) < δ then
    Γ_i := Γ'_i
  end if
  anneal T
end for
return Γ_i
```

Algorithm 10 follows immediately from the general description of the predecessor dependent adjustment strategy in Section 3.2.1 and Algorithm 7. First, the position of the nodes contained in the global drawing induced by the backbone $\widehat{\Gamma}_i^V$ are initialized with the position in the global drawing. The positions of all other nodes are initialized using the standard initialization of the spring embedder. Then a new drawing is computed by applying the forces to the nodes. If the new drawing preserves the mental map it is accepted as the new drawing, else it is discarded. The temperature anneals and we start

over.

The algorithms for the independent and context dependent adjustment strategies work analogously and differ only in the condition checking if the mental map is preserved: instead $\Delta(\Gamma_{i-1}, \Gamma'_i) < \delta$ the independent adjustment strategy uses $\Delta(\widehat{\Gamma}_i, \Gamma'_i) < \delta$ and the context dependent adjustment strategy uses $\Delta(\Gamma_{i-1}, \Gamma'_i) < \delta \wedge \Delta(\Gamma'_i, \widehat{\Gamma}_{i+1}) < \delta$.

Algorithm 11 implements the simultaneous adjustment strategy. The iterations of the embedder are performed simultaneously on all drawings. After each step we check whether the mental distances of a drawing and the drawings of its previous and next graphs are below the tolerance value. If this is not the case the drawing is discarded and the drawing of the previous iteration is used for the next iteration with reduced temperature.

Algorithm 11 adjust$(G_1, \ldots, G_n, \widehat{\Gamma}_1, \ldots, \widehat{\Gamma}_n, \delta)$ simultaneous

```
(Γ_1, ..., Γ_n) := (Γ̂_1, ..., Γ̂_n)
for i := 1 to n do
  for all v ∈ V_i do
    if v ∈ Γ̂_i^V then
      Γ_i^V(v) := Γ̂_i^V(v)
    else
      assign initial position to Γ_i^V(v)
    end if
  end for
end for
for j := t to #Iterations do
  for i := 1 to n do
    Compute forces for each node in Γ_i^V with global temperature T
    Compute new layout Γ'_i by applying forces to nodes in Γ_i^V
    if Δ(Γ_{i-1}, Γ'_i) < δ and Δ(Γ'_i, Γ_{i+1}) < δ then
      Γ_i := Γ'_i
    end if
  end for
  anneal T
end for
return (Γ_1, ..., Γ_n)
```

Short Summary In this chapter we introduced the concept of force-directed drawing algorithms and some basic models, like springs, electrical forces, and

magnetic fields. The force-directed algorithms fit well into our framework because they work iteratively and are able to adjust a given drawing. By using temperature annealing we achieve to limit the adjustment to a decreasing amount.

Chapter 5

Hierarchical Approach

This chapter is divided into two parts. The first part introduces the principles of hierarchical graph drawing. The second part shows, how to adapt hierarchical drawing algorithms such that they fit into the framework for offline drawing of dynamic graphs presented in Chapter 3.

5.1 Hierarchical Graph Drawing

Directed graphs are widely used in applications to model dependency relationships, for example call graphs of programs. Acyclic directed graphs are usually presented with the polyline downward (or upward) drawing convention. The hierarchical approach is intuitive and was originally proposed in [STT81]. An extension which can also be used when the input graph is not acyclic is shown in Figure 5.1.

- If the graph is not acyclic, convert it into a acyclic one by temporarily reversing a subset of its edges. The set of reversed edges should be kept as small as possible to obtain a drawing in which most of the edges follow the hierarchical direction.

- The *layer assignment* step receives an acyclic directed graph as input and produces a layered directed graph by assigning the nodes to layers $L_1, \ldots, L_h$, such that if (u, v) is an edge with $u \in L_i$ and $v \in L_j$, then $i > j$. Each layer corresponds to a row in the drawing plane and in the final drawing each node on layer L_i will have y coordinate equal to i, that is all edges are directed downward. Next the layered drawing is transformed into a *proper layered directed graph*, that is, a layered directed graph such that, if (u, v) is an edge with $u \in L_i$ and $v \in L_j$,

then $i = j + 1$. This is achieved by inserting dummy nodes in the intermediate layers along the edges that span more than two layers.

- The *crossing reduction* step receives a proper layered directed graph as input and produces a new proper layered directed graph in which an order is specified for the nodes on each layer. The orders of the nodes on the layers are chosen in such a way that the number of crossings is kept as small as possible. Together with the layer assignment the orders of the nodes define the topology of the final drawing.

- The *x-coordinate assignment* step receives a proper layered directed graph as input and produces final x coordinates for the nodes preserving the ordering computed in the last step. In the final drawing the dummy nodes are removed and build the bends of the edges.

- The original direction of the edges is restored that were reversed.

The hierarchical approach implicitly establishes an ordering among aesthetics through the ordering of the steps. The method can also be extended to undirected graphs by preprocessing the graph to give it an artificial acyclic orientation.

Several aesthetics can be taken into account during the x-coordinate assignment step. For example, the dummy nodes introduced by replacing the long edges can be aligned to reduce the number of bends, or the nodes can be horizontally displaced to emphasize symmetries, or nodes can also be packed to reduce the area of the drawing.

As the hierarchical approach produces a drawing in several steps where each step is based on and restricted by the result of the previous step, a bad decision in an early phase can reduce the quality of all subsequent phases. Furthermore theoretical problems underlying some of the phases have been proven to be NP-hard[GJ79]. This, in combination with the strong interdependence of the phases make the choice of good and compatible heuristics essential.

In the following sections we will take a closer look at the single steps.

5.1.1 Layer Assignment

The layer assignment has a strong influence on the area required by the drawing as it determines the height of the drawing and gives a lower bound on the width of the drawing. Many different approaches for this step exist. Layer assignment methods usually try to optimize criteria such as minimal

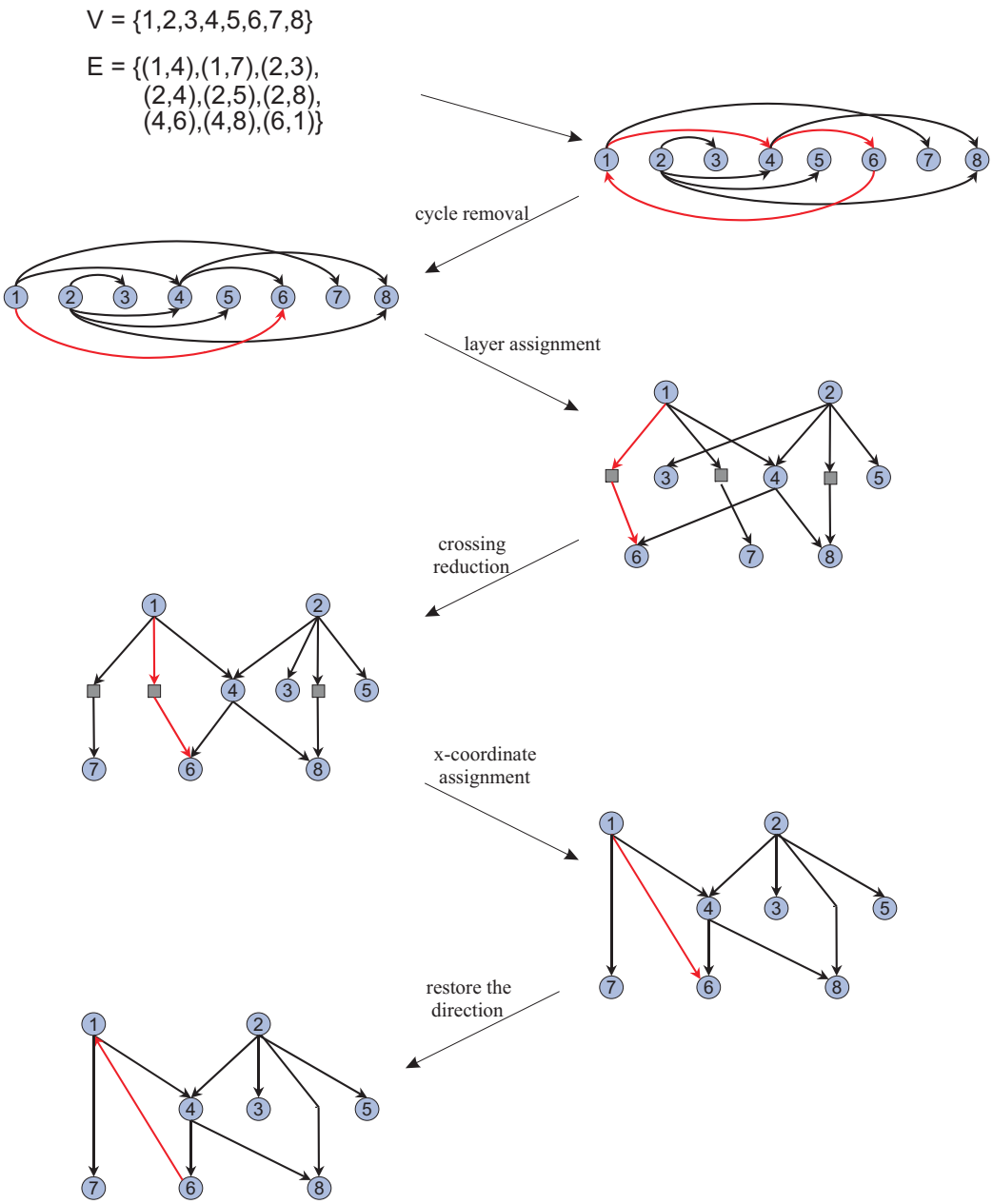


Figure 5.1: The hierarchical approach. Dummy nodes are represented by squares.

height or width of the final drawing. Minimizing both width and height in one drawing has been proven to be NP-hard [Lin92].

Definition 5.1 (Layering and Ranking) Let $\mathcal{L}$ be a *partition* of the node set V of a graph $G = (V, E)$, that is, $\mathcal{L} = \{L_1, L_2, \ldots, L_h\}$, $\bigcup_{i=1}^{h} L_i = V$. The function $R : V \to \{1, 2, \ldots, h\}$ denotes the *characteristic function* of this partition, that is $R(u) = i$ if and only if $u \in L_i$, and is also called *ranking*. $\mathcal{L}$ is called a *layering* if $\forall (v, u) \in E : R(u) > R(v)$ holds.

The *height* of a layering is the number of layers h, the corresponding graph is called an *h-layered graph*. The *width* of a layering is the number of nodes in the largest layer, that is $\max_{1 \leq i \leq h} |L_i|$ and the *span* of an edge (v, w) is defined as $R(w) - R(v)$. A layering is called *proper* if no edge has a span greater than one.

In the following we use the terms height and width also in the context of a ranking and they are interpreted as the height and width of the partition $\mathcal{L}$ implied by the ranking.

The aim of the layer assignment step is to transform an acyclic graph into a layered directed graph. The following requirements hold for a layering:

- The layer should be compact, that is its width and height should be small. A lower bound for the height is the maximum number of nodes in a path from a source to a sink.
- The layer should be proper. This can be easily achieved by inserting "dummy" nodes along edges with span greater than one.
- The number of dummy nodes should be small due to the following reasons:
  1. The computation time of the following steps depends on the number of nodes, including dummy nodes.
  2. Bends in the edges in the final drawing occur only at dummy nodes. Not all dummy nodes introduce bends (bends can be prevented by straightening at the horizontal coordinate assignment step), but is is desirable to avoid this problem by reducing the number of dummy nodes.
  3. The number of dummy nodes on an edges determines the length of the edge. For the eye it is easier to follow short edges.

In the remainder of this section we present different approaches to compute layer assignments. But first we show how to remove cycles in the case the input graph is not acyclic.

Cycle Removal

If the graph contains cycles the cycles can be removed by reverting individual edges. Although this can be done in linear time using a modified depth first search algorithm, trying to make a graph acyclic by reverting as few edges as possible has been proven to be NP-hard [GJ79].

The *maximum acyclic subgraph problem* is stated as follows: find a maximum set $E_a \subset E$ such that the graph (V, E_a) is acyclic. The maximum acyclic subgraph problem is equivalent to the *unweighted ordering problem*: find an ordering of the nodes of G, that is find a mapping $o : V \to \{1, 2, \ldots, |V|\}$ such that the number of edges $(u, v) \in E : o(u) > o(v)$ is minimized.

Thus the easiest heuristic for the maximum acyclic subgraph problem is to take an arbitrary ordering of the graph, which could be computed for example by applying breadth first search or depth first search, and delete all edges (u, v) with $o(u) > o(v)$. This heuristic is fast, but does not allow to give any quality guarantees.

Therefore we introduce another heuristic which guarantees an acyclic set of size at least $\frac{1}{2}|E|$. The idea is to delete for every node either the incoming or outgoing edges (see Algorithm 12).

Algorithm 12 Cycle Removal

```
E_a = ∅
for all v ∈ V do
  if outdeg(v) ≥ indeg(v) then
    append outdeg(v) to E_a
  else
    append indeg(v) to E_a
  end if
  delete pred(v) and succ(v) from G
end for
```

Further heuristics and an exact approach are presented in [BM01, BETT99]. Henceforth we assume that the graphs are acyclic.

Longest Path Layering

The longest path layering first places all sinks in the bottom layer L_1, then each remaining node v is placed in layer L_{p+1}, where the longest (maximum number of edges) path from v to a sink has length p. This produces a layering where many nodes stay at close to the bottom. It can be computed in linear time using a topological ordering of the nodes.

Minimizing the Number of Dummy Nodes

To solve this problem we formulate it as an integer program. The properties of a layering can be stated as follows.

$$R(u) - R(v) \geq 1 \; \forall (u,v) \in E$$

$$R(v) \in \mathbb{Z}^+ \; \forall v \in V$$

Thus minimizing over $\sum_{(u,v)\in E} (R(u) - R(v))$ minimizes the number of dummy nodes. It is shown in [GKNV93] that the corresponding relaxed linear programming problem has an integer solution. Frick [Fri97] presents a detailed study on the number of dummy nodes.

In Section 5.2.1 we will present an algorithm which computes a layering of a directed graph, which may contain cycles, using a topological sorting without removing the cycles first.

5.1.2 Crossing Reduction

In this step the relative positions of the nodes within each layer are computed. The main aim in this phase is to minimize the number of edge crossings. This problem has been proven to be NP-hard [GJ83], even if there are only two layers. In the following we introduce the *layer-by-layer sweep*, which is the general format of most techniques. The most important part of the layer-by-layer sweep is an algorithm solving the two-layer crossing problem, that is, a method for reducing crossings between two layers, which we present afterward.

Layer-by-Layer Sweep

This method works as follows: first, a node ordering of the layers is chosen, for example by computing a depth or breadth first search starting with the sources and assign positions to the nodes in a left to right order arising from the search process.

In the next step, a layer with an ordering, for example layer L_1 is chosen and for $i = 2, 3, \ldots, h$ the node ordering of layer L_{i-1} is fixed while the nodes in L_i are reordered to reduce the number of crossings between L_{i-1} and L_i. After that we can sweep from layer L_h to layer L_1 and repeat these two steps which means L_i is fixed and L_{i-1} is reordered until no further reduction of crossings is achieved.

Barycenter Heuristic

The most common methods employed for the two-layer crossing problem are variations from the *barycenter* method [STT81], which is also called *averaging*. It is based on the intuition that each node should be close to its adjacent nodes in a drawing with few crossings.

An ordering of layer L_i is defined by a permutation π_i of L_i. In this heuristic, we choose the position of a node as the barycenter (average) of the positions of its neighbors N:

$$\text{bary}(v) = \frac{1}{\text{degree}(v)} \sum w \in N(v)\pi(e)$$

for all v in L_2. If two values are equal we separate them arbitrarily by a small amount. Then the nodes are sorted according to their values.

The barycenter method is very popular because it runs fast, gives good results, and even gives a drawing without crossings if one is possible.

Further good and fast heuristics are presented in [EK86, JM97].

5.1.3 Horizontal Coordinate Assignment

So far a vertical coordinate and a horizontal ordering has been computed for all nodes. In the last step the horizontal position for each node is determined according to various aesthetic criteria. Important criteria include that dummy nodes of the same edge should be placed directly above each other to form a straight line and that nodes should be centered over their successors. The theoretical problems underlying the optimization of some of these criteria have also been proven to be NP-hard [GJ79, BETT99].

5.2 Hierarchical Drawing of Dynamic Graphs

In this section we show how a hierarchical drawing algorithm can be modified to be suitable for our framework. As we have seen in Section 5.1 the computation of a hierarchical drawing of a graph following the Sugiyama approach needs several phases: First all nodes are distributed in discrete layers (the layer assignment or ranking assignment phase), then the nodes of each layer are arranged (the crossing reduction phase), and finally the drawing is computed from the layers and their arrangements. One of the problems that occur when trying to apply foresighted drawing with tolerance to hierarchical drawing is that there is no option for global drawing adjustment such as temperature annealing in the force-directed approach. Instead, we

have to divide the drawing adjustment applied in foresighted drawing into two different adjustments: one adjustment for the ranking assignment and another adjustment for the crossing reduction. However, after the ranking adjustment has been performed, we cannot apply standard metrics, as there exists no final drawing of the graphs yet. Therefore we will introduce a new kind of metrics which only concern the rankings of two graphs.

The general algorithm for hierarchical foresighted drawing using the predecessor dependent adjustment is shown in Algorithm 13.

Algorithm 13 Hierarchical Foresighted Drawing with Tolerance

$B :=$ computeBackbone$(G_1, \ldots, G_n, I, \delta_I)$
$R_B :=$ computeGlobalRanking$(G_1, \ldots, G_n, B)$
for $i := 1$ **to** n **do**
  $R_i^l := R_B|_{V_i}$
  $R_i :=$ adjustRanking(R_i^l, R_{i-1}, G_i) // with $\mathfrak{D}(R_0) = \emptyset$
  $(\Gamma_i, \sigma_i) :=$ adjustOrder$(G_i, R_{i-1}, R_i, \sigma_{i-1}, \Gamma_{i-1})$ // with $\mathfrak{D}(\sigma_0) = \emptyset$ and $\mathfrak{D}(\Gamma_0) = \emptyset$
end for
animate drawings $\Gamma_1, \ldots, \Gamma_n$ of graph sequence $G_1, \ldots, G_n$

Starting from the input sequence, we compute the backbone first. As the nodes of the backbone are of highest importance, we try to preserve the mental map of the graph sequence by fixing these nodes to a certain rank for the entire graph sequence. After that, we compute local rankings for each graph with respect to the ranking of the backbone. In the second phase, we try to arrange the nodes on each layer, such that we preserve the mental map, but try to reduce the edge crossings at the same time.

5.2.1 Rank Assignment

In this section we show how to compute a ranking for each graph of a sequence which is suitable to compute drawings preserving the mental map. To achieve that we first compute a ranking of the backbone which serves as a global ranking. The rankings of the individual graphs take this ranking into account and adjust the rank of the nodes not contained in the backbone.

Global and Local Rankings

To apply foresighted graph drawing with tolerance we need an algorithm providing the ability to adapt a given ranking. Sander [San96] introduced an algorithm to do so by applying a topological sorting.

Definition 5.2 (Topological Sorting) The function $\tau : V \to \mathbb{N}$ for a directed acyclic graph $G = (V, E)$ is called *topological sorting*, if the following holds: $\forall (v, w) \in E : \tau(v) < \tau(w)$

Algorithm 14 computes a layering for a graph applying a topological sorting while taking into account that nodes could already have been assigned to a layer – defined by R' (see [San96]).

Algorithm 14 Computation of a layering by topological sorting
$\mathsf{topSort}(G = (V, E), R')$

$M := \emptyset, U = V$
$Z := \{v \in V \mid \mathsf{indeg}(v) = 0\}$
while $U \neq \emptyset$ **do**
  while $\exists v \in Z$ **do**
    $Z := Z - \{v\}, M := M \cup \{v\}, U := U - \{v\}$
    $r := \max(\{R(w) \mid w \in \mathsf{pred}(v) \wedge w \in M\} \cup \{0\}) + 1$
    if $v \in \mathfrak{D}(R')$ **then**
      $R(v) := R'(v)$
    else
      $R(v) := r$
    end if
    $Z := Z \cup \{w \in U \cap \mathsf{succ}(v) \mid \mathsf{pred}(w) \cap U = \emptyset\}$
  end while
  $s_{min} := \infty$
  $s_{max} := 0$
  for all $v \in U$ **do**
    $s_1 := |\mathsf{pred}(v) \cap U|$
    $s_2 := |\mathsf{succ}(v) \cap U| + \sum_{w \in \mathsf{pred}(v) \cap U} |\mathsf{pred}(w) \cap U|$
    if $s_1 < s_{min}$ **then**
      $breaknode := v, s_{min} := s_1, s_{max} := s_2$
    else
      if $s_1 = s_{min} \wedge s_2 > s_{max}$ **then**
        $breaknode := v, s_{max} := s_2$
      end if
    end if
  end for
  if $s_{min} < \infty$ **then**
    $Z := Z \cup \{breaknode\}$
  end if
end while

As a topological sorting only exists for acyclic graphs, existing cycles are broken by reversing one edge of the cycle. The choice of such an edge is important for the quality of the resulting layering. The algorithm works as follows: the set M contains all nodes assigned to a layer, the set U contains all other nodes. Z contains all nodes under consideration in the inner *while* loop which assigns a layer with value greater than that of all predecessor nodes to all nodes not assigned to a layer yet. Nodes already assigned to a layer in advance keep their layer and such the algorithm fulfills the condition of adjusting a given layering.

The *for all* loop chooses a node to break an existing cycle. A cycle is found if Z is empty but U is not. The breaking node is then the starting point for a new iteration.

Using this algorithm we compute a global ranking of the backbone which serves as a basis for the computation of the rankings of the individual graphs of the sequence.

Algorithm 15 computeGlobalRanking$(G_1, \ldots, G_n, B)$

for $i := 1$ **to** n **do**
  $R'_i := \text{topSort}(G_i, \emptyset)$
end for
for all $v \in V_B$ **do**
  $R_B(v) := \text{median}(R'_1(v), \ldots, R'_n(v))$
end for
return R_B

Algorithm 15 works as follows: first of all, for each graph of the sequence a ranking is computed by means of the topological sorting. Afterward a ranking of the backbone is computed. As the backbone is a graph it would be possible to compute the ranking of the backbone also by means of the topological sorting. But in general this would not be a good choice because the backbone does not reflect the hierarchical structure of the individual graphs of the sequence. Instead we use another method: each node of the backbone is assigned to the layer which is the median of all rankings of the sequence which contain this node. Thus, the ranking of the backbone is defined by:

$$\forall v \in V_B : R_B(v) := \text{median}(R_1(v), \ldots, R_n(v))$$

where only defined values in a ranking are taken into account.

The function median returns the value that would be in the middle position after sorting the arguments. Instead of the median it would also be

possible to take the average. But then it is possible to get unexpected results: if a node is mostly on the same layer but in one case on another layer far away, the average rank would be somewhere in the middle. Thus, this node would be assigned to a layer which is never the optimal layer.

Using the median "outliers" are ignored and the node is assigned to a rank which is optimal for at least one graph of the sequence.

Rank Metrics

In Section 2.4 we introduced different metrics to determine the mental distance between two drawings. Unfortunately, these metrics require complete drawings and cannot be applied to graphs with associated rankings, that is partial drawings. If we would apply these metrics to a final drawing and the mental map would not be preserved, we could not decide if the rank assignment or the crossing reduction or both were causing to many changes. Therefore, we introduce a new kind of metrics operating on rankings of graphs. Thus, we are able to check if two rankings are suitable to compute drawings which can preserve the mental map.

Definition 5.3 (Rank metric) Let (G, R) be a graph G with a ranking R. Then the function Δ_R that maps $((G, R), (G', R'))$ to a positive real number is called a *rank metric*. In particular, $\Delta_R((G, R,), (G', R')) = 0$ means that G and G' have a non-distinguishable ranking.

Rank metrics yield only a necessary condition to preserve the mental map of drawings computed with the considered rankings, but not a sufficient condition. That is if a rank metric yields a small value it is possible to compute drawings which preserve the mental map using the considered rankings, but it is also possible that adjustments applied in the crossing reduction step will destroy the mental map of the resulting drawings. If a rank metric yields a large value it is not possible any more to compute drawings preserving the mental map.

In the following we present different rank metrics Δ_R belonging to two different categories:

1. metrics based on differences of nodes between rankings (difference metrics)

2. metrics based on the structure of rankings (structure metrics)

First, we present some difference metrics. Let R_1 and R_2 be two rankings and $V = \mathfrak{D}(R_1) \cap \mathfrak{D}(R_2)$.

Euclidean Rank Metric The *Euclidean rank metric* sums up the distances of the nodes between two rankings:

$$\Delta_R^e(R_1, R_2) := \sum_{v \in V} |R_1(v) - R_2(v)|$$

This metric does not take into account the degree of a change, that is many small changes are equally rated as less large changes. A more sensitive version is achieved by parametrization:

$$\Delta_R^{e,r}(R_1, R_2) := \sum_{v \in V} |R_1(v) - R_2(v)|^r$$

The parameter r is used to weight large changes compared to small changes.

Discrete Rank Metric In some cases it is sufficient to know only the number of nodes whose ranks changed. This is taken into account by the *discrete rank metric*:

$$\Delta_R^d(R_1, R_2) := \sum_{v \in V} \text{sgn}|R_1(v) - R_2(v)|$$

In contrast to difference metrics which examine the change of rank of each node between two rankings, structure metrics examine the relationships of nodes between each other in two rankings.

Orthogonal Rank Metric The *orthogonal rank metric* takes the relative ranking position of the nodes into account, that is if two nodes have been on the same layer in one ranking they should be also on the same layer in the other ranking:

$$\Delta_R^o(R_1, R_2) := \sum_{v_1, v_2 \in V} |\text{sgn}(R_1(v_1) - R_1(v_2)) - \text{sgn}(R_2(v_1) - R_2(v_2))|$$

Horizontal Rank Metric The *horizontal rank metric* takes the width of the implied layers of a ranking into account:

$$\Delta_R^h(R_1, R_2) := \sum_{i=1}^{\text{height}(R_1)} |w(R_1, i) - w(R_2, i)|$$

where $w(R, i) = |\{v \in \mathfrak{D}(R) \mid R(v) = i\}|$ is the width of layer i in ranking R.

Properties of Rank Metrics

Similar to traditional distance metrics for drawings (see Section 2.4) also rank metrics can be invariant under different operations. Given a ranking R, we can compute a new ranking R' by applying the following operations:

1. Translation: R' is the translation of R for $a \in \mathbb{Z}$ with:

$$R'(v) := a + R(v)$$

2. Scaling: R' is the scaling of R for $n \in \mathbb{N}$ with:

$$R'(v) := n \cdot R(v)$$

3. Reflection: R' is the reflection of R with:

$$R'(v) := \text{height}(R) + 1 - R(v)$$

Only the orthogonal rank metric is invariant under translation and scaling, that is the orthogonal rank metric yield zero for two rankings where one is the translation or the scaling of the other. All other metrics are not invariant under these operations.

Adaptability of Rank Metrics

Now, we look at two important properties of rank metrics which provide information about the behavior of the metrics during the adaptation steps. Therefore, we need the following definition:

Definition 5.4 (Fixed Node) For two given rankings $R_1 : V_1 \to \mathbb{N}$ and $R_2 : V_2 \to \mathbb{N}$ a node $v \in V_1 \cap V_2$ is called *fixed* if holds:

$$R_1(v) = R_2(v)$$

With *fixing a node* v we denote the change of a ranking R_2 such that node v that was not fixed in R_1 and R_2 is fixed now.

Fixing nodes of a ranking with respect to another ranking can change the value computed by a rank metric. From that we can derive two further properties of rank metrics.

Definition 5.5 (Monotonous Rank Metric) A rank metric Δ_R is called *monotonous*, if for all rankings $R_1, R_2 \in (V \to \mathbb{N})$ holds:

$$\Delta_R(R_1, R_2') \leq \Delta_R(R_1, R_2)$$

where R_2' results from R_2 by fixing one node.

Definition 5.6 (Fixable Rank Metric) A rank metric Δ_R is called *fixable*, if for two rankings R_1 and R_2 holds:

$$\forall v \in \mathfrak{D}(R_1) \cap \mathfrak{D}(R_2) : R_1(v) = R_2(v) \Rightarrow \Delta_R(R_1, R_2) = 0$$

where R_2' results from R_2 by fixing one node.

The term "fixable" means that the rank metric computes a value of zero if all nodes of one ranking are fixed with respect to another ranking. The following table shows the properties of the presented rank metrics.

Rank Metric	monotonous	fixable
Euclidean	yes	yes
Param. Euclidean	yes	yes
Discrete	yes	yes
Orthogonal	no	yes
Horizontal	no	no

Adjustment of the Rank Assignment

In this section we describe how the rank assignments can be adjusted. After computing a global ranking of the backbone we derive a partial local ranking of each individual graph from the ranking of the backbone (see Algorithm 13). Now we complete the local ranking R_i^l and adjust it such that it is close enough to the predecessor ranking to preserve the mental map, if this is possible (see Algorithm 16).

Algorithm 16 adjustRanking(R_i^l, R_{i-1}, G_i)

$R_i := \text{topSort}(G_i, R_i^l)$
while $(\Delta_R(R_{i-1}, R_i) > \delta_R) \wedge ((V_i \cap V_{i-1}) \setminus \mathfrak{D}(R_i^l) \neq \emptyset)$ **do**
  add node $v \in \{w \mid w \in (V_i \cap V_{i-1}) \setminus \mathfrak{D}(R_i^l) \wedge$
    $\forall u \in (V_i \cap V_{i-1}) \setminus \mathfrak{D}(R_i^l) : I(w) \geq I(u)\}$ to R_i^l
  $R_i^l(v) := R_{i-1}(v)$
  $R_i := \text{topSort}(G_i, R_i^l)$
end while
return R_i

First, we compute a new ranking by sorting G_i topologically, but all nodes contained in the local ranking R_i^l keep their rank. If the rank metric applied to the current and the predecessor ranking yields a value which exceeds the

given threshold δ_R, we fix the rank of one more node to the rank of the node in the predecessor ranking. We choose a node with maximal importance from the node set with the following property: the nodes are contained in the current and previous graph, but not in the backbone. Then we compute a new topological sorting. We repeat this process until the given threshold is no longer exceeded or until all nodes are fixed. In the second case, which can only occur if the used rank metric is not fixable, we stop with a result whose value of the rank metric exceeds the given threshold, but there is no more space for improvement.

5.2.2 Crossing Reduction

In this phase we try to minimize edge crossings while staying as close as possible to the predecessor arrangement of layers. To achieve this we use a two-phased strategy: first we compute an order with little crossings and then we adjust this order such that it preserves the mental map.

Initialization of the Order within Ranks

First, we define an order in each rank.

Definition 5.7 (Order within ranks) Given a ranking R of a graph $G = (V, E)$, the function $\sigma : V \to \mathbb{Z}$ assigning each node v a position within rank $R(v)$ denotes the order within ranks, if the following property holds:

$$\forall v, w \in V : R(v) = R(w) \Rightarrow \sigma(v) \neq \sigma(w)$$

From the function σ we derive the partial order $<_\sigma$ on nodes: $v <_\sigma w \Leftrightarrow \sigma(v) < \sigma(w)$.

Algorithm 17 computes an initial order σ_i of nodes which fulfills the following relative orderedness conditions with respect to its predecessor (for $i > 1$):

1. $\forall v, w \in V_i \cap V_{i-1} \wedge R_i(v) = R_{i-1}(v) \wedge R_i(w) = R_{i-1}(w) :$
   $v <_{\sigma_i} w \iff v <_{\sigma_{i-1}} w$
2. $\forall v \in V_i \cap V_{i-1} \wedge R_i(v) \neq R_{i-1}(v) :$
   $\left|\sigma_i(v) - \frac{\sigma_{i-1}(v)}{|\{w|R_{i-1}(w)=R_{i-1}(v)\}|} \cdot |\{w \mid R_i(w) = R_i(v)\}|\right| \leq 1$

The first condition states that the relative order of the nodes in the same rank in the current and predecessor graph is preserved. The second condition

says that nodes which have changed their rank from the predecessor to the current ranking preserve their relative position within the ranks.

Then we compute $\hat{\sigma}_i$ by smoothly sorting the layers of G_i, where $<_{\hat{\sigma}_i}$ restricted to the j-th layer $\{v|R_i(v) = j\}$ forms a total order. As there exists no constraints for σ_1, $\hat{\sigma}_1$ is obtained by sorting the layers of G_1.

The layers of g_i can be sorted either by the barycenter heuristic or the median heuristic (see [BM01]). Sorting smoothly with respect to sortmax means using an arbitrary comparison-based sorting algorithm where $a \leq b \cdot \mathsf{sortmax}$ is used instead of $a \leq b$. Similarly to simulated annealing, we can use linear, logarithmic or exponential decrease of the factor sortmax.

Definition 5.8 (Final layout) Given a ranking R and an order of ranks σ of graph G, then $\mathcal{H}(R, \sigma)$ is the final hierarchical drawing of G.

Computing the final drawing includes all remaining phases after sorting the ranks and yields the absolute positions of all nodes and edges. Thus we can now check whether the mental map is preserved using some standard difference metrics (see Section 2.4.2). If not, we decrease sortmax and start over.

Algorithm 17 adjustOrder$(G_i, R_{i-1}, R_i, \sigma_{i-1}, \Gamma_{i-1})$

```
sortmax := 1
σ_i := initialOrder(σ_{i-1}, R_{i-1}, R_i)
repeat
  σ̂_i := smoothSort(g_i, σ_i, R_i, sortmax)
  Γ_i := H(R_i, σ̂_i)
  dec(sortmax)
until Δ(Γ_{i-1}, Γ_i) ≤ δ ∨ sortmax < 0
return (Γ_i, σ̂_i)
```

5.2.3 Simultaneous Drawing Adjustment

In this section we illustrate how to apply the simultaneous adjustment strategy to hierarchical drawing. The predecessor adjustment strategy of the previous section tries to adjust a drawing as much as possible with respect to its predecessor. In contrast the simultaneous adjustment strategy provides a uniform adjustment of all graphs.

The main problem in applying the simultaneous adjustment strategy to hierarchical drawing arises in the rank assignment phase. A possible approach in the rank phase would be to perform a topological sorting on all

graphs simultaneously. But this requires that in each iteration one node in each graph is ranked and the mental distance on ranks has to be checked. If the check fails, backtracking has to be performed and the rank of the last node that was ranked has to be fixed. Indeed, this approach is not a good choice for the layer assignment of large graph sequences – in that case it is more efficient to limit the simultaneous adjustment strategy to the crossing reduction phase and to use the predecessor dependent rank assignment phase.

The goal of the simultaneous arrangement of layers is to preserve the relative node order in ranks over the whole sequence. Nodes which change their ranks should preserve at least their relative position. To achieve this goal we compute a global enumeration σ^* of the nodes which is consistent throughout the entire graph sequence. Therefore we build the super graph, compute a drawing of it using a static hierarchical drawing algorithm and after that we retrieve the desired enumeration by projecting the nodes on the x-axis and reading them from left to right.

A local improved enumeration σ' can be derived from σ^* by adjusting the enumeration such that nodes which have changed their rank preserve their relative position (as described in Section 5.2.2, second relative orderedness condition). Using σ^* and σ' we define $\sigma = (\sigma_1, \ldots, \sigma_n)$:

$$\sigma_i = \begin{cases} \sigma_1^*, & \text{if } i = 1 \\ \sigma_i^*, & \text{if } i > 1 \text{ and} \\ & \Delta(\mathcal{H}(R_i, \sigma_i^*), \mathcal{H}(R_{i-1}, \sigma_{i-1})) < \Delta(\mathcal{H}(R_i, \sigma_i'), \mathcal{H}(R_{i-1}, \sigma_{i-1})) \\ \sigma_i', & \text{otherwise} \end{cases}$$

In Algorithm 18, starting with this initial order, we now use the same iteration as in Algorithm 17, except that we use a global sortmax-variable.

Further Comments

We are confronted with another problem concerning the mental map when drawing of dynamic hierarchical graphs: preserving the hierarchy of the graph in the drawing comes into conflict with preserving the mental map. If we restrict node movements by fixing nodes to a certain rank in order to preserve the mental map it is not always possible to preserve the hierarchy in the drawing and vice versa.

Our approach using a backbone to compute a global ranking and fixing the nodes in the backbone to a rank attaches more importance to preserving the mental map. If we would like to guarantee to preserve the hierarchy in the drawing there are two possibilities to achieve this:

Algorithm 18 adjustOrder$((G_1, \ldots, G_n), (R_1, \ldots, R_n))$ simultaneous

```
sortmax := 1
σ* := initialGlobalOrder((G_1, ..., G_n))
σ' := initialLocalAdjustedOrder(σ*, (R_1, ..., R_n))
σ := initialSimultaneousOrder(σ*, σ', (R_1, ..., R_n))
repeat
  for i := 1 to n do
    σ̂_i := smoothSort(G_i, σ_i, R_i, sortmax)
    Γ_i := H (R_i, σ̂_i)
  end for
  dec(sortmax)
until ∀i : Δ(Γ_{i-1}, Γ_i) ≤ δ ∨ sortmax < 0
return (Γ_1, ..., Γ_n)
```

- remove restrictions globally: reduce the size of the backbone by choosing a more restrictive importance function
- remove restrictions locally: remove nodes from the backbone only in these graphs of the sequence where the hierarchy cannot be preserved

In both cases the dynamic stability decreases.

Another interesting observation is the following: in general the nodes in the backbone are of the highest importance and should have a fixed rank while nodes of less importance can change their rank. Thus, it is not always possible to integrate the nodes of the backbone appropriate into the hierarchy. Therefore it is sometimes better to do it the other way round: fix the nodes of lowest importance to improve stability and let the nodes of highest importance move around so that they can be integrated into the hierarchy. In Section 8.2.2 we show an example of this approach.

Chapter 6

Orthogonal Approach

This chapter is divided into two parts. The first part introduces the basic principles of orthogonal graph drawing. The second part shows, how to adapt, extend, and combine existing orthogonal drawing approaches such that they fit into the framework for offline drawing of dynamic graphs presented in Chapter 3.

6.1 Orthogonal Graph Drawing

6.1.1 The Topology-Shape-Metrics Approach

The topology-shape-metrics approach (originally introduced in [BNT86, Tam87, TDB88]) has been devised to construct orthogonal drawings on a grid. The idea of the approach is that an orthogonal drawing is characterized by three fundamental properties, defined in terms of equivalence classes:

- Topology: Two drawings have the same topology if one can be obtained from the other by means of a continuous deformation that does not alter the sequences of edges outlining the faces.
- Shape: Two drawings have the same shape if they have the same topology and one can be obtained from the other by modifying only the lengths of the segments of the edges without changing the angles built by them.
- Metrics: Two drawings have the same metrics if they are congruent up to a translation and/or rotation.

The hierarchical relation between topology, shape, and metric suggests a stepwise generation of the drawing, where at each step an intermediate representation is produced (see Figure 6.1).

- Planarization: This step determines the topology of the drawing, which is described by a planar embedding. For non-planar graphs dummy nodes are inserted which represent crossings. Usually algorithms try to minimize the number of crossings.

- Orthogonalization: This step determines the angles and the bends in the drawing. Only multiple of 90° are assigned as angles which ensures that the drawing is orthogonal. Usually algorithms try to minimize the number of bends in this step.

- Compaction: In this step the final coordinates are assigned to the nodes and the to the edge bends. The dummy nodes introduced in the planarization step are removed. In this phase the main goal is to minimize the sum of the lengths of all edges and/or the area of the drawing.

Detailed description of the planarization, orthogonalization, and compaction step are given in [KW01, BETT99].

6.1.2 Sketch Driven Orthogonal Graph Drawing

Brandes et al. presented in [BEKW02] an orthogonal graph drawing algorithm that produced an orthogonal drawing with few bends in the Kandinsky model while preserving the general appearance of a given sketch. To this end they combine the Kandinsky model which was introduced by Fößmeier and Kaufmann [UK96] and allows to draw planar graphs with maximum degree beyond 4 with the Bayesian Paradigm (see Section 9.2.1).

The remainder of this section is a short summary of the paper [BEKW02]. We adapt our notation to the common one for orthogonal drawing approaches: objects of a graph are called vertexes and the term nodes is used for objects in a network. We introduce the following notations: An *embedded planar graph* $G(V, E, F)$ is a planar graph with a specific circular order of edges around vertices and a specific external face, admitting a planar drawing that respects the given embedding.

A *planar orthogonal box drawing* of a planar graph is a planar drawing that maps each vertex to a box and each edge to a sequence of horizontal and vertical segments. An *orthogonal shape* Q is a mapping from the set of faces F of a graph G to clockwise ordered lists of tuples (e_i, a_i, b_i), $1 \leq i \leq |Q(F)|$, where e_i is an edge, $a_i \in \{1, \ldots, 4\}$ represents the angle formed with the following edge inside the appropriate face in multiples of 90°, and b_i is the list of bends of the edge. A *quasi-orthogonal shape* is an orthogonal shape with $a_i = 0$ allowed, where a 0° angle means that the succeeding edge is adjacent to the same side of the vertex as the preceding edge. We denote

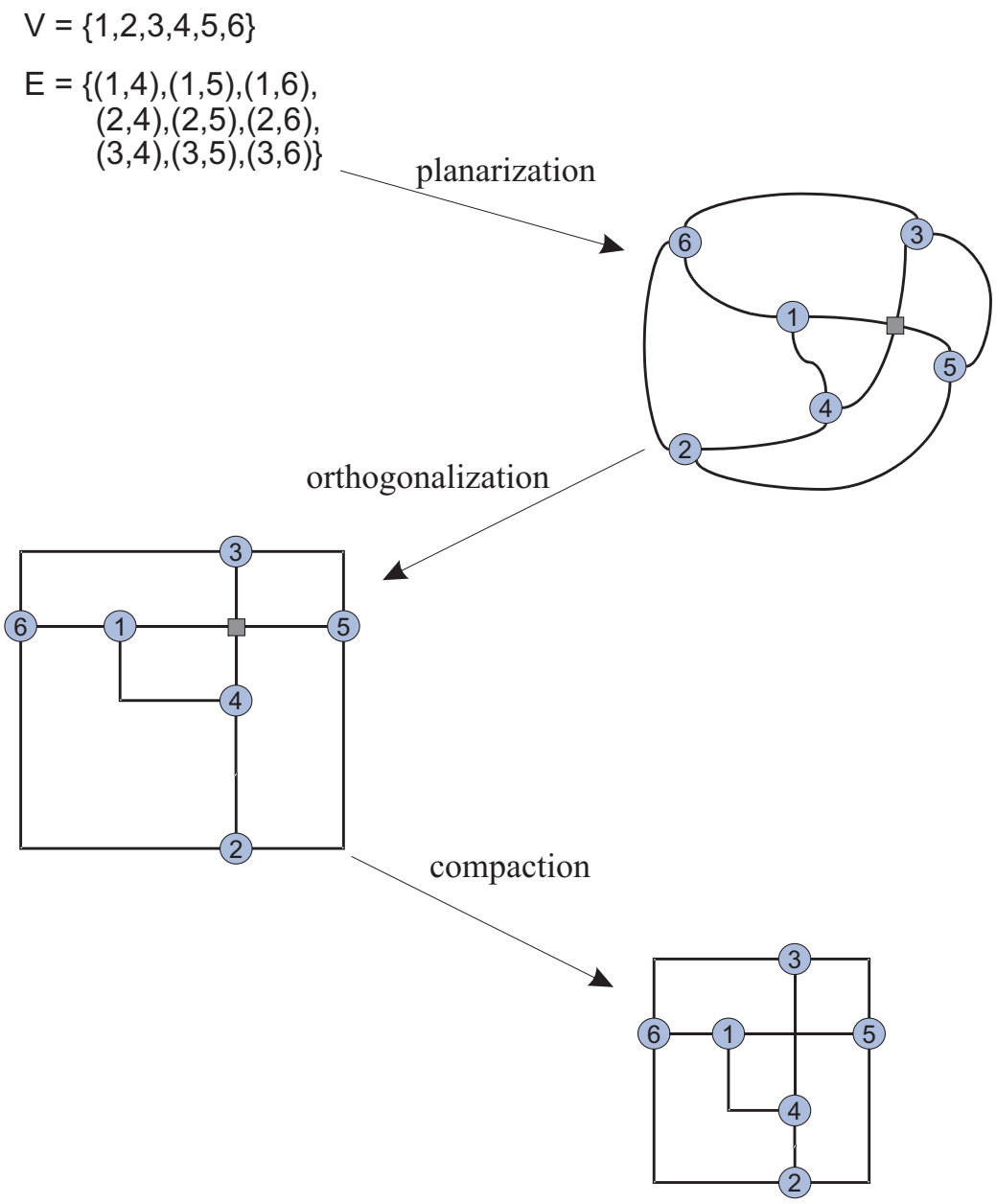


Figure 6.1: The orthogonal approach. Dummy nodes are represented by squares.

with $Q(f,i)$ the i-th tuple of $Q(f)$, with $a(Q,f,i)$ the value of the angle field of $Q(f,i)$, and with $b(Q,f,i)$ the value of the bend field of $Q(f,i)$. A quasi-orthogonal shape Q is called *valid*, if there is a planar orthogonal box drawing with quasi-orthogonal shape Q.

The problem of finding a quasi-orthogonal shape Q is a bi-criteria optimization problem where the two objectives are readability (number of bends) and stability (change in shape).

The readability of a shape Q is independent of the given sketch and defined as the total number of bends, namely

$$B(Q) = \frac{1}{2} \sum_{f \in F} \sum_{(e,a,b) \in Q(f)} |b|$$

The stability of an orthogonal shape Q is expressed in terms of the difference between angles in Q and corresponding angles in the shape S of the sketch

$$\Delta_A(Q,S) = \sum_{f \in F} \sum_{1 \leq i \leq |f|} |a(S,f,i) - a(Q,g,i)|$$

and the difference in edge bends

$$\Delta_B(Q,S) = \sum_{f \in F} \sum_{1 \leq i \leq |f|} \Delta(b(S,f,i), b(Q,f,i))$$

where $\Delta(s1, s2)$ denotes a restricted edit-distance between two strings, in which only insert and delete operations are permitted. The objective function is

$$D(Q|S) = \alpha \cdot \Delta_A(Q,S) + \beta \cdot \Delta_B(Q,S) + \gamma \cdot (B(Q) - B(S))$$

where parameters α, β, and γ control the relative importance of angle or bend changes and bend numbers.

6.2 Orthogonal Drawing of Dynamic Graphs

In this section, we show how to extend the sketch driven orthogonal approach so that it fits into our framework, that is it applies the backbone concept and is guided by metrics.

The general algorithm for orthogonal foresighted drawing using the predecessor dependent adjustment is shown in Algorithm 19.

Starting from the input sequence, we compute the backbone first. After computing the orthogonal drawing for the backbone by applying a standard orthogonal drawing algorithm and obtaining the corresponding quasi-orthogonal shape, we build a sketch S_i for each graph of the sequence. The sketch is a combination of the drawing of the previous graph and the drawing of the backbone restricted to the current graph. If a conflict between the drawing of the backbone and the drawing of the previous graph exists,

Algorithm 19 Orthogonal Foresighted Drawing with Tolerance

$B :=$ computeBackbone$(G_1, \ldots, G_n, I, \delta_I)$
$\Gamma_0 :=$ computeOrthogonalDrawing(B)
$Q_0 :=$ quasiOrthogonalShape(Γ_0)
for $i := 1$ **to** n **do**
  $S_i := (\Gamma_0 \oplus \Gamma_{i-1})|_{G_i}$ // $(\Gamma_i \oplus \Gamma_j)(x)$ is defined as $\Gamma_i(x)$ if $x \in \mathfrak{D}(\Gamma_i)$ and $\Gamma_j(x)$ otherwise.
  $Q_i :=$ adjustShape(S_i, G_i, Q_{i-1})
  $\Gamma_i :=$ adjustMetrics(S_i, Q_i, Γ_{i-1})
end for
animate drawings $\Gamma_1, \ldots, \Gamma_n$ of graph sequence $G_1, \ldots, G_n$

we choose the one of the backbone. Then we compute an adjusted shape using the adjustShape() algorithm and finally adjust this shape using the adjustMetrics() algorithm.

6.2.1 Shape Adjustment

The adjustShape() algorithm first computes the extended network of the sketch. Since the sketch was restricted to the current graph G_i, we only have to handle insertions of new nodes and edges. The insertion of a new node creates a new vertex-node in the Kandinsky network. How to insert new edges adjacent to vertex-nodes with a degree greater than 0 is presented in [BEKW02]. The insertion of a new edge adjacent to a vertex-node with a degree of 0 does not create a new face-node.

We initialize the locally (for every edge) used parameters α and β. Then we compute the quasi-orthogonal shape as described in [BEKW02]. To compare this shape with that of the previous graph, we define a new metrics for quasi-orthogonal shapes. To this end, we extend the definition of a quasi-orthogonal shape given in [BEKW02]. With $Q(f,i)$ we denote the i-th tuple of $Q(f)$, with $\mathsf{edge}(Q,f,i)$ the value of the edge field, with $a(Q,f,i)$ the value of the angle field, and with $b(Q,f,i)$ the value of the bend field of $Q(f,i)$. The value of the edge field of the successor tuple of $Q(f,i)$ is $\mathsf{succEdge}(Q,f,i) = \mathsf{edge}(Q,f,(i+1) \bmod |Q(f)|)$.

Definition 6.1 (Quasi-orthogonal-shape metrics) Let $\mathcal{Q}$ be the set of quasi-orthogonal shapes. The function $\mathsf{diff}_\alpha : \mathcal{Q} \times \mathcal{Q} \to \mathcal{P}(E)$,

$(Q_1, Q_2) \mapsto \{e = \mathsf{edge}(Q_1,f,i) \mid \exists f', j : e = \mathsf{edge}(Q_2,f',j) \ \wedge$
$\mathsf{succEdge}(Q_1,f,i) = \mathsf{succEdge}(Q_2,f',j) \ \wedge \ a(Q_1,f,i) \neq a(Q_2,f',j)\}$

defines the set of edges with the same successor edge, but with different angles in two quasi-orthogonal shapes. The function $\mathsf{diff}_\beta : \mathcal{Q} \times \mathcal{Q} \to \mathcal{P}(E)$,

$$(Q_1, Q_2) \mapsto \{e = \mathsf{edge}(Q_1, f, i) \mid \forall f', j \text{ with } e = \mathsf{edge}(Q_2, f', j) : b(Q_1, f, i) \neq b(Q_2, f', j)\}$$

defines the set of edges with different bends in two quasi-orthogonal shapes. Then the function Δ_α with $\Delta_\alpha(Q_1, Q_2) = |\mathsf{diff}_\alpha|$ is called angle metrics and the function Δ_β with $\Delta_\beta(Q_1, Q_2) = |\mathsf{diff}_\beta|$ is called bend metrics.

If the angle metrics does not fulfill the given angle threshold and there is an α that is lower than the maximal value (the maximal value $6 \cdot |V_i|$ results from the construction of the Kandinsky network and [UK96]), we increment the corresponding α. We deal analogously with the bend metrics and β. The construction of the modified Kandinsky network implies that incrementing β could lead also to a change of angle between two edges. If angle stability is more important than bend stability, then both β and α have to be incremented if the bend metrics does not fulfill the given bend threshold.

Algorithm 20 $\mathsf{adjustShape}(S_i, G_i, Q_{i-1})$

$N_i :=$ compute extended network(S_i, g_i)
$\forall e \in E_i : \alpha_e := 0, \beta_e := 0$
repeat
  $Q_i := \mathsf{quasiOrthogonalShape}(N_i, \alpha, \beta)$
  if $\Delta_\alpha(Q_i, Q_{i-1}) > \delta_\alpha \wedge \exists e \in \mathsf{diff}_\alpha(Q_i, Q_{i-1}) : \alpha_e < 6 \cdot |V_i|$ **then**
    $\forall e \in \mathsf{diff}_\alpha : \mathsf{inc}(\alpha_e)$
  end if
  if $\Delta_\beta(Q_i, Q_{i-1}) > \delta_\beta \wedge \exists e \in \mathsf{diff}_\beta(Q_i, Q_{i-1}) : \beta_e < 6 \cdot |V_i|$ **then**
    $\forall e \in \mathsf{diff}_\beta : \mathsf{inc}(\beta_e)$
  end if
until done
return (Q_i)

6.2.2 Metrics Adjustment

The last step concerns compaction. To be able to preserve the edge length of the sketch S_i, we extend the compaction algorithm from [EK02] by edges of prescribed length. This extension is done straightforwardly by extending the length function: let $e = (u, v)$ be an edge and (u_x, u_y) the position of u in S_i, then

$$\mathsf{length}'(e) = \begin{cases} |u_x - v_x| + |u_y - v_y|, & \text{if e is fixed} \\ \mathsf{length}(e), & \text{otherwise} \end{cases}$$

An edge can be fixed if it is in the current graph as well as in the previous one, and if the values of the corresponding bend fields are equal. We compute the final drawing by applying the extended compaction algorithm. If the metrics does not fulfill the given threshold we fix one more edge if there are any left.

Algorithm 21 adjustMetrics(S_i, Q_i, Γ_{i-1})

```
fixedEdges := ∅
repeat
  Γ_i = compact(Q_i, S_i,fixedEdges)
  if Δ(Γ_{i-1}, Γ_i) > δ ∧ fixedEdges ⊂ {E_i ∩ E_{i-1}} − {diff_β} then
    extend fixedEdges by one edge of {E_i ∩ E_{i-1}} − {diff_β}
  end if
until done
return (Γ_i)
```

6.2.3 Simultaneous Drawing Adjustment

So far, we have seen how to apply orthogonal drawing to the predecessor adjustment strategy. But it is also possible to apply it to the simultaneous adjustment strategy. In this case the drawing of the backbone is used as sketch and we use global parameters α and β instead of local ones to achieve a more uniform adjustment of angles and bends over the whole sequence. The adjustShape() algorithm first computes the quasi-orthogonal shapes for all graphs. If the condition for the angle metrics $\exists i : \Delta_\alpha(Q_{i-1}, Q_i) > \delta_\alpha \ \wedge \ \alpha < 6 \cdot |V_i|$ is not fulfilled, that is there is a tuple of successive shapes which do not hold the angle metrics condition and there is some space for improvement, α is increased. Analogously, β is changed depending on the bend metrics. To compute the final drawings we use the predecessor-dependent adjustMetrics() algorithm.

Chapter 7

The Dynamic Graph Drawing System DGD

This chapter describes the dynamic graph drawing system DGD which implements the presented framework. This system has been developed by the Graph Animation project group [Pro].

The DGD system provides different input and output modules, and drawing algorithms. It is easy to extend the system by implementing the defined interfaces. The system is implemented in Java [Lan], which provides appropriate concepts to model interfaces and classes. Also, Java is a suitable programming language to design web applications.

Importing a graph sequence

To compute a graph animation of a given sequence of graphs, the sequence has to be read by an input module first. The GAML (Graph Animation ML) reader is one input module of the DGD system. GAML is an XML format based on the definition of GraphML [For]. GAML describes sequences of graphs. A sequence consists of several graphs and a graphs consists of nodes and edges. A node provides the following properties:

- identifier
- width of the shape
- height of the shape
- type of the shape (either a rectangle, a circle, or an image of type jpg, gif, svg)
- source of the image (only for the case that the shape is an image)

- color of the node
- color of the outline of the node
- label
- color of the label

An edge provides the following properties:

- identifier
- source node
- target node
- type (directed or undirected)
- drawing convention (straight-line, polyline, or splined)
- type of the lines (lined, dashed, dotted)
- thickness
- type of arrow (none, standard, triangle, or diamond)
- label
- color of the label

Furthermore, it is possible to define a sequence of drawings in GAML. To this end it is possible to assign x- and y-coordinates to the nodes and positions of bends to the edges.

Figure 7.1 shows the classes used to represent the graph sequences:

- Graph stores a list of its nodes and edges
- Node represents a node in a graph
- Edge represents an edge in a graph
- LayoutedNode represents a node with drawing information for all graphs it belongs to
- LayoutedEdge represents an edge with drawing information for all graphs it belongs to

The GAML reader creates objects of these classes. For each node and each edge only one instance is created, even if it is contained in several graphs of the sequence. All graphs of the sequence are stored in a list.

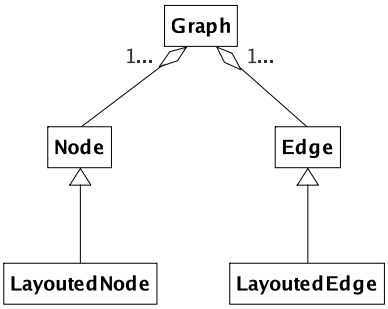


Figure 7.1: Class diagram of the graph structure of DGD

Computing Drawings for a Sequence of Graphs
The DGD system provides implementations of the presented force-directed, hierarchical, and orthogonal tolerant foresighted drawing algorithms. Details about the hierarchical implementation are given in [Poh05], about the orthogonal implementation in [Bir05].

The input data of the drawing algorithms is a list of graphs. After computing the drawings of the sequence, the drawing algorithms return again a list of graphs, which contains objects of type LayoutedNode and LayoutedEdge instead of node and edge.

The result can either be displayed using the DGDViewer or can be exported as SVG in a file.

7.1 The Viewer DGDView

The viewer DGDView is integrated into the DGD system. Figure 7.2 shows its class diagram. The class DGDView is the core of the program. It creates all necessary controlling elements and delegates the events to the corresponding components of the system.

The class DGDViewer provides the drawing area on which the objects of types DGDViewerNode, DGDViewerEdge, and DGDViewerGraph draw the graphs of the sequence.

Smooth transition between two drawings of graphs are animated in the following phases. First all deleted nodes and edges fade out. Next all remaining nodes and edges are moved to their new positions using linear interpolation. Finally all new nodes and edges appear.

Furthermore, the viewer provides a color coding scheme for the outlines of the nodes and the edges to communicate the past and future changes of

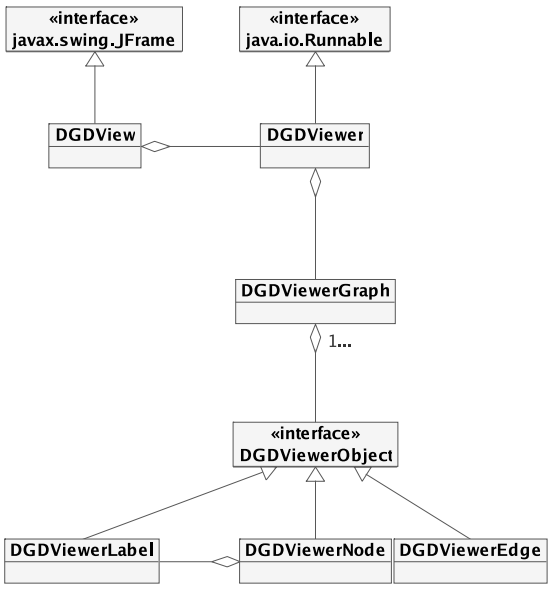


Figure 7.2: Class diagram of `DGDView`

the current graph. Newly inserted nodes and edges are colored green, nodes and edge which will be deleted from the current graph are colored red.

After starting DGDView the user sees the main window. It provides the possibility to load graph sequences, to choose a drawing algorithm, to set the parameter for the chosen drawing algorithm, to show the computed graph animation (see Figure 7.3) and to export the graph animation as an SVG file.

SVG Export

Details of the implementation of the SVG export are presented in [Zim05].

In [GW05a, GW05b] we used the features of SVG to visualize refactorings which we extracted from software archives (see Figure 7.4). In this case, only static graphs representing parts of the inheritance hierarchy are shown. Classes are represented by nodes. The image of a node is an interactive SVG file. It shows the method names (color coded by the type of refactoring) of the class and provides mouse-over tooltips which display the detected refactorings.

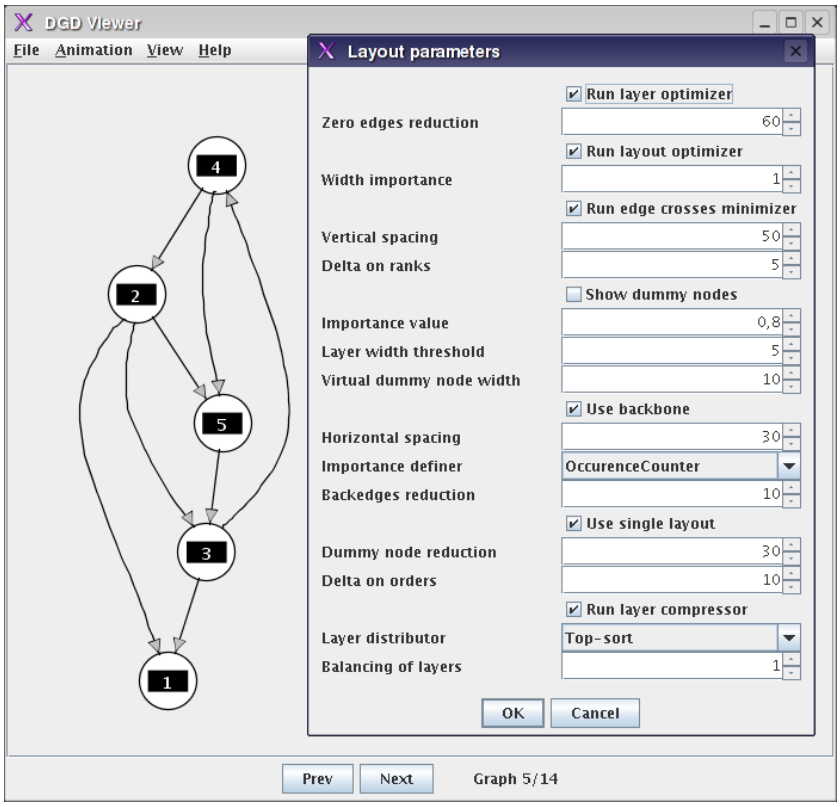


Figure 7.3: Main window of DGDView

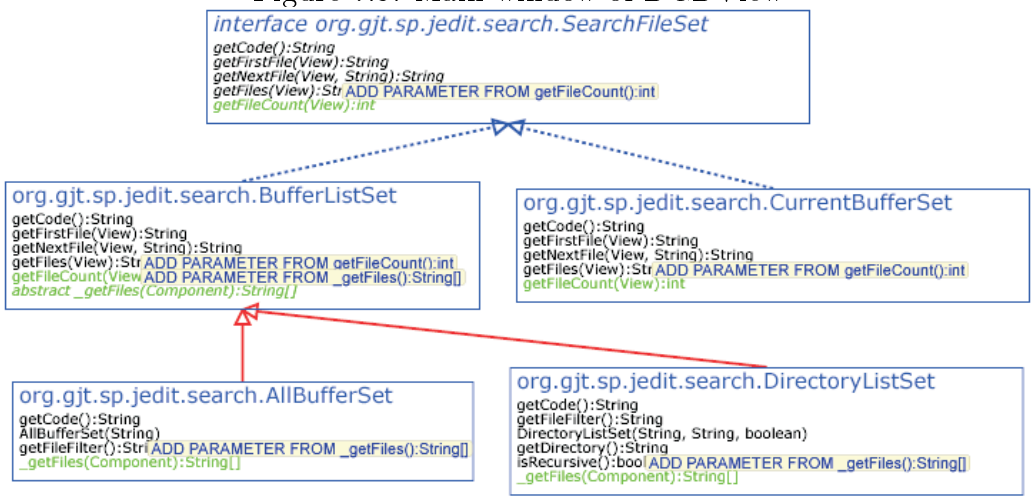


Figure 7.4: Visualization of Refactorings

7.2 Web Application

The web application offers the same functionality as the DGDViewer besides the resulting animations are shown as interactive SVG animations instead of being displayed in the internal viewer. Details of the implementation are presented in [Zim05].

Figure 7.5 shows the first web page of the application. It provides the possibility to submit general information about the node and edges appearance, the chosen drawing algorithm and the graph sequence. Figure 7.6 shows the second web page of the application. It provides the possibility to submit the parameters for the selected drawing algorithm. Figure 7.7 shows the computed SVG graph animation.

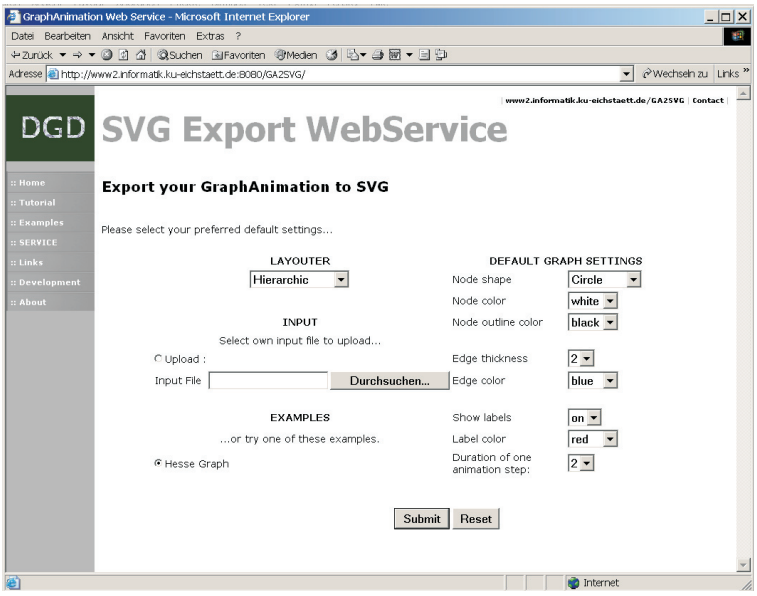


Figure 7.5: Web Application (1).

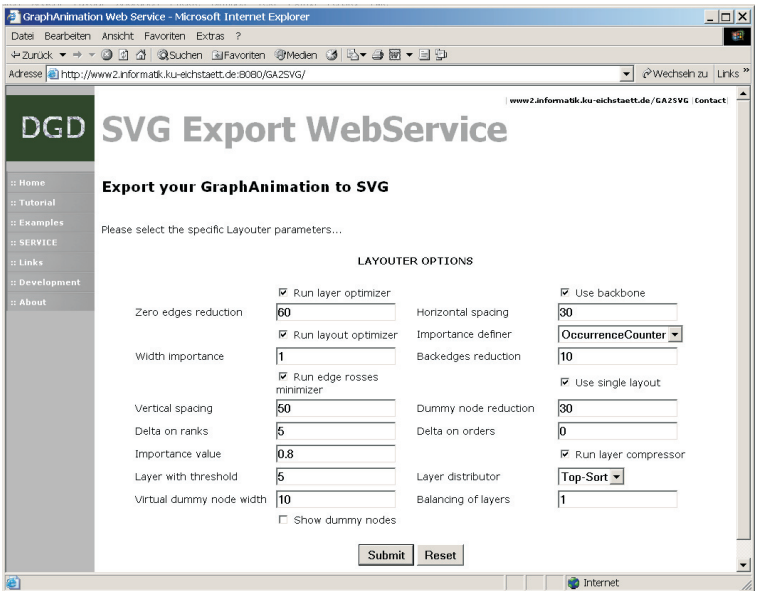


Figure 7.6: Web Application (2).

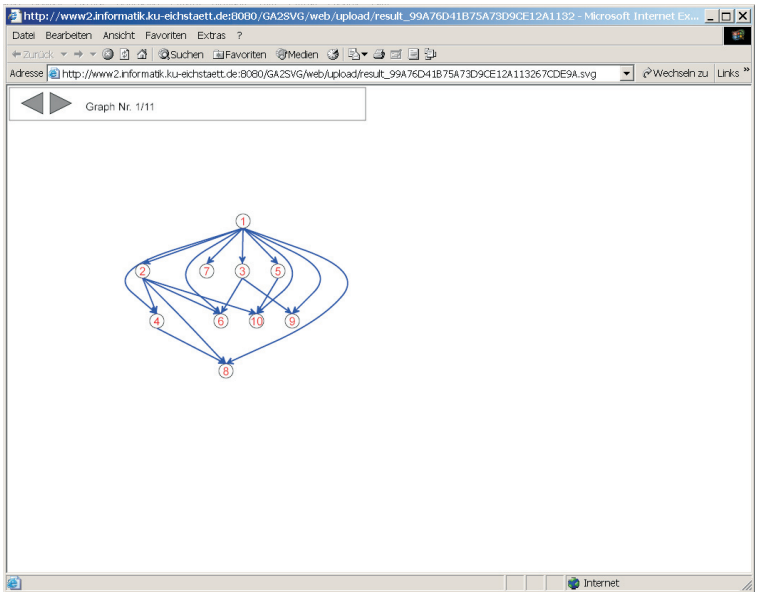


Figure 7.7: Web Application (3).

Chapter 8

Case Studies

In this chapter we present several applications in different domains and show the effects of different parameters and strategies. The first section gives examples for foresighted graph drawing, the second section shows examples for the tolerant version of foresighted graph drawing. To conclude this chapter we discuss the difficulties with evaluating the effectiveness of our approach.

8.1 Foresighted Graph Drawing

8.1.1 Algorithm Animation

Algorithm animation is one of the most prominent areas of software visualization. The GaniFA applet visualizes and animates several generation algorithms from automata theory including the generation of a non-deterministic finite automaton (NFA) from a regular expression RE [WM96]. We have included GaniFA into an electronic textbook on automata theory to allow interactive exercises [DK00, DK01, GAN].

In case of visualizing transition diagrams of finite automata a static drawing algorithm is a good choice, but the algorithm $RE \rightarrow NFA$ changes the graph successively. Animations of algorithms which change graphs, that is add or delete nodes and edges, are often very confusing, because after each change a new drawing of the current graph is computed. In this new drawing nodes are moved to different places although the algorithm did not actually change these nodes. As a result it is not clear to the user what changes of the graph are due to the graph algorithm and what changes are due to the drawing algorithm.

The lower part of Figure 8.1 shows how foresighted drawing can be used to animate the conversion of a regular expression $(a|b)^*$ into an appropriate

nondeterministic finite state automaton ($RE \rightarrow NFA$). In contrast to the upper part of Figure 8.1, which shows the same conversion, this visualization is significantly more clear because once created, a node does not change its position.

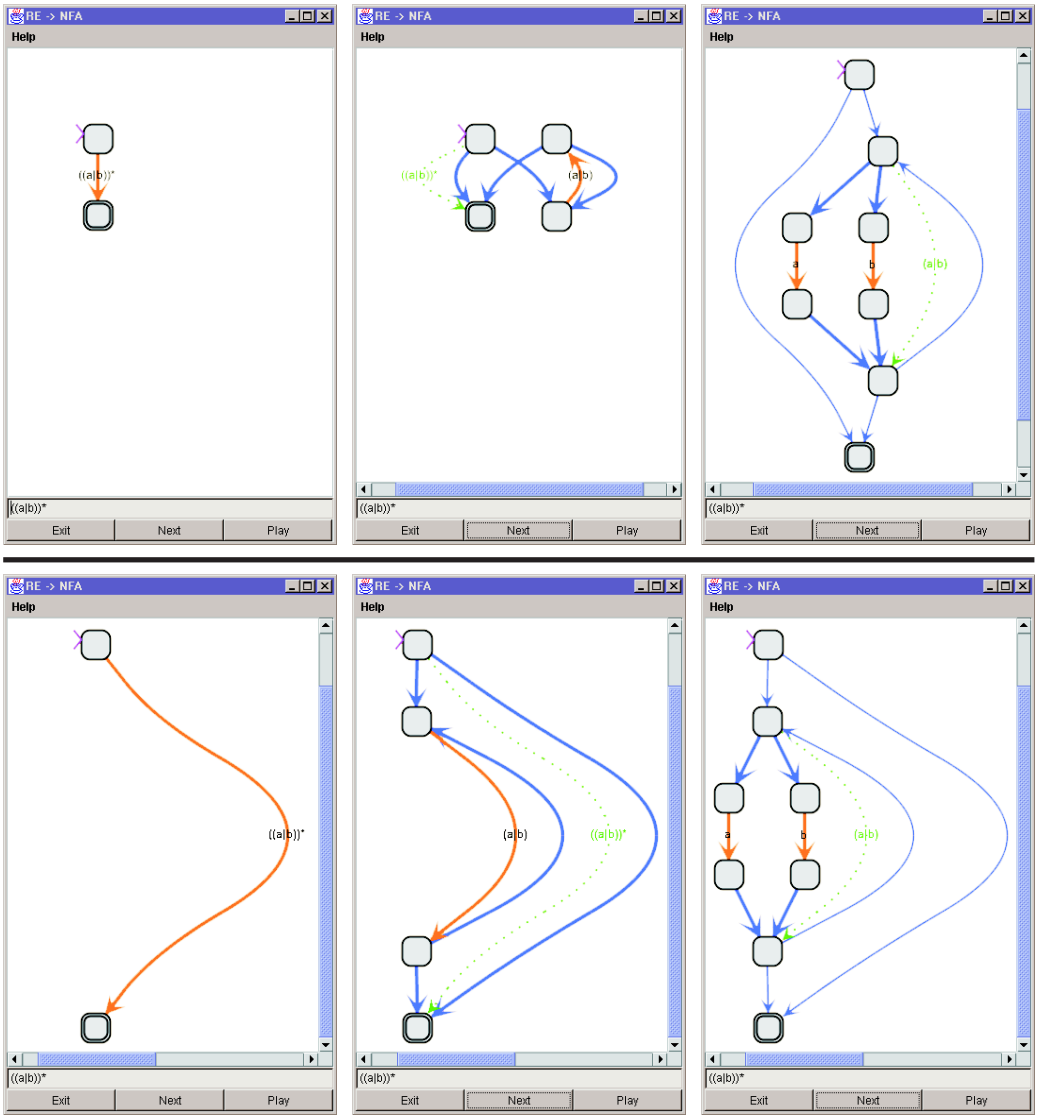
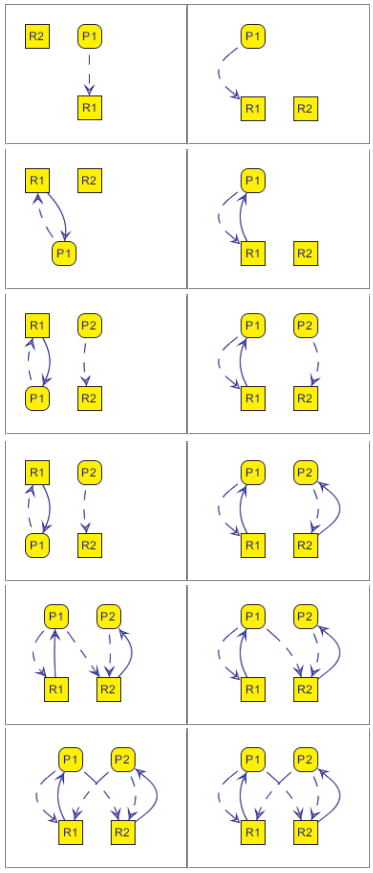


Figure 8.1: Ad-hoc and foresighted graph drawing of the generation of a NEA for $(a|b)^*$



(a) ad-hoc (b) foresighted

Figure 8.2: Ad-hoc and foresighted graph drawing for a deadlock situation.

8.1.2 Resource Allocation

In Figure 8.2 we show a classical deadlock situation which in practice can be avoided by using ordered resources. Process P1 requests resource R1 (indicated by the dashed arrow) and gets exclusive access (indicated by the solid arrow). Then P2 requests and gets exclusive access to R2. Next P1 requests access to R2, but the resource is locked by P2. Then P2 requests access to R1 which is locked by P1. As long as none of the two processes releases its lock both are stuck.

In the left sequence using the ad-hoc approach (compute a drawing from scratch for each graph using a static drawing algorithm) it is difficult to see what is changed between subsequent graphs. In the 6 pictures of the ad-hoc approach P1 is drawn at 4, P2 at 2, R1 at 3 and R2 at 4 different positions. As a result for example the dashed arrow from P1 to R1 is drawn upwards

in some and downwards in other pictures. Using foresighted graph drawing nodes remain at their position and edges are drawn consistently in all graphs.

8.1.3 Buffered IO

In Figures 8.3 we compare the different kinds of foresighted graph drawing (using a super graph, a GSP, or a RGSP) and the ad-hoc approach with each other. In this example several users share a printer, but the access to the printer is buffered by using a printer spool. In the drawing of the ad-hoc drawing approach positions of the nodes "Printer" and "Spool" change several times. In the super graph based foresighted drawing node positions and edge routings are fixed, but there is much unused space. In the drawing based on a GSP node positions do also not change, but as the nodes "User1" and "User2" share the same position, the drawing is more compact. But there are sharp bends in the edges of the first three graphs and normal bends in those of the last four graphs. Finally in the RGSP based foresighted drawing there are no sharp bends as edges at different life times share bend positions. Obviously, for this example RGSP-based foresighted drawing produces the best results.

(a) ad-hoc (b) super graph (c) GSP (d) RGSP

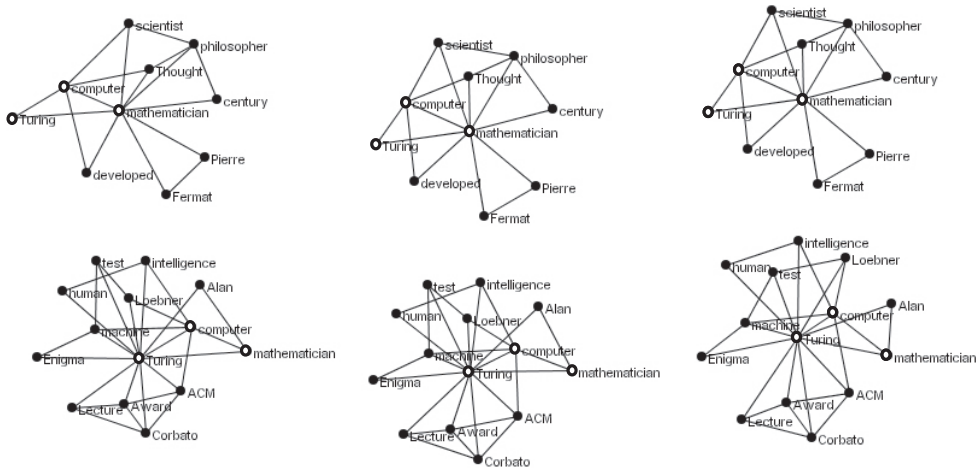
Figure 8.3: Different graph drawings of the buffered I/O example.

8.2 Foresighted Graph Drawing with Tolerance

8.2.1 Force-directed Approach

Word Collocation Graphs

In Figure 8.4 navigation through word collocation graphs is shown, more precisely the sequence consists of the two graphs for the words `mathematician` and the word `Turing`. In the first column the induced drawings of the graphs are shown. In the second column predecessor dependent adjustment with orthogonal mental distance and $\delta = 0$ is shown. Compared to the induced drawings the nodes in the drawing of the first graph are more evenly distributed. Unfortunately for the second graph no drawing could be computed that fulfills the constraint, that is gets close to the drawing of the previous graph and thus the induced drawing is shown for the second graph. By increasing δ also a drawing for the second graph can be computed, but the horizontal alignment of the nodes labelled `Turing` and `mathematician` has changed.

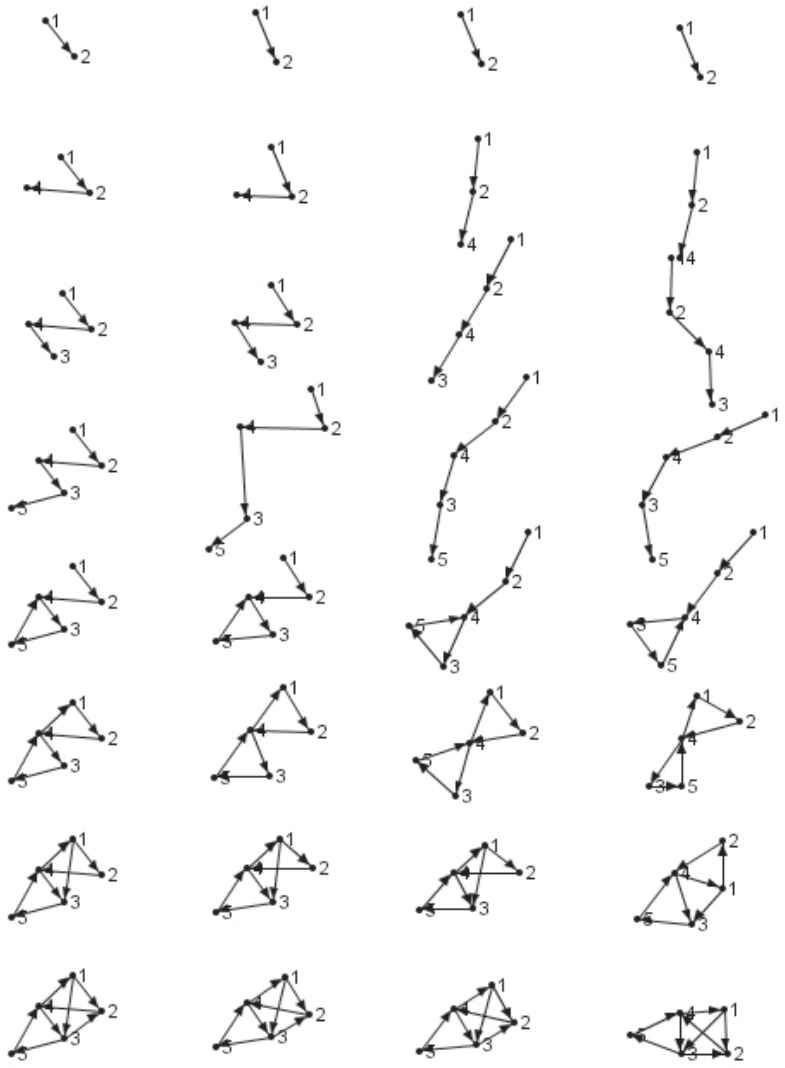


(a) induced (b) predecessor $\delta = 0$ (c) predecessor $\delta = 2$

Figure 8.4: Predecessor dependent adjustment fails for $\delta = 0$ and yields induced layout.

Drawing a House

Figure 8.5 depicts drawings of a sequence of drawing a house in a single stroke computed with simultaneous adjustment, orthogonal mental distance and increasing values of δ. In particular for the 6th and 7th graph the orthogonality with respect to previous and subsequent graphs decreases for increasing values of δ.

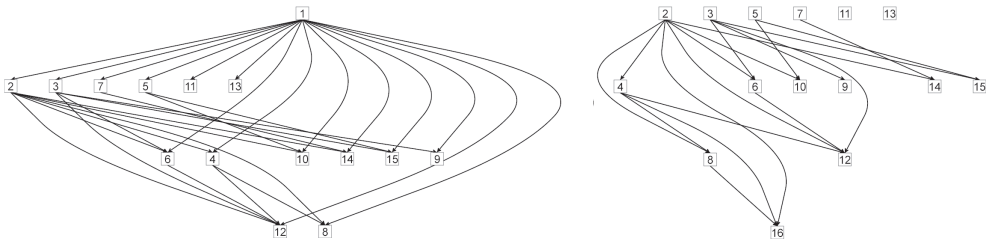


(a) induced (b) $\delta = 0$ (c) $\delta = 5$ (d) $\delta = 10$

Figure 8.5: Drawing a house with increasing tolerance and simultaneous adjustment.

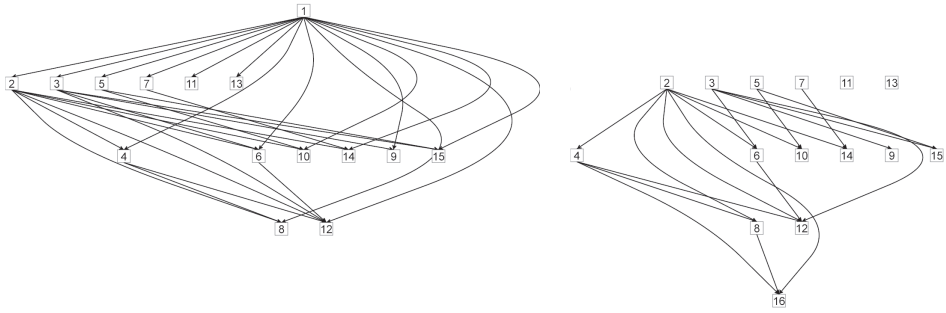
8.2.2 Hierarchical Approach

Hasse-Diagrams



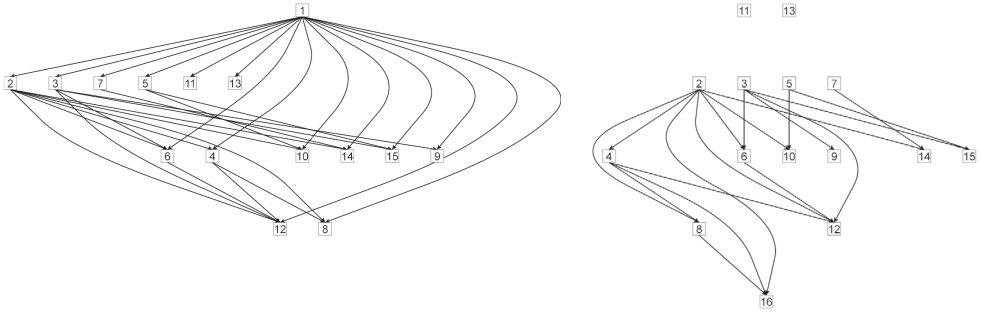
(a) Graph 15 (b) Graph 16

Figure 8.6: Ad-hoc approach.



(a) Graph 15 (b) Graph 16

Figure 8.7: Foresighted with small δ, $delta_R = 0$.



(a) Graph 15 (b) Graph 16

Figure 8.8: Foresighted with large δ, $delta_R = 2$.

In Figure 8.6, 8.7 and 8.8 we show snapshots from three different animations of the same graph sequence, which consists of evolving Hasse-diagrams.

(Hasse-diagrams represent divisibility on natural numbers: there is an edge between v and w, if w is divisible by v.) In the graphs 1 to 15 the nodes representing these numbers are inserted successively. In graph 16, node 1 is deleted and node 16 is inserted. In Figure 8.6 the ad-hoc approach is shown: for each graph a new drawing is computed by using a static drawing algorithm. The mental map is poorly preserved as all nodes change their ranks and more than half of the nodes also change their order within the ranks. In Figure 8.7 the predecessor dependent adjustment strategy with $\delta_R = 0$ and a small δ is shown: the mental map is well preserved. No node changes its rank, and the order within the ranks is stable as well. But the local drawings are worse as there are more edge crossings. In Figure 8.8 the predecessor dependent adjustment strategy with $\delta_R = 2$ and a large δ is shown: the left graph is equal to that produced by the ad-hoc approach. But in the next graph, all nodes contained in the backbone do not change their rank. So it is a good compromise between preserving the mental map and achieving local quality.

Call Graphs

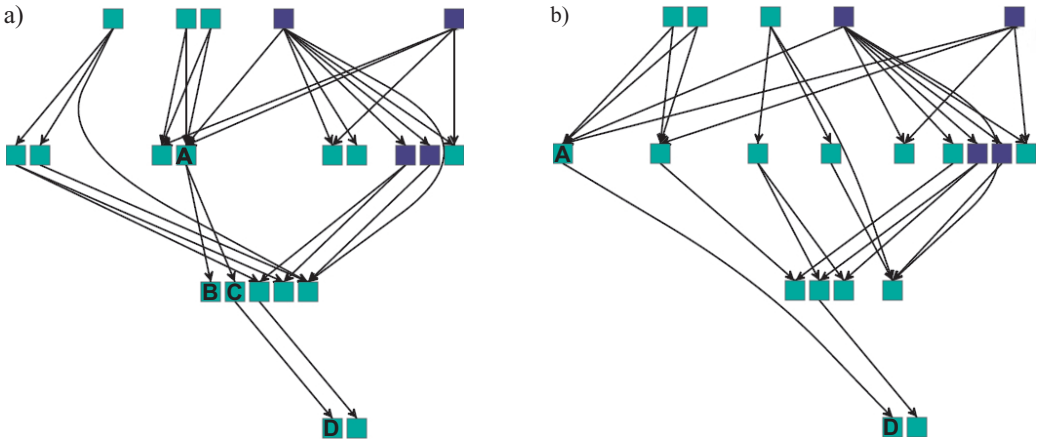


Figure 8.9: Call graphs before and after a refactoring: The labeled nodes represent the following functions: A is `DiffParser.getResults()`, B is `DiffParser.parseItCPP()`, C is `DiffParser.parseIt()` and D is `DiffType.getDiffNumber()`. The colors of the nodes encode the package to which the method belongs.

This example is taken from the real world: the CVS-miner developed in our group. Figure 8.9 shows two call graphs before and after an "In-

line Method" refactoring. After the refactoring the function `getResults()` calls the function `getDiffNumber()` directly instead of through `parseIt()`. A closer look at the source code reveals that the code of the functions `parseIt()` and `parseItCPP()` has been inserted into `getResults()`.

Visualization of Results in Soccer League

Using the described method of drawing a sequence of dynamic graphs it is possible to visualize the results of a sports league.

The following example visualizes a part of the season 2003/2004 of the German Bundesliga of soccer. The visualization is done by drawing a sequence of graphs that was created as follows:

Each graph of the sequence denotes two consecutive match-days of the league, that is the first graph shows results of the first and the second match-day. Each node of the graph represents a team of the league and an edge indicates the result of a game, that is an edge from a to b indicates that a defeated b. In case of a draw the edge is undirected (that is the arrow is omitted).

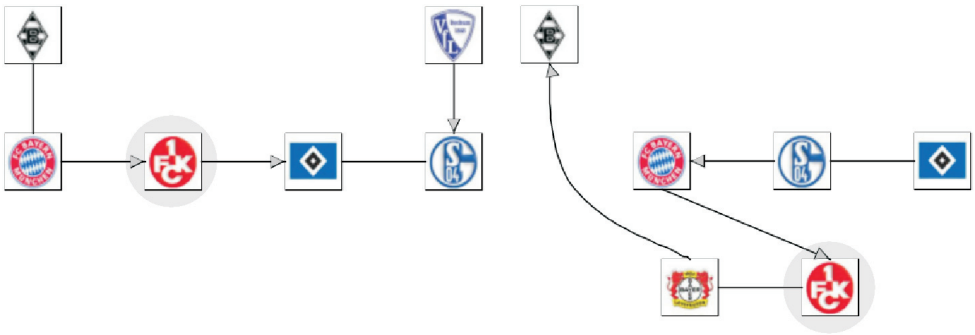
To keep the visualization clear, only the matches of one selected team are taken into account. Furthermore all matches of the selected team's opponents of the corresponding match-day are added to the graphs. It turns out that this is sufficient for a good graph sequence.

The following example shows the results of the match days 9 to 13 in a sequence of four graphs. The selected team that should be taken care of is the 1. FC Kaiserslautern - marked in the pictures with a dark circle around the corresponding node.

In the first graph (Figure 8.10(a)) the node of the 1. FC Kaiserslautern is in one row like those from FC Bayern München, Hamburger SV and FC Schalke 04. One could think that the latter two nodes have to be drawn on the third row as Kaiserslautern won against Hamburg, but as FC Schalke 04 won against FC Bayern München later on, the team of Schalke can be considered as good as the team of München at that time.

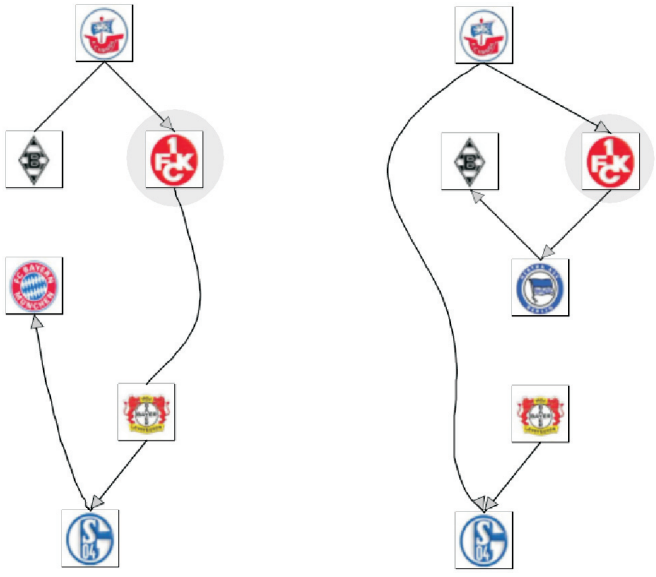
In the second graph(Figure 8.10(b)), Kaiserslautern's node sinks down in the drawing as the team could not win against Bayer 04 Leverkusen. In the third drawing, the selected team lost against the team of Hansa Rostock – but as Hansa Rostock could also win against Schalke 04, it has to be considered rather strong in that moment. Another reason for that behavior is that Kaiserslautern won against the team from Hertha BSC Berlin later on. One can say that the third graph already shows the tendency of 1. FC Kaiserslautern - and also that of FC Schalke 04 which will be defeated by Hansa Rostock.

The fourth graph (Figure 8.11(b)) shows the tendency of the teams after the 13th match-day. Hansa Rostock is in a very good condition but is followed by Kaiserslautern and also by the team of Borussia Mönchengladbach. The teams from Schalke 04 and Bayer Leverkusen seems to have had problems at that time.



(a) match day 9 and 10 (b) match day 10 and 11

Figure 8.10: Soccer League (1).



(a) match day 11 and 12 (b) match day 12 and 13

Figure 8.11: Soccer League(2).

8.2.3 Orthogonal Approach

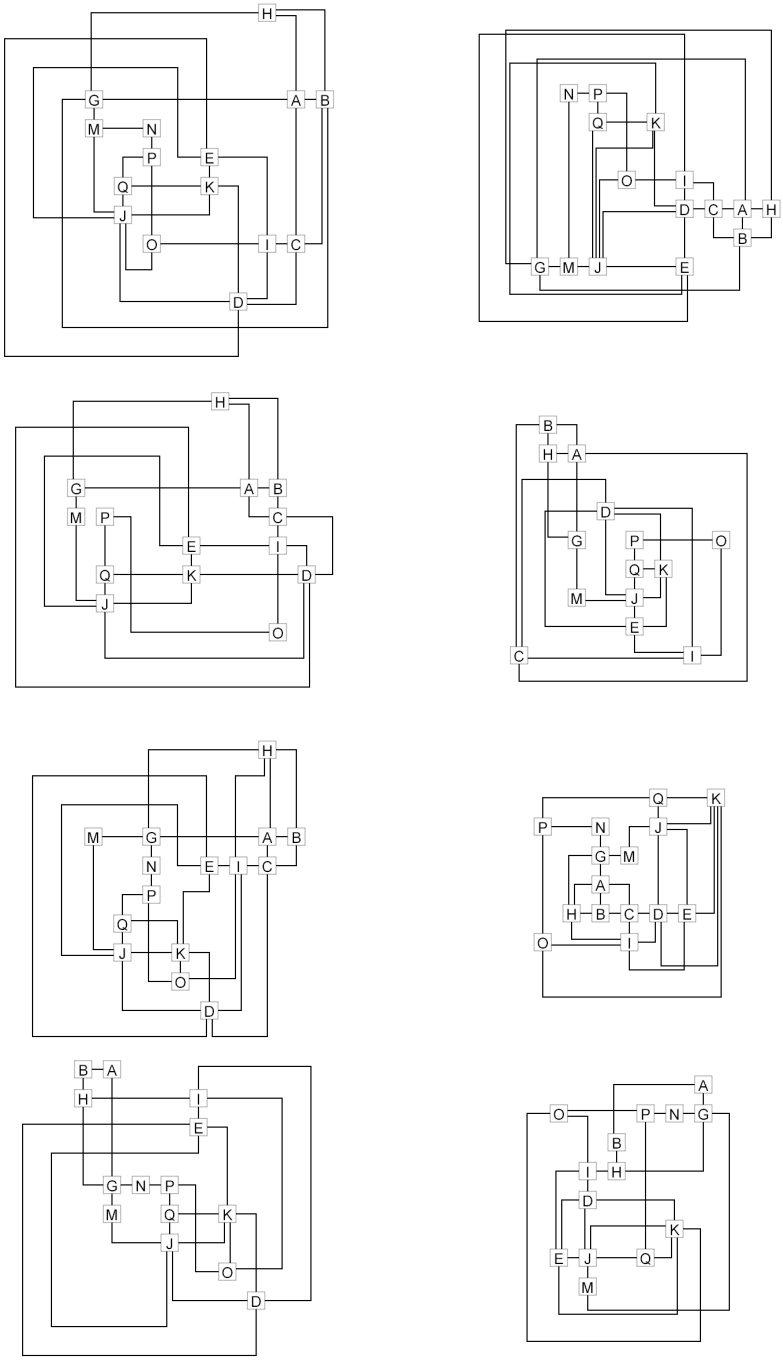
Social Network

Figure 8.12 shows two sequences, each consisting of four drawings of graphs out of a sequence representing a social network. Nodes represent people and edges a relation between them.

From graph 1 to graph 2 person N leaves the group and the relations between J and O, and G and B respectively are broken off. From graph 2 to graph 3 N joins the group again and has relations to G and P. From graph 3 to graph 4 C leaves the group.

Figure 8.12(a) shows the resulting drawings using a large backbone and a small threshold for the angle and bend metrics ($\delta_I = 0,1$; 20% of the angles and 50% of the bends in a graph and its predecessor can change without resulting in a new flow computation). There are some sets of nodes preserving their neighborhood: $\{H, A, B, C, I\}$,$\{Q, J, P\}$, and $\{G, M, A\}$ over the whole sequence, $\{G, N, P, Q\}$ and $\{K, O, D\}$ in graph 3 and 4. Also the global appearance of the drawings is quite stable. Thus, the users mental map can be preserved.

Figure 8.12(b) shows the resulting drawings without using a backbone and large thresholds for the angle and bend metrics (90% and 100% repectively). In this sequence we try to minimize the number of bends in the drawings (the original idea of the network flow algorithms) by increasing the parameter γ for optimal static drawings. The drawings have less bends than the drawings in Figure 8.12(a): graph 1 has only 23 instead of 25, graph 2 has only 17 instead of 21, graph 3 has only 16 instead of 25, and graph 4 has only 17 instead of 20 bends. Due to the lower bend number they are also more compact. It is hard to identify stable sets of nodes: $\{A, B, C, H\}$ is relatively stable in drawings 1 to 3 but the position in the shape is changing.



(a) graph 1 (b) graph 2

Figure 8.12: Social Network.

8.3 Evaluation

In the previous sections we have seen that it is possible to compute drawings of a sequence of graphs which either emphasize the global stability or the local quality – at least according to the metrics and aesthetic criteria. But the question arises if this is also accurate for a real user, that is if the concepts of metrics and aesthetic criteria, which are the computational crutches to substitute real models of human cognition, are suitable. Purchase [PCJ96] performed an evaluation for aesthetic criteria of static graph drawings, Bridgeman [BT01] studied the similarity measures for orthogonal graph drawing, but for the effectiveness of preserving the mental map by restricting changes or using graph animations, there is no published evaluation available so far.

One of the reasons could be that there are many parameters to take into account (for more background information see [Kos94]):

1. Concerning the data:
   - with or without semantics
   - ordered or unordered
2. Concerning the graph:
   - number of nodes and edges
   - color, length, and thickness of the edges
   - color, shape, and size of the nodes
   - number of bends
   - minimal and maximal distance between two nodes
   - polylined, straight-lined, or curved edges
   - bounding box of the drawing
3. Concerning the animation:
   - number of animation steps
   - speed of animation
   - number of deleted nodes, edges, and bends
   - number of inserted nodes, edges, and bends
   - number of moved nodes, edges, and bends
   - positions of the deleted nodes, edges, and bends

- positions of the inserted nodes, edges, and bends
- positions of the moved nodes, edges, and bends
- type and distance of movements

4. Concerning the user:
   - limited short-term memory
   - perception and differentiation of details
   - previous knowledge of graphs
   - quality of sight (glasses)
   - motivation
5. Concerning the presentation medium (computer):
   - size and type of monitor
   - refreshing rate

Nevertheless we started a collaboration with psychologists. The effectiveness of the resulting animations is currently being studied as part of a master thesis in psychology at the Catholic University Eichstätt.

Chapter 9

Related Work

In this chapter related work is described. It is divided into two main sections. One section deals with the problem of dynamic drawing of a graph, that is given different drawings of the same graph, preserve the mental map by communicating the changes. The other sections concerns the problem of drawing of dynamic graphs, that is drawing of graphs which change over time by adding or deleting nodes and edges.

9.1 Dynamic Drawing of a Graph

The algorithms in this thesis for foresighted graph drawing with tolerance try to minimize the changes between successive drawings of a graph sequence, but they do allow changes up to a predefined threshold to improve the local quality of a drawing.

There is an evidence that animation in user interfaces can help people to interact more efficiently with information visualization systems [BB99, DK97]. Further, there is an agreement on the conjecture that this is also true for the special case of systems which visualize dynamically changing graphs. Therefore, to communicate changes between subsequent graphs we use the technique of graph animation to create smooth transformations which help the users in maintaining or quickly update their mental map.

The straightforward method to transform one drawing into another using linear interpolation often yields animations of poor quality, as illustrated in Figure 9.1. A better animation between the same graphs is shown in Figure 9.2.

Friedrich [FH01, Fri02] introduces a formal model describing animations, informal and formal criteria and measures for evaluating the quality of animations as well as a general framework for specifying and implementing

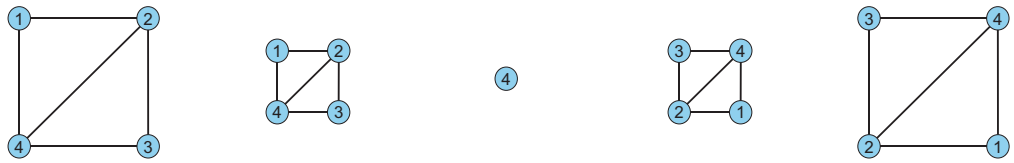


Figure 9.1: When linear interpolation is used for the animation from (a) to (e), all nodes meet in the middle and the drawing collapses at one point (c).

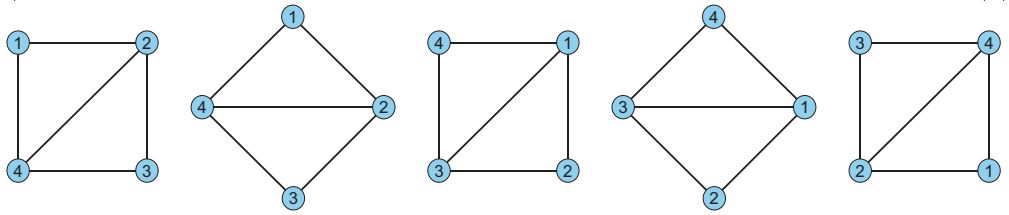


Figure 9.2: When rotation is used for the animation from (a) to (c), the changes in the drawing are more clearly communicated.

animation methods. Furthermore, he presents new approaches to automatically compute animations from given initial and target graph drawings. Some of them will be presented in the following sections.

9.1.1 Quality Measures for Graph Animation

Friedrich proposes the following criteria (among others) to measure the quality of a graph animation:

- The path of the nodes and edges should be smooth
- Uniform node movement: If the distance between two nodes in the initial frame is similar to their distance in the target frame, this distance should be preserved during the animation.
- Constant edge length: If the length of an edge in the initial frame is similar to its length in the target frame, then the length of the edge should be preserved during the animation.
- Minimize edge crossings: It should be avoided to introduce new edge crossings during the animation.

- Maintain a minimum distance between nodes which do not move uniformly: If nodes lie close to each other, then it is more difficult to follow their individual movements.

- Maximize symmetry: Since symmetrical movements are easier to follow, the symmetry should be maximized.

- Display of non-existing structures should be avoided (see Figure 9.3)

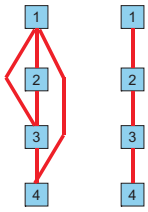


Figure 9.3: Example of misleading drawing.

In the following two naive methods and two advanced methods for graph animation are introduced and shortly discussed.

9.1.2 Naive Methods for Graph Animation

Direct Linear Interpolation

The *Direct Linear Interpolation* method can produce good animations for simple changes between graph drawings. In trivial cases, the *Direct Linear Interpolation* method can even find an optimal solution. However, in the majority of cases the resulting animation is often rather poor.

Orthogonal Interpolation

The *Orthogonal Interpolation* method moves the nodes of the graph in parallel to the y-axis and subsequently in parallel to the x-axis. This induces the illusion of a rotation rigid three-dimensional object. The animation is perceived as aesthetically very pleasing. However, it is doubtful whether this kind of animation is actually increasing the users ability in maintaining or adjusting the mental map. Especially for changing graphs, the illusion of a constant three-dimensional object can be counter-intuitive.

9.1.3 Linear Regression Analysis Method

The *Linear Regression Analysis* 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 close as possible to their final positions. To this end the method uses affine transformation which consists of four operations: translation, scaling, rotation, and shearing. In most cases it is not possible to move all nodes exactly to their final positions. Therefore the method uses linear interpolation or an adjusted force-directed approach (forces are used to attract nodes to their final positions) to move the nodes to their final positions.

The linear regression analysis method produces good results in many cases. It is designed to identify and animate changes between drawings of graphs which can be described by affine linear transformations. These especially include transformations with a significant rotation component.

9.1.4 Motion Cluster Analysis Method

The linear regression analysis fails if certain nodes or whole subgraphs have to be moved in different directions. The *Motion Cluster Analysis* solves this problem by introducing clusters. Nodes performing similar movements are assigned to the same cluster. Friedrich proposes two different heuristics to compute appropriate clusters:

- K-Means Clustering: *K-Means Clustering* is a well known method used in various fields. It first creates an initial partitioning $P = P_1, \ldots, P_k$ 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.

  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 closest to its endpoint. The number of clusters is restricted to ten because too many clusters, which represent movements in the animation, would be hard to follow. The results of this approach depends heavily on the initial partitioning.

- Distance-Based Clustering: The *Distance-Based Clustering* methods respect the distances of the nodes in the initial or in the final drawing. The idea is that nodes positioned close to each other will also perform

similar movements. This method uses a delaunay-triangulation with edge elimination to build the clusters: First, the triangulation of the set of nodes is computed and every triangle is a partition. Then triangles next to each other are merged by removing the edge between them if they have similar transformations. Here the choice of the threshold is important which determines if transformations are similar.

It is also possible to combine these two approaches: Compute a distance based clustering and use it as an initial partitioning for the k-means clustering.

9.2 Drawing of Dynamic Graphs

This section presents the basic ideas of a number of dynamic graph drawing algorithms which have been categorized according to the kind of drawing they produce.

Most work on dynamic graph drawing [Bra01b] is related to the online problem, which means that only information about the previous graphs in a sequence is used for computing a drawing.

9.2.1 General Frameworks

Brandes and Wagner [BW97a, BW97b] present a generic framework that uses a Bayesian perspective to state a cost model representing the trade-off between local quality and dynamic stability. With X being the current drawing, and Y representing the previous drawing, the aim is to search for a current drawing X that maximizes

$$P(X = x \mid Y = y) = \frac{P(Y = y \mid X = x) \cdot P(X = x)}{P(Y = y)}$$

where $P(X = x)$ is the static cost function for the current drawing, and $P(Y = y \mid X = x)$ represents the cost for the difference between the previous and the current drawing.

The framework is independent of a drawing algorithm and also of the difference metric used. The conditional probability for a drawing depends on those of the preceding ones, that is no look-ahead in the sequence is available.

Böhringer and Paulisch [BP90] transfer the dynamic graph drawing problem – aesthetic criteria as well as criteria preserving the mental map – into a set of linear constraints and solve it applying constraint propagation. It is possible to assign priorities to the constraints to resolve inconsistencies

between aesthetic criteria. An additional advantage of this approach is that user constraints (like node A should be above node B) can also be stated as linear constraints and thus easily be integrated.

9.2.2 Force-Directed Drawings

Brandes and Wagner [BW97a] present two ways to adapt the force-directed model to deal with the stability criteria using their Bayesian framework. One of them considers the change of absolute node positions as distance criterion between drawings, which translate nicely into introducing additional forces keeping the nodes at their position in the previous drawing (springs with natural length zero). The second approach considers the change of relative node positions instead of the absolute positions. This is achieved by emphasizing the forces between nodes that are contained in both drawings - the current and the previous. In other words, the invariant parts of the graphs are connected by a stiffer structure than new or changed parts. It is also possible to take the history of the sequence into account: by cumulating the stiffening effect, the longer a relation existed, the less will it be changed.

Eades et al. [ECH97] present an approach which animates the movement of the nodes according to the forces to make the changes more gradual. However, even a small change to a graph could lead to a quite different equilibrium state and the mental map could not be preserved.

9.2.3 Hierarchical Drawings

Böhringer and Paulisch [BP90] show how the Sugiyama heuristic can be modeled in terms of constraints, and how it can then be adapted to preserve dynamic stability. In the layer assignment step for each edge (u, v) a constraint is introduced stating node u should be placed above node v. In the crossing reduction step the barycenter ordering is used to derive constraints determining whether a node should be left or right of another node.

When the graph is modified, stability constraints are derived from the previous drawing. They state:

1. the ordering of the nodes in each layer

2. that nodes which have been on the same layer in the previous drawing should also be on the same layer in the current drawing.

Nodes close to a change in the graph, that is nodes in the neighborhood of the change, are removed from these constraints and allowed to move freely.

By setting the size of the neighborhood it is possible to influence the emphasis on dynamic stability.

North [Nor96, NW02] presents another adaptation of the Sugiyama heuristic: The DynaDAG system allows interactive changes to the graph structure - insert, optimize, or delete single nodes or edges. Nodes are initially placed on the highest possible layer and could be moved down when it becomes necessary by an insertion of another node or edge. Then nodes are moved down layer by layer, positioned in each layer at its median position (according to its adjacent edges). When the nodes are finally placed the adjacent edges are adjusted (shrunk or stretched if its span changed and moved to the new bends). New edges are routed by a heuristic. The final coordinates are computed by a linear program with a linear penalty for moving a node from its position in the previous drawing.

9.2.4 Orthogonal Drawings

Brandes and Wagner [BW97b] demonstrate how the minimum cost flow approach suggested by Tamassia can be extended to account to their framework and to minimize the changes of angles at nodes as well as of bends in edges. The idea of that approach is to modify the flow network of the old drawing by adding "residual" arcs in opposite direction to the current flows and by adapting the cost and capacity constraints of these arcs to reflect the additional cost of changing a flow.

Bridegman et al. [BFG+97] present the InteractiveGiotto tool which is an interactive variant of the Giotto tool for computing orthogonal drawings. A user is required to specify the position of new nodes and to sketch the desired edge routing. Then the tool transforms the current drawing into a planar one by replacing each edge crossing and each bend with a dummy node, where the embedding and the edge crossings are preserved. This graph is then optimized applying a variant of a minimum cost flow approach.

Papakostas and Tollis [PT98] propose two algorithms for orthogonal planar graphs of maximum degree four for two different scenarios. In both cases, it is tried to minimize the number of bends in the drawing.

In the no-change scenario, the current drawing may not be changed when adding new nodes with adjacent edges. The approach here is to start with an empty drawing and to add nodes sequentially one after another. New nodes are always placed outside of the current drawing area.

In the relative coordinates scenario, the current drawing may be modified by introducing a limited number of rows and/or columns anywhere in the current drawing.

9.2.5 Offline Problem

All approaches presented so far consider the online problem. The only approach apart from this thesis that considers the offline problem, that is all graphs in the sequence, is TGRIP [CKN+03, EHK+03] which is an extension of the spring embedder GRIP for large graphs. The basic idea is very intuitive: time is modeled by springs in the third dimension. To this end for each graph of the sequence a drawing in a 2D plane is computed. Nodes in the individual drawings representing the same node of the sequence in subsequent graphs are connected by additional springs, but each node can only move within the 2D plane to which it belongs. In contrast to the approach presented in this work, the approach does not allow using different mental map metrics, because the metrics is built into the heuristic for minimizing the forces and cannot be applied to other drawing methods, for example hierarchical or orthogonal.

Chapter 10

Conclusions

In this thesis we introduced a framework for offline drawing of dynamic graphs. To the best of our knowledge this is the first framework for the offline problem - there exist only other frameworks for the online variant of the problem. In this chapter we conclude the achievements of this thesis and discuss possible directions for future research.

10.1 Achievements

Foresighted Graph Drawing
We presented the motivation and theory behind foresighted graph drawing. This approach projects the graphs of the sequence to the so called super graph, which represents an approximation of the whole sequence, then computes a drawing of the super graph and uses this drawing as a template, that is the drawings of the individual graphs of the sequence are induced by the drawing of the super graph. The advantages of this approach are that it is generic and therefore existing static graph drawing algorithms can be used to compute the drawing of the super graph and the mental map is well preserved because nodes do not change their position at all. The disadvantage is that the local quality of individual drawings is sometimes poor because the quality is restricted by the drawing of the super graph.

Foresighted Graph Drawing with Tolerance
We extended the foresighted graph drawing approach by allowing local optimization of the individual drawings up to a given threshold. We use difference metrics to measure if the drawing resulting from the local adaptation is in the allowed range to guarantee the preserving of the mental map.

Thus, is is possible to trade local quality (of the drawing) for global

stability (of all drawings of the sequence) by using a low or a high threshold respectively: a low threshold ensures global stability, but we have to pay for this with a reduced quality of the drawing and a high threshold allows drawings of better quality, but the stability is reduced.

We discussed several strategies for drawing adjustment which differ with regard to what other graphs are chosen for comparison: the projection graph, the predecessor graph, or the predecessor graph as well as the projection of the successor graph. Furthermore, we presented another strategy which tries to adjust all graphs of the sequence in parallel.

Finally we introduced the concept of a backbone as generalization of the super graph. The backbone does not contain all graphs of the sequence like the super graph, but only nodes of a high importance for the sequence of graphs. The importance of the nodes can be arbitrarily defined. Is is possible to use statistical information, for example nodes contained in many graphs of the sequence are more important than nodes only contained in few graphs, or it is possible as well to use semantical information if they are available.

The advantage of the tolerant version of foresighted graph drawing is that it is possible to decide what is more important for a specific graph sequence: emphasizing the quality of the individual drawings or preserving the mental map. The disadvantage is that it is no longer possible to use a standard static drawing algorithm because this approach requires an algorithm which allows to compute a drawing by adjusting an existing one.

Adjusting Drawing Algorithms to fit into the framework

While implementing tolerant foresighted drawing for a force-directed approach was relatively straight forward, applying the approach to orthogonal and hierarchical layout turned out to require many more changes to the static layout algorithms.

The force-directed approach fitted well in our framework because the possibility to adjust a given drawing is already built in. The drawing to be adjusted is taken as initialization for the algorithm and the amount of allowed changes can be modeled using simulated annealing: if the adjusted drawing changed too much the temperature is annealed and a new drawing is computed.

Hierarchical and orthogonal approaches work both in phases, and we had to introduce new metrics which work on the intermediate results of these phases instead of on the final drawings. When the mental distance of two intermediate results exceeds a given threshold, then we restrict the search space either locally, that is for some nodes or edges, or globally, that is for all nodes or edges.

For the hierarchical approach we computed a global ranking for the backbone and introduced rank metrics to adjust the rankings of the individual graphs up to a given threshold. For the crossing reduction phase we introduced an adjusted sorting approach called smooth sorting which allows to restrict the number of changes resulting from the sorting. For the orthogonal approach we adjusted the computation of the quasi orthogonal shape by using a parameterized version of the network flow computation.

We used two different kinds of restrictions:

- **Global restrictions**: For spring embedding, the global temperature was reduced, which resulted in allowing fewer position changes of all nodes. Similarly, for hierarchical layout the smooth sort parameter influences all nodes in the sorting phase.

- **Local Restrictions**: In the ranking phase of the hierarchical layout, we fix the rank of the not yet fixed nodes of highest importance. Thus, all remaining nodes can still change their ranks. For orthogonal layout the metrics, in fact, also gives a hint what to restrict. As a side-effect of computing the quasi-orthogonal-shape metrics, we do get a set of edges for which we can increment the corresponding parameters of one or more of these edges, that is restrict the number of angle and bend changes.

Implementation and Applications
The framework has been implemented in Java and is available as a standalone application as well as a web application. We discussed the influence of different parameters and applied our approach to applications from different domains, like algorithm animation, word collocation graphs, call graphs, and social networks.

10.2 Future Work

This section describes possible directions for future research. There are three major directions:

- evaluate the effectiveness of preserving the mental

- improve the graph animation technique for changing graphs

- enhance the framework for offline drawing of dynamic graphs

10.2.1 Evaluate the Effectiveness

The framework presented in this thesis lays the foundation for dealing with the problem of offline drawing of dynamic graphs. By choosing an appropriate drawing algorithm and an appropriate difference metric with an associated threshold, it is possible to compute drawings of a sequence of graphs which either emphasize the global stability or the local quality – at least according to the metrics and aesthetic criteria.

But the question arises if this is also accurate for a real user, that is if the concepts of metrics and aesthetic criteria, which are the computational crutches to substitute real models of human cognition, are suitable. Purchase [PCJ96] performed an evaluation for aesthetic criteria of static graph drawings, but for the effectiveness of preserving the mental map by restricting changes or using graph animations, there is no published evaluation available so far.

Therefore, it seems to be important to get a clearer concept of how changes in the drawing influence a user's mental map. This should be examined by user studies to learn about importance of the many different criteria suggested to preserve the mental map.

10.2.2 Graph Animation

Another possible direction for future research concerns the graph animations. Friedrich [FH01, Fri02] investigated the problem to communicate changes between two different drawings of the same graph and proposed to use fade in and fade out operations for inserted or deleted objects if the graph changes. But the users apparently cannot perceive all changes, especially if they are located at different places in a drawing of a graph, using this fading approach.

This approach also fails, if there are too many changes between two graphs – the user just looses the overview. For that, a possible solution would be to build different clusters of the changes and to communicate the changes in several steps. This would avoid to overwhelm the user with too many changes, but the problem occurs that the user could think that there are intermediate graphs which do not exist.

10.2.3 The Framework

A further possible direction for research concerns dealing with the knowledge about the future in the graph sequence. Our first approach was to project the whole sequence of graphs to a super graph which contains all information given in the sequence and therefore approximates the knowledge of the future.

Then we introduced the backbone as a generalization of the super graph to allow to take into account that nodes are of different importance for the sequence. A further improvement would be to consider an evolving backbone, that is a backbone changing over time. Then it would also be possible to model the fact that the importance of nodes in the sequence can change over time: a node which is very important in the beginning of the sequence does not have to be important at the end as well – this especially applies in long sequences.

A more general approach to handle long sequences would be to split the sequence in several small sequences using a sliding time window: for every graph of the sequence consider the graph itself and its n predecessor and successor graphs. Thus, each graph has its own backbone which approximates its limited past and future.

Finally, for sequences containing large graphs it would be useful to investigate a focus context interface which allows to concentrate on a specific change while providing the necessary context of the graph. The context should also contain information of the past or future of the concerned nodes and edges. A first idea to achieve that would be to integrate the concerned parts of the predecessor or successor drawing into the current one.

Bibliography

[BB99] B. Bederson and A. Boltman. Does Animation Help Users Build Mental Maps of Spatial Information. In *Info Vis '99 Proceedings*, pages 28–35. IEEE, 1999.

[BEKW02] U. Brandes, M. Eiglsperger, M. Kaufmann, and D. Wagner. Sketch-Driven Orthogonal Layout. In *Proc. of Graph Drawing 2002*. Springer LNCS 2528:1-11, 2002.

[BETT99] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing – Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[BFG+97] S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and R. Vismara. INTERACTIVE GIOTTO: An Algorithm for Interactive Orthogonal Graph Drawing. In *Proceedings of 5th International Symposium on Graph Drawing GD'97*, pages 303–308. Springer LNCS 1353, 1997.

[BFN85] C. Batini, L. Furlani, and E. Nardelli. What is a Good Diagram? A Pragmatic Approach. In *Proc. 4th International Conference on the Entity Relationship Approach*, 1985.

[Bir05] P. Birke. *Orthogonales Zeichnen dynamischer Graphen*. Diplomarbeit, University of Saarland, Saarbrücken (Germany), to appear, 2005.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Macmillan, London, 1976.

[BM01] O. Bastert and C. Matuszewski. Layered Drawings of Digraphs. In *Drawing Graphs [KW01]*. Springer, 2001.

[BNT86] C. Batini, E. Nardelli, and R. Tamassia. A Layout Algorithm for Data-Flow Diagrams. *IEEE Transactions Softw. Eng.*, SE-12(4):538–546, 1986.

[BP90] K.-F. Böhringer and F. N. Paulisch. Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms. In *Proceedings of the ACM Human Factors in Computing Systems Conference (CHI'90)*, pages 43–51, 1990.

[BPCJ95] S. Bhanji, H. C. Purchase, R. F. Cohen, and M. James. Validating Graph Drawing Aesthetics: A Pilot Study. Technical Report 336, University of Queensland, Department of Computer Science, 1995.

[Bra01a] Ulrik Brandes. Drawing on Physical Analogies. In *Drawing Graphs [KW01]*. Springer, 2001.

[Bra01b] J. Branke. Dynamic Graph Drawing. In *Drawing Graphs [KW01]*. Springer, 2001.

[BT98] S. Bridgeman and R. Tamassia. Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms. In *Proceedings of 6th International Symposium on Graph Drawing GD'98*. Springer LNCS 1457, 1998.

[BT01] S. Bridgeman and R. Tamassia. A User Study in Similarity Measures for Graph Drawing. In *Proceedings of 8th International Symposium on Graph Drawing GD'00*. Springer LNCS 1984, 2001.

[BW97a] U. Brandes and D. Wagner. A Bayesian paradigm for dynamic graph layout. In *Proc. of Graph Drawing 1997*. Springer LNCS 1353:236-247, 1997.

[BW97b] U. Brandes and D. Wagner. Dynamic Grid Embedding with few Bends and Changes. In *Proceedings of the 9th Annual International Symposium on Algorithms and Computation (ISAAC'98)*. Springer LNCS 1533:89-98, 1997.

[CKN+03] C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A System for Graph-based Visualization of the Evolution of Software. In *Proc. of ACM Symposium on Software Visualization SOFTVIS'03*, San Diego, 2003. ACM SIGGRAPH.

[Den91] M. Dennis. *Image and Cognition*. Harvester Wheatsheaf, New York, 1991.

[DETT94] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.

[DG02] S. Diehl and C. Görg. Graphs, They are Changing – Dynamic Graph Drawing for a Sequence of Graphs. In *Proceedings of Graph Drawing 2002*. Springer LNCS 2528:23-30, 2002.

[DGK00] S. Diehl, C. Görg, and A. Kerren. Foresighted Graphlayout. Technical Report A/02/2000, FR 6.2 - Informatik, University of Saarland, December 2000. http://www.cs.uni-sb.de/tr/FB14.

[DGK01] S. Diehl, C. Görg, and A. Kerren. Preserving the Mental Map using Foresighted Layout. In *Proceedings of Joint Eurographics – IEEE TCVG Symposium on Visualization VisSym'01*. Springer Verlag, 2001.

[DH96] R. Davidson and D. Harel. Drawing Graphics Nicely Using Simulated Annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996.

[DK97] M. Donskoy and V. Kaptelinin. Window Navigation with and without Animation: A Comparison of Scroll Bars, Zoom and Fisheye View. In *Proceedings of Extended Abstracts of Human Factors in Computing Systems (CHI 97)*, pages 279–280. ACM Press, 1997.

[DK00] S. Diehl and A. Kerren. Increasing Explorativity by Generation. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000*. AACE, 2000.

[DK01] S. Diehl and A. Kerren. Levels of Exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001*. ACM, 2001.

[Ead84] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42, 1984.

[ECH97] P. Eades, R. F. Cohen, and M. L. Huang. Online Animated Graph Drawing for Web Navigation. In *Proc. of Graph Drawing 1997*. Springer LNCS 1353:330-335, 1997.

[EHK+03] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. GraphAEL: Graph Animations with Evolving Layouts. In *Proc. of Graph Drawing 2003*. Springer LNCS 2912:98-110, 2003.

[EK86] P. Eades and D. Kelly. Heuristics for Reducing Crossings in 2-layered Networks. *Ars combinatoria*, 21:89–98, 1986.

[EK02] M. Eiglsperger and M. Kaufmann. Fast Compaction for Orthogonal Drawings with Vertices of Prescribed Size. In *Proceedings of Graph Drawing 2001*. Springer LNCS 2265:124-138, 2002.

[ELMS91] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the Mental Map of a Diagram. Technical Report IIAS-RR-91-16E, Fujitsu Laboratories Ltd., Japan, 1991.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Potomac, Maryland, 1979.

[FE02] C. Friedrich and P. Eades. Graph Drawing in Motion. *Journal of Graph Algorithms and Applications*, 2002.

[FH01] C. Friedrich and M. Houle. Graph Drawing in Motion II. In *Proceedings of Graph Drawing 2001*, pages 220–231. Springer LNCS 2265, 2001.

[FLM95] A. Frick, A. Ludwig, and H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. In *Proc. of Graph Drawing 1994*. Springer LNCS 894:388-403, 1995.

[For] The GraphML File Format. Project Homepage. http://graphml.graphdrawing.org.

[FR91] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-Directed Placement. *Software – Practice and Experience*, 21:1129–1164, 1991.

[Fri97] A. Frick. Upper Bounds on the Number of Hidden Nodes in Sugiyama's Algorithm. In *Proc. of Graph Drawing 1996*. Springer LNCS 1190:160-183, 1997.

[Fri02] C. Friedrich. *Animation in Relational Information Visualization*. Dissertation, University of Sydney, 2002.

[GAN] GANIMAL. Project Homepage. http://www.cs.uni-sb.de/GANIMAL.

[GBPD04] C. Görg, P. Birke, M. Pohl, and S. Diehl. Dynamic Graph Drawing of Sequences of Orthogonal and Hierarchical Graphs. In *Proc. Graph Drawing 2004*. Springer LNCS 3383, 2004.

[Gib80] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, 1980.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. Freeman and Company, 1979.

[GJ83] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions Softw. Eng.*, 19:214–230, 1993.

[GW86] P. Gould and R. White. *Mental Maps*. Allen and Unwin Inc., Winchester, Mass., USA, 1986.

[GW05a] C. Görg and P. Weißgerber. Detecting and Visualizing Refactorings from Software Archives. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, Missouri, U.S., 2005.

[GW05b] C. Görg and P. Weißgerber. Error Detection by Refactoring Reconstruction. In *Proceedings of International Workshop on Mining Software Repositories (MSR 2005)*, St. Louis, Missouri, U.S., 2005.

[Har72] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1972.

[HMM00] I. Herman, G. Melancon, and M. S. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[JM97] M. Jünger and P. Mutzel. 2-Layer Straightline Crossing Minimazation: Performance of Exact and Heuristic Methods. *Journal of Graph Algorithms and Applications*, 1(1):1–25, 1997.

[Kos94] S. M. Kosslyn. *Elements of Graphs and Design.* W. H. Freeman, 1994.

[KW01] M. Kaufmann and D. Wagner, editors. *Drawing Graphs – Methods and Models*, volume 2025 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[Lan] The Java Programming Language. Project Homepage. http://java.sun.com.

[Lin92] X. Lin. *Analysis of Algorithms for Drawing Graphs.* Dissertation, University of Queensland, 1992.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms.* Springer-Verlag, Volumes 1-3, 1984.

[MELS95] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[NC88] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorihtms.* Ann. Discrete Math., 32, 1988.

[Nor96] S.C. North. Incremental Layout in DynaDAG. In *Proc. of Graph Drawing 1995.* Springer LNCS 1027:409-418, 1996.

[NW02] S.C. North and G. Woodhull. Online Hierarchical Graph Drawing. In P. Mutzel, J. Jünger, and S. Leipert, editors, *Proceedings of 6th International Symposium on Graph Drawing GD 2001.* Springer LNCS 2265, 2002.

[PCJ96] H.C. Purchase, R.F. Cohen, and M. James. Validating graph drawing aesthetics. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Computer Science*, pages 435–446. Springer-Verlag, 1996.

[Poh05] M. Pohl. *Hierarchisches Zeichnen dynamischer Graphen.* Diplomarbeit, University of Saarland, Saarbrücken (Germany), 2005.

[Pro] The Graph Animation Project. Project Homepage. http://rw4.cs.uni-sb.de/ diehl/ganimation.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer Verlag, New York, NY, 1985.

[PT98] A. Papakostas and I.G. Tollis. Interactive Orthogonal Graph Drawing. *IEEE Transactions on Computers*, 47(11), 1998.

[Pur97] H. C. Purchase. Which Aesthetic has the Greatest Effect on Human Understanding. In G. DiBattista, editor, *Graph Drawing (Proc. GD '97)*. Springer LNCS 1027:248-290, 1997.

[Pur00] H. C. Purchase. Effective Information Visualization: A Study of Graph Drawing Aesthetics and Algorithms. In *Interacting with computers, Vol. 13(2), 477-506*, 2000.

[San96] G. Sander. *Visualization Techniques for Compiler Construction.* Dissertation (in german), University of Saarland, Saarbrücken (Germany), 1996.

[SM95a] K. Sugiyama and K. Misue. A Simple and Unified Method for Drawing Graphs: Magnetic Spring Algorithm. In *Proc. of Graph Drawing 1994*. Springer LNCS 894:364-375, 1995.

[SM95b] K. Sugiyama and K. Misue. Graph Drawing by Magnetic-Spring Model. *Journal of Visual Lang. Comput.*, 6(3):1–25, 1995.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical Systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.

[Tam87] R. Tamassia. On Embedding a Graph in the Grid with the Minimum Number of Bends. *SIAM Journal on Computing*, 16(3):538–546, 1987.

[TDB88] R. Tamassia, G. DiBattista, and C. Batini. Automatic Graph Drawing and Readability of Diagrams. In *IEEE Trans. Syst. Man. Cybern., SMC-18, no. 1, 61-79*, 1988.

[Tol48] E. Tolman. Cognitive maps in rats and men. In *Psychological Review, Vol. 55, 189-208*, 1948.

[UK96] U.Fößmeier and M. Kaufmann. Drawing high Degree Graphs with low Bend Numbers. In *Proceedings of Graph Drawing 1995*. Springer LNCS 1027:254-266, 1996.

[WM96] Reinhard Wilhelm and Dieter Maurer. *Compiler Design: Theory, Construction, Generation.* Addison-Wesley, 2nd printing edition, 1996.

[Zim05] S. Zimmer. *Visualisierung layouteter dynamischer Graphen mit SVG*. Bachelor-arbeit, University of Saarland, Saarbrücken (Germany), 2005.