UNIVERSITÄT DORTMUND

Endbericht

PG 478

OGDF: An Open Graph Drawing Framework

Lehrstuhl 11

Inhaltsverzeichnis

1 Einleitung

Die Projektgruppe fand statt im Wintersemester 2005/06 und im Sommersemester 2006 am Lehrstuhl 11, Fachbereich Informatik an der Universität Dortmund. Geleitet wurde die PG von Prof. Dr. Petra Mutzel, Markus Chimani, Carsten Gutwenger und Karsten Klein.
In Unterkapitel 1.1 wird der fachliche Kontext erläutert, in dem diese PG stand. Es werden zunächst die Aufgaben und Ziele des automatischen Graphenzeichnens beschrieben, damit sich der Leser schnell in der Thematik der von der PG behandelten Aufgaben zurechtfindet. Der Text ist zum Teil entnommen aus [6]. Zum Andern werden hier auch schon kurz die einzelnen Aufgabenbereiche der PG angesprochen, sowie eine Formulierung dieser, als Minimalziele. In Kapitel 2 wird die Organisation und die zeitliche Planung, bzw. der zeitliche Verlauf der PG geschildert.

Im Anschluss an die allgemeinen und organisatorischen Dinge werden dann in den Kapiteln 2-5 die einzelnen, bearbeiteten Themengebiete im Detail vorgestellt. Dies umfasst jeweils eine kurze Einführung in den speziellen, behandelten Themenbereich, Beschreibung und Motivation der in diesem Bereich gestellten und zu bearbeitenden Aufgaben und Fragestellungen, die konkrete Umsetzung im Rahmen dieser PG, sowie ein kurzes Resümee ï¿½ber die Arbeit.

1.1 Thematik

Die PG 478 – An Open Graph Drawing Framework beschäftigte sich mit ausgewählten Themen zur Erweiterung einer schon existierenden Bibliothek zur Zeichnung von Graphen.

Beim Graphenzeichnen geht es darum, Graphen bzw. Diagramme (die durch Graphen modelliert werden können) nach formalen und ästhetischen Kriterien zu zeichnen. Die Platzierung der Graphenelemente (Knoten, Kanten, Attribute, ...) bezeichnet man als Layout. Ziel ist es, eine übersichtliche Zeichnung automatisch und effizient zu generieren. Diagramme und Graphen treten in der Praxis vielfach und in unterschiedlichen Formen auf:

- Computernetzwerke, soziale Netzwerke
- Geschäftsprozessmodellierung, Organisationsdiagramme
- Softwaresysteme (z.B. UML Klassendiagramme)
- Datenbankschemata

- Flussdiagramme
- Biologische Reaktionswege

Je nach Anwendungsanforderung werden verschiedene Algorithmen zur Berechnung von Layouts herangezogen. Die Komplexität der vielfältigen Verfahren reicht von einfach bis sehr anspruchsvoll. Viele der dabei auftretenden Probleme stellen eine Herausforderung für die aktuelle wissenschaftliche Forschung dar. Das Gebiet des Graphenzeichnens vereint dabei unter anderem Grundlagen aus Algorithmik, Graphentheorie, Kombinatorik und Visualisierung. Im Bereich des interaktiven Zeichnens werden auch Ergebnisse aus psychologischen Studien herangezogen.

Es existiert bereits eine Reihe von Softwarepaketen für das automatische Graphenzeichnen, sowohl aus dem kommerziellen als auch aus dem akademischen Bereich. Eines dieser Pakete ist die C++ Bibliothek AGD (Algorithms for Graph Drawing), die zu Forschungszwecken an den Universitäten Saarbrücken, Köln, Halle, Wien und Dortmund entwickelt wurde. Im Laufe der Jahre haben sich dabei diverse Abhängigkeiten zu kommerziellen Softwarepaketen ergeben, die der weiteren Forschung hinderlich sind. Daher wurde mit der Entwicklung einer freien Alternative begonnen, dem Open Graph Drawing Framework (OGDF), dessen Funktionsumfang allerdings noch relativ beschränkt ist. Das OGDF ist ebenfalls eine C++ Bibliothek, auf die der GoVisual Editor (GDE) aufbaut.

Aufgabe und Ziel dieser Projektgruppe war es, auf Basis des bestehenden Open-GD-Frameworks eine Erweiterung dieser Bibliothek und des Editors in folgenden Bereichen vorzunehmen:

Compound-Graphen: In dieser Graphenklasse existiert die Möglichkeit, dass Knoten andere Knoten enthalten können, was zu einer zusätzlichen hierarchischen Struktur führt. Derartige Konstrukte sind beispielsweise in Organisations- und UML-Diagrammen sehr gebräuchlich, und stellen einen regen Forschungsbereich dar. Daher ist es von besonderer Bedeutung, eine entsprechende Repräsentation im OGDF zu besitzen, um derartige Graphen editieren zu können, und vor allem auch dafür entwickelte Layoutalgorithmen zur Verfügung zu stellen. Ein wichtiger Punkt ist weiterhin, dass derzeit kein allgemeiner Standard zur Speicherung derartiger Graphen existiert. Daher soll auch ein erweiterbares Format entwickelt werden, das möglichst vielen der ständig wachsenden Ansprüche genügt.

Constraints: Ein wichtiges Teilgebiet des Graphenzeichnens ist die Behandlung von sogenannten Constraints. Dies sind zusätzliche Bedingungen, die für eine gute, bzw. sinnvolle Zeichnung erfüllt werden müssen oder sollen, und reichen von geometrischen Constraints (wie z.B. Alignments von Knoten) bis zu durch eine Semantik definierten Einschränkungen (wie z.B. dem Verbot von Kreuzungen bestimmter Kanten). Um ein flexibles Framework für diese Eigenschaften zu schaffen, gilt es einen allgemeinen Constraint-Mechanismus zu entwickeln, mit dem sich Constraints formulieren und auch visualisieren lassen. Für viele derartige Constraints

existieren bereits einzelne Zeichenmethoden, die diese berücksichtigen. Die wichtigsten dieser Methoden sollen im OGDF zur Verfügung stehen.

Dateiformat zur Graphspeicherung: Ein weiters Ziel der Projektgruppe ist es, ein einheitliches Dateiformat zur Graphspeicherung zu entwickeln. Im Hinblick dessen soll eine Graphbeschreibungssprache entwickelt werden, die insbesondere auch Compound-Graphen und Constraints unterstützen soll. Um das entwickelte Format auch im Rahmen des OGDF nutzbar zu machen, sollten zusätzlich Funktionen zum fehlerfreien Laden und Speichern von auf dem entwickelten Format basierenden Dateien ergänzt werden. Dabei galt es insbesondere zunächst noch zu entscheiden, ob das bereits bestehende GML-Format erweitert, oder eine neue Graphbeschreibungssprache entwickelt wird.

Animation: Oft ist es notwendig für einen gezeichneten Graphen ein weiteres alternatives Layout zu berechnen - sei es wegen leichter Änderungen am Graphen selbst, oder um den Schwerpunkt der Darstellung zu verschieben. Aus der Psychologie ist der Begriff der Mental Map bekannt. Damit umschreibt man die Anhaltspunkte im Layout, die sich der Betrachter merkt, um sich im Diagramm zu orientieren. Solche Anhaltspunkte können z.B. die Lage besonders markanter Objekte zueinander oder hervorstechende Muster in der Darstellung sein. Wird nun ein anderes Layout gewählt, sollte sich die Mental Map des Benutzers nicht zu stark ändern, bzw. muss man dem Benutzer dabei helfen, die Tranformation nachvollziehen zu können. Es existeren diverse Forschungsergebnisse wie man eine Darstellung in eine andere transformieren kann, so dass der Anwender diese Änderung leicht verstehen kann. Darauf basierend ist ein Animationsmodul zu entwickeln, das geeignet ist, verschiedene Transformationsmodelle anzuwenden.

Editorkomponente: Ein graphisches Frontend für Graphenzeichenalgorithmen ist weit mehr als eine einfache Zeichenfläche für Knoten und Kanten. Es erfordert sowohl programmiertechnisches als auch wissenschaftliches Geschick, damit das Ergebnis den Anforderungen der Graphenlayout-Community gewachsen ist. Ziel ist also nicht nur die Integration der oben aufgeführten Punkte in ein homogenes System, sondern dieses System auch - mit Verständnis für die derzeitigen Graphenzeichen-Methoden und Weitblick für die kommenden Entwicklungen - möglichst erweiterbar zu gestalten. Im Rahmen dieser PG sollte der bestehende Graph-Editor GDE auch noch um weitere, nützliche Features ergänzt werden.

1.1.1 Minimalziele

Konkret wurden für die gerade beschriebenen im Rahmen dieser Projektgruppe umzusetzenden Aufgaben Minimalziele formuliert, die im Folgenden kurz aufgelistet sind:

1. Lauffähiger Editor
2. Unterstützung von Compounds inklusive mindestens eines Layoutalgorithmus

3. Unterstützung von Constraint-Mechanismen inklusive geeigneter Layoutalgorithmen
4. Unterstützung eines geeigneten Dateiformats (für Compounds, Constraints, Semantik)
5. Entwicklung eines erweiterbaren Mechanismus zur Animation
6. Teilberichte und Abschlussbericht

1.2 Teilnehmer und Organisatoren

Es folgt eine Auflistung der Organisatoren und Teilnehmer der PG.

Organisatoren

- Prof. Dr. Petra Mutzel
- Markus Chimani
- Carsten Gutwenger
- Karsten Klein

Teilnehmer

- Bihui Dai
- Martin Gronemann
- Mathias Jansen
- Wan-Hi Joh
- Jana Ludolph
- Thomas Rothvoß
- Jasmin Smula
- Sebastian Sondern
- Benjamin Stähr
- Hendrik Stroh
- Christian Wolf
- Bernd Thomas Zey

2 Organisation und zeitliche Planung

Dieses Kapitel zeigt auf, wie wir uns als Projektgruppe organisiert haben, um die anstehenden Aufgaben zu bewältigen. Dies umfasst sowohl offizielle Treffen und Veranstaltungen, als auch die interne, eigenständige Organisation und Aufgaben-Koordination der Teilnehmer untereinander, sowie die Beschreibung der uns zur Verfügung gestellten Mittel. Desweiteren wird auch auf den zeitlichen Verlauf der Projektgruppenarbeit eingegangen.

2.1 Seminarphase

Begonnen wurde die PG mit der für Projektgruppen typischen Seminarphase, die vom 17.10.2005 bis zum 31.10.2005 stattfand. Hierbei hielt jeder der Teilnehmer einen Vortrag zu einem bestimmten Thema des Graphenzeichnens. Die Themenvergabe fand bereits am Ende des SS05 statt, bei der die zu referierenden Themen zufällig den einzelnen Teilnehmern zugeordnet wurden.

Die Seminarphase dient dazu, den Teilnehmern einen allgemeinen Überblick über die Thematik zu verschaffen und ein erstes Verständnis dafür zu entwickeln. Die Vortragsserie wurde durch die PG-Betreuer abgeschlossen, indem sie eine Einführung in grundlegende Strukturen der für die Projektgruppe wichtigen Bibliothek OGDF gaben.

Im Folgenden ist die Seminarphase, zeitlich und inhaltlich in tabellarischer Form dargestellt. Im Anschluss sind die Themen der einzelnen Vorträge jeweils kurz zusammegenfasst.

2.1.1 Zusammenfassung der Seminarvorträge

Force Directed Graph Drawing Das Paper „Graph Drawing by Force-Directed Placement“ [9] von Thomas M. J. Fruchtermann und Edward M. Reingold beschäftigt sich mit kräftegerichtetem Zeichnen von Graphen, wobei der Graph als eine physikalische Einheit betrachtet wird (z.B. Stahlringe entsprechen Knoten, Federn entsprechen Kanten). In jeder Iteration werden abstoßende und anziehende Kräfte auf jeden Knoten berechnet. Dann wird die Kraft der Temperatur angepasst, so dass die Kraft (Bewegung) bei kälterer Umgebung geringer wird. Anschließend wird die angepasste Kraft jeweils auf den Knoten angewandt und somit bewegt. In dem Zusammenhang werden u.a. das langsame Abkühlen der Temperatur („simulated annealing“), die Modellierung von physikalischen Mauern, die Rastervariante des Algorithmus und die Auswirkungen auf symmetrische, planare und 3D-Graphen angesprochen.

Zeit	Referent	Thema
Mo, 17.10. 15:00-18:00	*Jana Ludolph:*	Force-Directed Graph Drawing
	Benjamin Stähr:	Hierarchical Graph Drawing
Do, 20.10. 14:00-17:00	*Sebastian Sondern:*	Planarization
	Bihui Dai:	Cluster Planarization
	Mathias Jansen:	Shape Optimization
Mo, 24.10. 15:00-18:00	*Jasmin Smula:*	Constraints in Graph Drawing I
	Wan-Hi Joh:	Constraints in Graph Drawing II
	Thomas Rothvoß	Constraints in orthogonal Graphs
Do, 27.10. 14:00-17:00	*Martin Gronemann:*	Navigation
	Bernd Thomas Zey:	Compounds & Force-Directed
	Hendrik Stroh:	Compounds & hierarchical Layout
Mo, 31.10. 15:00-18:00	*Christian Wolf:*	Animation
	PG-Organisatoren:	OGDF

Tabelle 2.1: Seminarphase – Übersicht

Hierarchical Graph Drawing In dem Paper „A Technique for Drawing Directed Graphs“ von Gansner, Koutsofios, North und Vu [10] erfolgte zunächst eine Einführung zum Thema „Hierarchische Graphen“, sowie zu „ästhetischen Zeichenkriterien“. Hauptaugenmerk des Vortrages lag auf dem Algorithmus von Sugiyama et al. zum hierarchischen Graphenzeichnen. Dieser Algorithmus erzeugt sequentiell in vier Phasen ein Layout zu einem gerichteten Graphen. In der ersten Phase wird jedem Knoten eine ganzzahlige Ebene zugewiesen, danach werden jeweils Knoten auf der gleichen Ebene in eine eindeutige Reihenfolge gebracht. In der dritten Phase werden den Knoten absolute Koordinaten zugewiesen. Zum Schluss werden noch die Kanten eingezeichnet.

Planarization Das Paper „Crossings and Planarization“ von Mutzel, Jünger, Gutwenger, Buchheim und Ebner [12] befasst sich mit dem grundsätzlichen Verfahren, einen Graphen in der Ebene mit so wenig Kantenkreuzungen wie möglich zu zeichnen. Dabei wird in erster Linie auf den von Batini, Talamo und Tamassia entwickelten heuristischen Ansatz „Planarization Approach“ zurückgegriffen. Hier bestimmt man in dem gegebenen Graphen zuerst einen Teilgraphen mit so vielen Kanten wie möglich und versucht dann auf geschickte Art und Weise, die restlichen Kanten wieder so hinzuzufügen, dass möglichst wenig Kantenkreuzungen entstehen. Neben einer statischen Variante, in der beim Einfügen der restlichen Kanten von einer festen Einbettung des Graphen ausgegangen wird, stellt das Paper zusätzlich noch eine dynamische Variante vor. In dieser können beim Hinzufügen einer Kante sämtliche Einbettungen mit Hilfe von SPQR-Bäumen beachtet werden.

Cluster Planarization Das Paper „Planarization of Clustered Graphs “ von G. Di Battista, W. Didimo und A. Marcandalli [1] beschäftigt sich mit einem Planarisierungs-

algorithmus für Cluster-Graphen, wobei ein Planarisierungsalgorithmus für einen Cluster-Graphen C mit n Knoten, m Kanten und c Clustern vorgeschlagen wurde. Inzwischen wurde ein Algorithmus für die Konstruktion eines „Spann-Baums“ in C entwickelt.

Shape Optimization In dem Paper „Computing Orthogonal Drawings with the Minimum Number of Bends“ von Paola Bertolazzi, Giuseppe Di Battista, Walter Didimo [3], befassen sich die Autoren mit dem orthogonalen Zeichnen von Graphen. Sie beschreiben u.a. einen polynomiellen Algorithmus, basierend auf der Konstruktion eines Flussnetzwerks, der für eine gegebene planare Einbettung eines Graphen, ein orthogonales Layout mit der minimal möglichen Anzahl von Kantenknicken berechnet. Außerdem beinhaltet das Paper auch einen Branch & Bound Algorithmus, der basierend auf dem zuvor für einen Graphen berechneten SPQR-Baum, ein optimales orthogonales Layout berechnet, also ein Layout mit der minimal möglichen Anzahl von Kantenknicken unter allen Einbettungen des Graphen. Desweiteren enthält es statistische Auswertungen in Bezug auf Güte und Laufzeit, und die Beschreibung verschiedener Konventionen im Bereich des orthogonalen Zeichnens von Graphen.

Constraints in Graph Drawing I In dem Paper „Using constraints to achieve stability in automatic graph layout algorithms“ [15] von Kathy Ryall, Joe Marks und Stuart Shieber wird eine Methode vorgestellt, wie Constraints benutzt werden können, um stabile Graphen zu erhalten. Am Beispiel des Sugiyama-Algorithmus wird erklärt, wie man Constraints in einen schon vorhandenen Algorithmus zur Berechnung automatischer Graphenlayouts einfügen kann. Das Paper „An interactive constraint-based system for drawing graphs“ [4] von Karl-Friedrich Böhringer und Frances Newbery Paulisch stellt den Editor GLIDE vor, der zu einem gegebenen Graphen mit Hilfe des Force-Directed Graph Drawing automatische Layouts erzeugt. GLIDE bietet für den Benutzer die Möglichkeit, Constraints einzugeben, die dann in Kräfte in einem physikalischen Modell umgerechnet werden und so das Graphenlayout beeinflussen können.

Constraints in Graph Drawing II In dem Paper „Constrained Graph Layout“ [13] von Weiqing He und Kim Marriott führen die Autoren ein Model zur Layout Gestaltung mittels Constraints ein, das *Constrained Graph Layout Model*. Das Modell stützt sich auf ein modulares Black-Box Konzept, das eine algorithmische Flexibilität garantieren und eine Einbindung der Layout-Stabilität ermöglichen soll. Als Eingabe wird ein Graph gefordert, dessen Layout vom sogenannten *Layout Modul* unter Zuhilfenahme von Constraints und vorgeschlagenen Wertzuweisungen erstellt wird. Trotz der theoretischen Formulierung des Konzepts konnte sich das *Constrained Graph Layout Model* in der Praxis nicht durchsetzen.

Constraints in orthogonal Graph Drawing Der Seminarvortrag, basierend auf „Sketch-Driven Orthogonal Graph Drawing“ [5] von Brandes, Eiglsperger, Kaufmann und Wagner, thematisierte, wie sich ein gegebener Graph orthogonalisieren lässt (d.h. es

werden nur horizontal und vertikal verlaufende Kantensegmente zugelassen), wobei sowohl die Anzahl der Kantenknicke, als auch die Abweichung vom Ursprungsgraphen minimiert werden soll. Dazu wurde zunächst der Algorithmus von Tamassia vorgestellt, der in der Lage ist, mittels der Anwendung eines Minimalkostenflussproblems ein orthogonales Layout mit einer minimalen Anzahl von Kantenknicken zu berechnen. Leider hat dieser Algorithmus den Nachteil, dass er nur auf Graphen mit einem maximalen Knotengrad von höchstens 4 anwendbar ist. Dies führte zum komplexeren Kandinskymodell (auch als Podevsnef-Modell bekannt), das diesen Nachteil beseitigte, was aber mit einem Verlust der effizienten Optimierung bezahlt werden musste. Schließlich wurde eine erneute Erweiterung des Modells vorgestellt, der das Flussnetzwerk noch um Strafkosten für Abweichung vom Ausgangsgraphen erweiterte, so dass nun neben der Minimierung der Kantenknicke auch eine Minimierung der Abweichung erzielt werden konnte.

Navigation Im Paper „Graph Visualisation & Navigation in Information Visualisation: a Survey“ von Herman [14], geht es um Folgendes. Für große Eingabemengen werden an die Repräsentation der Graphen zwei Anforderungen gestellt: Zum Einen muss das Layout dem Zweck entsprechen und auch in annehmbarer Zeit zu berechnen sein. Zum Anderen muss der Benutzer in der Lage sein, durch den Graphen zu navigieren. Hierfür gibt es verschiedene Techniken, um die Arbeit des Benutzers so effizient wie möglich zu gestalten. Dazu zählen verschiedene Werkzeuge, wie Fisheye View, Zoom & Pan und Techniken wie Space-Scale Diagramme. Das Anwendungsgebiet großer Graphen umfasst viele Bereiche wie z.B. die Genforschung, Soziale Netze und Computernetze.

Compounds & Force Directed François Bertault und Mirka Miller präsentieren in dem Paper „An Algorithm for Drawing Compound Graphs“ [2] einen Ansatz für einen auf dem Force-Directed-Prinzip basierenden Algorithmus, um Compound-Graphen zu zeichen.

Ein Compound-Graph wird definiert als Graph, in dem sowohl Adjazenz- als auch Inklusions-Beziehungen ausgedrückt werden können. Dies wird durch die Erweiterung des klassischen Graphen um einen Baum erreicht, welcher die identische Knotenmenge wie der zugrunde liegende Graph hat und die Hierarchie zwischen den Knoten widerspiegelt.

Der vorgestellte Algorithmus *Nuage* bearbeitet zuerst den Compound-Graphen, um ihn anschließend einfacher zeichnen zu können. Dabei wird der Inklusionsbaum traversiert und all jene Kanten werden bearbeitet, welche zwischen Knoten mit unterschiedlicher Tiefe verlaufen. Diese werden auf die Vorgänger der Knoten umgebogen, so dass jede Kante zwischen zwei Geschwistern verläuft. Dieser Preprocessing-Schritt ermöglicht eine rekursive Abarbeitung jedes Compound-Knotens beim Zeichnen. Um den ursprünglichen Kantenverlauf wiederherzustellen, werden beim Zeichnen die Kanten durch ihr altes Gegenstück ersetzt.

Compounds & hierarchisches Layout In seiner Publikation „Layout of Compound Directed Graphs “ [16] stellt Georg Sander ein Verfahren zum Layout von (gerichte-

ten) Compound-Graphen vor. Das Verfahren ähnelt der von Sugiyama und Misue [18] präsentierten Layout-Methode für hierarchische Graphen, liefert jedoch kompaktere Darstellungen, da ein anderer Ansatz im Rahmen der Schichtzuweisung verfolgt wird. Prinzipiell wird versucht, die Knoten so zu positionieren, dass in Folge überschneidungsfreie Rechtecke um die Knoten jedes Compounds gezeichnet werden können und die Anzahl der Kantenkreuzungen minimal ist.

Animation Bei der Veränderung von Graphen/-strukturen durch Layoutalgorithmen ist es für den Ersteller/Betrachter solcher Visualisierungen wichtig, dass er nach der Transformation des Graphen weiterhin alle Zusammenhänge erkennt. Diese so genannte *Mental Map Preservation* wird realisiert, indem man dem Betrachter Veränderungen des Graphen mittels einer Animation ausgehend vom Ursprungsgraphen hin zum Ausgabegraphen des Layoutalgorithmus mitteilt. Ein Hauptbeurteilungskriterium einer gelungenen Animation ist eine einheitliche Bewegung der Graphelemente, also eine gleichzeitige Animation von Knoten und Kanten. Dadurch wird es dem menschlichen Benutzer vereinfacht, Veränderungen nachzuvollziehen, da das menschliche Gehirn eine 2-dimensionale einheitliche Bewegung als eine 3-dimensionale interpretiert und somit einprägsamer ist.
C. Friedrich und P. Eades stellen in „Graph Drawing in motion“ [7] ein Verfahren zur Berechnung solcher einheitlicher Bewegungen auf Basis der linearen Regression vor. Abbildungen werden hierbei durch affine Transformationen beschrieben. In dem auf [7] aufbauenden Artikel „Graph Drawing in motion II“ [8] werden erweiterte Verfahren für Teilgraphanimationen mittels Clusterverfahren behandelt.

2.2 Organisation der PG

2.2.1 Zeitliche Planung und Bildung von Arbeitsgruppen

Im ersten PG-Semester wurde uns die grundsätzliche Vorgehensweise und die Auswahl der anstehenden Aufgabenbereiche, die wir zuerst bearbeiten wollten, von den Betreuern vollkommen frei gestellt. Nach Abschluss der Seminarphase galt es demnach zunächst, uns eigenständig zu organisieren und unser Vorgehen für das erste PG-Semester zu planen. Dazu entschieden wir uns zunächst dafür, mehrere Aufgabenbereiche parallel zu bearbeiten, und demnach Arbeitsgruppen zu bilden, die für je ein ausgewähltes Themengebiet zuständig sind. Die Wahl viel dabei auf die drei Themen **Compounds**, **Dateiformat** und **Constraints**. Diese Bereiche waren in unseren Augen zum Einen die wichtigsten, zum Anderen bot es sich aus Gründen der Einarbeitung in den bereits bestehenden Code an, zunächst OGDF-seitige Aufgaben zu bearbeiten, da die OGDF das Gerüst darstellt, auf der alles aufbaut. GDE-spezifische Themen, sowie die spätere Nutzbarmachung der von uns in der OGDF implementierten Funkionalität, hatten wir planungsmäßig auf das zweite PG-Semester verlegt.

Im ersten Semester galt es also, entsprechende Funktionalität für diese drei Bereiche in der OGDF bereitzustellen, bzw. diese dahingehend zu erweitern.

Als nächstes legten wir dann unter Berücksichtigung individueller Interessen die Einteilung in gleich große Arbeitsgruppen fest. Dabei wurde auch darauf geachtet, dass jede Gruppe zumindest ein Mitglied besitzt, welches sich auch schon im Rahmen der Seminarphase mit dem entsprechenden Themengebiet beschäftigt hat. Die Zuteilung der Teilnehmer zu den drei Gruppen sah wie folgt aus:

Gruppeneinteilung des ersten PG-Semesters

Compounds

- Bihui Dai
- Thomas Rothvoß
- Sebastian Sondern
- Benjamin Stähr

Constraints

- Martin Gronemann
- Mathias Jansen
- Jana Ludolph
- Jasmin Smula

Dateiformat

- Wan-Hi Joh
- Hendrik Stroh
- Christian Wolf
- Bernd Thomas Zey

Am Ende des ersten Semesters waren die OGDF-seitigen Arbeiten bezüglich der drei Arbeitsgruppen weitgehend abgeschlossen, so dass das zweite PG-Semester unter dem Gesichtspunkt „Usability“ stand. Das heißt, der Schwerpunkt der Arbeiten im zweiten Semster war nun, die implementierte Funktionalität auch im GDE nutzbar zu machen und geeignete Benutzer-Schnittstellen zu entwickeln und zu implementieren. Hinzu kamen noch die Themengebiete **Animation** und **GDE-Features**, die zusätzlich bearbeitet wurden. Dazu mussten neue Gruppen gebildet werden.

Die Einteilung der Teilnehmer zu den neuen und alten Gruppen (diesmal mit anderem Schwerpunkt) wurde zu Beginn des zweiten Semsters von den Betreuern übernommen. Das heißt, auch die bis dato bestehenden Gruppen wurden neu besetzt. Die Zuteilung war wie folgt:

Gruppeneinteilung des zweiten PG-Semesters

Compounds

- Sebastian Sondern
- Benjamin Stähr
- Hendrik Stroh
- WanHi Joh

Constraints

- Martin Gronemann
- Jana Ludolph
- Christian Wolf

GDE-Features

- Bihui Dai
- Jasmin Smula
- Bernd Thomas Zey

Animation

- Mathias Jansen
- Thomas Rothvoß

Da zu Beginn des zweiten Semesters die Arbeiten noch nicht in allen Bereichen vollständig abgeschlossen waren, wurden diese erst zu Ende geführt, sodass anfangs manche der Gruppen mit weniger als den oben aufgeführten Teilnehmern starteten. Nach Abschluss der noch abzuschließenden Arbeiten aus dem ersten Semester stießen die entsprechenden Teilnehmer dann zu ihren Gruppen dazu.

2.2.2 Offizielle Treffen und Veranstaltungen

Grundsätzlich war die PG so organisiert, dass jeweils in der Vorlesungszeit des WS05/06 und SS06 wöchentliche Treffen in den Räumlichkeiten des Lehrstuhls 11 stattgefunden haben, an denen alle PG-Teilnehmer und die Betreuer teilgenommen haben. Dies umfasste jeweils einen festen, sowie einen optionalen Zusatztermin, der im Falle wichtiger zu besprechender Dinge zusätzlich wahrgenommen wurde. Diese offiziellen Treffen wurden in erster Linie dazu genutzt, Probleme zu besprechen (sowohl technischer als auch

inhaltlicher Natur) und die anderen Arbeitsgruppen über Fortschritte und den aktuellen Stand der Dinge innerhalb der einzelnen Arbeitsgruppen zu informieren.

Zu manchen vorher abgestimmten Treffen, wurden diese Fortschrittsberichte etwas offizieller gestaltet, indem jede Arbeitsgruppe in einem kurzen Vortrag von ca. 10 Minuten, evtl. unterstützt durch Folien oder Tafelbilder, ihren derzeitigen Arbeitsstand präsentierte. Da die Arbeiten der in einem Semester gebildeten Arbeitsgruppen, aufgrund der meist klar trennbaren Aufgabenbereiche, größtenteils unabhängig voneinander stattfinden konnten, zielten diese Fortschrittsberichte darauf ab, den Informationsfluss nicht nur zwischen Betreuern und Teilnehmern, sondern auch zwischen den einzelnen Arbeitsgruppen zu erhöhen. Besonders im zweiten Semester wurde darauf verstärkt Wert gelegt, da sich dort teilweise Abhängigkeiten zwischen den bearbeiteten Bereichen ergaben, die eine intensivere Koordination erforderten.

Neben diesen offiziellen Treffen arbeiteten die gebildeten Arbeitsgruppen weitgehend autonom. Die Verantwortung für die interne Organisation lag bei jeder Gruppe selbst.

Am Anfang des zweiten PG-Semesters fand ein zusätzliches Treffen statt, bei dem die bis dato gebildeten Arbeitsgruppen ihre im ersten PG-Semester geleistete Arbeit in Semiar-Form präsentierten. Diese Veranstaltung diente auch dem (inhaltlichen) Abschluss des ersten Semesters. Im Anschluss an die Vorträge fand dann die Arbeitsgruppenbildung für das zweite Semester statt.

Das offizielle Ende der PG erfolgte durch einen fachbereichsweiten Abschluss-Vortrag im Rahmen des Diplomanden-/Doktorandenseminars am 19.10.2006, bei dem die Projektgruppe geschlossen die Ergebnisse ihrer geleisteten Arbeit präsentierte.

3 Arbeitsgruppe Animationen

3.1 Einleitung

Im Folgenden wird berichtet, welche Arbeiten im Bereich der Entwicklung und Einbindung eines Animations-Mechanismus in die OGDF und den GDE stattgefunden haben. Dieser Bereich wurde im zweiten Semester der PG (SS06) angegangen und bearbeitet.

Der Stand der Dinge vor Beginn der Projektgruppe sah so aus, dass in der OGDF-Bibliothek keinerlei Strukturen zur Animations-Unterstützung bestanden. Im GDE war die Möglichkeit eingebaut, die Position der Knoten per linearem Übergang vom Quell- zum Ziellayout zu bewegen. Aufgabe dieser Arbeitsgruppe war es, ein Konzept zu entwerfen und umzusetzen, dass OGDF-seitige Klassen-Strukturen für Animations-Algorithmen umfasst. Der GDE sollte dann um ein User-Interface erweitert werden, das dem User die Auswahl und Anwendung von Animationen ermöglicht. Desweiteren sollten dann natürlich auch Animations-Algorithmen implementiert werden, die dann im GDE ausgewählt und eingesetzt werden können.

Wozu Animationen?

Die grundlegende Aufgabe einer Animation besteht darin, zu einem Paar aus gegebenem Ausgangs- und Ziellayout eines Graphens eine Folge von Zwischenlayouts zu berechnen, die einen stetigen Übergang vom Ausgangs- zum Ziellayout darstellen. Die Motivation für diese Funktionalität besteht darin, dass der Benutzer bei einem sprunghaften Wechsel zwischen den Layouts den durch den Graphen dargestellten Zusammenhang verlieren kann. Die sogenannte *Mental Map* ([7]), die sich der Benutzer vom Graphen gebildet hat, würde verloren gehen.

Die Eigenschaften, die eine gute Animation aufweisen sollte, lauten (nach [7])

- Der Benutzer muss in der Lage sein, die Bewegung der Knoten und Kanten nachzuvollziehen
- Die Bewegung des Graphens sollte strukturiert sein
- Die Bewegung sollte keine Strukturen suggerieren, die der Graph gar nicht enthält
- Kantenkreuzungen sollten vermieden werden
- Die Länge der Pfade, auf denen Knoten bewegt werden, sollte minimiert werden

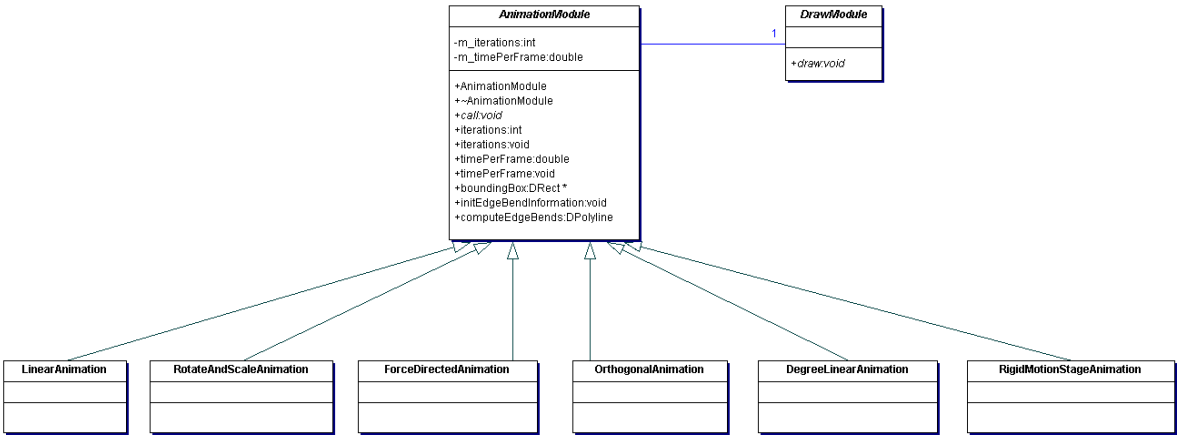

Abbildung 3.1: OGDF-Klassenstruktur zur Animation

3.2 Klassenstruktur

Eine der wichtigsten Anforderungen an den umzusetzenden Animations-Mechanismus war, dass er dazu ausgelegt sein soll, verschiedene Animations-Algorithmen zu unterstützen und insbesondere auch diesbezüglich leicht erweiterbar zu sein. Das erforderte einen modularen Aufbau der Klassenstruktur und eine einheitliche Schnittstelle, so dass sich ein neu implementierter Animations-Algorithmus leicht in die bestehende Struktur integrieren lassen würde. Da die OGDF ja als eine Algorithmen-Bibliothek anzusehen war, sollte es zum Anderen prinzipiell auch möglich sein, die implementierten Animations-Algorithmen anwendungsunabhängig zu gestalten, so dass diese auch von anderen Anwendungen als dem GDE nutzbar würden.

Um diesen Anforderungen gerecht zu werden, war der Entwurf und die Umsetzung folgender Klassenstruktur naheliegend und sinnvoll (siehe Abbildung 3.1):

Die Klasse `AnimationModule` ist eine abstrakte Oberklasse, von der jeder neu implementierte Animations-Algorithmus erben muss. Sie beinhaltet zum einen Attribute und Methoden, die allgemeiner Natur und somit unabhängig von einem speziellen Algorithmus sind. Zum Anderen ist dort auch die Signatur der virtuellen Methode *void call()* definiert, die von einem Animations-Algorithmus implementiert werden muss. Über diese Methode wird die spezielle Animation gestartet, sodass sie sozusagen den Kern eines Animations-Algorithmus darstellt. Auf diese Weise ist eine einheitliche Struktur vorgegeben, die somit den Anforderungen Erweiterbarkeit und Unabhängikeit zur konkreten Anwendung genügt.

Die Schnittstelle zur Anbindung an eine konkrete Anwendung wurde durch die abstrakte Klasse `DrawModule` gebildet. Die Idee dabei war, dass das eigentliche Zeichnen der einzelnen berechneten Frames nicht durch den Animations-Algorithmus selbst, sondern durch eine spezielle auf die konkrete Anwendung zugeschnittene Klasse erfolgen

sollte. Dadurch wurde erreicht, dass ein Animations-Algorithmus frei von anwendungsspezifischem Code bleiben konnte. Die Klasse `DrawModule` enthält nur die Signatur der virtuellen Methode *void draw()*, die für das Zeichnen an sich zuständig ist. Diese muss von einer Implementierung dieser Klasse überschrieben werden und im Hinblick auf die konkrete Anwendung implementiert werden.

3.3 Einbindung in den GDE

Wie im vorangegangenen Kapitel beschrieben, sieht die entworfene Klassenstruktur vor, dass die abstrakte Klasse `DrawModule` anwendungspezifisch implementiert werden muss, um die berechneten Frames der Animation auch auf dem Bildschirm zeichnen zu können. Dies erfolgte im GDE durch die Implementierung der Klasse `FrameDrawer`, die von der Klasse `DrawModule` erbt und dabei insbesondere die virtuelle Methode *draw()* implementiert.

Die Implementierung der Methode *void draw()* ist sehr simpel gehalten. Das übergebene GraphAttributes-Objekt enthält Informationen bezüglich der aktuellen Knoten- und Kantenknick-Positionen. Diese werden iterativ ausgelesen und die jeweils korrespondierenden Zeichenobjekte (NodeObjects und EdgeObjects) gemäß der neuen Werte aktualisiert. Das Zeichnen selbst erfolgt dann mittels eines Aufrufs der Methode *update()*, die all diejenigen Zeichen-Objekte neu auf dem Canvas zeichnet, für die sich die Koordinaten geändert haben.

Durch das Erstellen eines neuen Layouts für einen Graphen, ändert sich im Allgemeinen auch die Größe der rechteckigen Fläche (bounding box), auf der der Graph gezeichnet ist. Dadurch kann es während der Animation dazu kommen, dass Zeichenobjekte aus dem aktuell sichtbaren Bereich „herauswandern“. Deshalb haben wir eine optionale Zoom-Funktion implementiert, die (falls vom Benutzer ausgewählt) dafür sorgt, dass während der gesamten Animation der Focus immer vollständig auf dem Graphen bleibt, der sichtbare Bereich also immer auf die aktuelle Größe der bounding box skaliert wird. Dies ist im Kern durch einen Aufruf der Methode *zoomToContent()* nach jeder Iteration realisiert, die genau diese Funktionalität besitzt.

Das User-Interface zum Zwecke des Aktivierens und Deaktivierens sowie verschiedener Einstellungsmöglichkeiten für die Animation, haben wir in Form eines einfachen Fensters gestaltet, dass über das Menü *Extras...Animation* aufgerufen werden kann. Dies ist in Abbildung 3.2 dargestellt.

Über die Checkbox **Layout** kann eingestellt werden, ob bei der Ausführung eines Layout-Algorithmus überhaupt eine Animation erfolgen soll oder nicht. Die Funktion den Focus während der Animation vollständig auf dem Graphen zu halten, wird durch die Checkbox **zooming** aktiviert. Desweiteren gibt es noch Optionen zum Einstellen der Animationsdauer, der Anzahl der Frames, die pro Sekunde gezeichnet werden sollen, und natürlich für die Auswahl des Animations-Algorithmus, der verwendet werden soll.

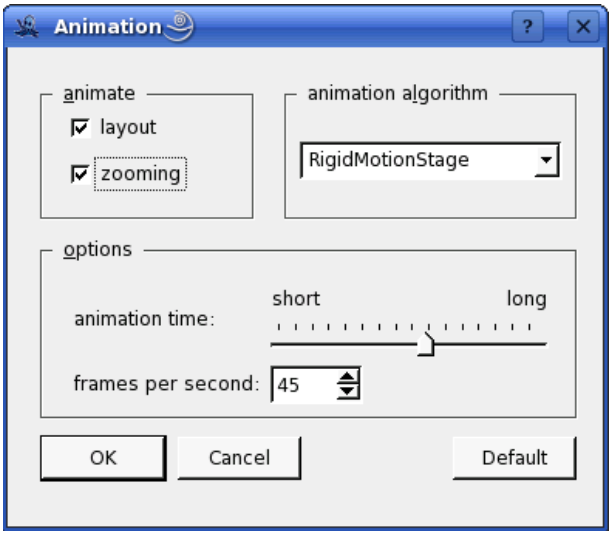

Abbildung 3.2: Fenster mit animations-spezifischen Optionen

3.4 Animation der Kantenknicke

Grundsätzlich musste natürlich auch die Animation der Kantenknicke von einem Animations-Algorithmus implementiert werden. Da diese allerdings in vielen Fällen „unabhängig“ von der Knoten-Animation betrachtet werden konnte, haben wir ein allgemeines Schema zur Animation der Kantenknicke implementiert, dass von einem Animations-Algorithmus ohne großen Implementierungsaufwand verwendet werden kann. Das Prinzip ist wie folgt:

Zunächst werden alle Kantenknicke die im Quell-Layout vorhanden sind bis zur Hälfte der Animationsdauer allmählich abgeflacht, so dass diese letztendlich verschwinden und als Kante nur noch die direkte Verbindung zwischen den betroffenen Knoten bleibt. Anders ausgedrückt werden die Winkel an den Kantenknicken, also zwischen zwei Kantensegmenten, gleichmäßig zu 180 Grad „geöffnet“. Ab dann werden bis zum Ende der Animation die „neuen“ Kantenknicke (also die Kantenknicke aus dem Ziel-Layout) auf die gleiche aber entgegengesetzte Weise animiert.

Die Vorteile dieser Art der Kantenknick-Animation sind z.B., dass die Knicke nicht losgelöst von den inzidenten Knoten animiert werden, sondern immer relativ zur Position der direkten Verbindung zwischen Start- und Zielknoten liegen (im Gegensatz z.B. bei einer linearen Animation der Kantenknicke). Dadurch wird zum Einen verhindert, dass sich die Knicke während der Animation zu weit von ihren inzidenten Knoten weg bewegen, oder umgekehrt. Zum Anderen können dadurch die Bewegungen der Kantenknicke besser nachvollzogen werden. Abgesehen davon erfolgt durch dieses Animationsschema eine klare Trennung zwischen Kantenknicken im Quell-Layout und Kantenknicken im Ziel-Layout, was insbesondere bei großen Unterschieden in der Anzahl der Kantenknicke im Quell- und Ziellayout die Üebersichlichkeit erhöht.

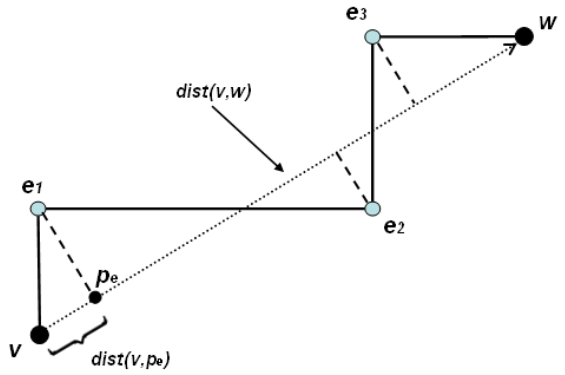

Abbildung 3.3: Skizze zur Veranschaulichung der Kantenknick-Animation

3.4.1 Konkrete Umsetzung

Die konkrete Umsetzung wird im Folgenden unter Zuhilfenahme der Skizze in Abbildung 3.3 für die Kantenknicke im Quell-Layout beschrieben (die Umsetzung für die Kantenknicke im Ziellayout ist absolut analog):

Vor Beginn der Animation werden zunächst für alle Kanten drei Werte für jeden vorhandenen Kantenknick berechnet und gespeichert, sowohl für das Quell-Layout als auch das Ziel-Layout mitels der Methode *initEdgeBendInfo()*. Diese sind:

- Der (orthogonale) Abstand $d(e, p_e)$ des Kantenknicks zur Geraden, die die Straightline zwischen inzidentem Start- und Zielknoten darstellt
- Das Längenverhältnis zwischen der Straightline und der Strecke zwischen Startknoten und Lotpunkt des Kantenknicks auf die Straightline (Abbildung 3.3 ($dist(v, p_e)$/ $dist(v, w)$))
- Die Information, ob der Kantenknick „rechts“ oder „links“ des Richtungs-Vektors zwischen Start- und Zielknoten liegt

Die Methode *computeEdgeBends()* berechnet nun für alle Kantenknicke der übergebenen Kante deren neue Positionen. Dazu wird anhand des initialen Abstands des Kantenknicks zur Straightline und der aktuellen Iteration der Anteil berechnet, um den der Kantenknick entlang seines Abstandes in Richtung Straightline bewegt werden muss. Anhand des oben beschriebenen initialen Längenverhältnisses und der aktuellen Knotenpositionen kann der relative, horizontale Abstand des Kantenknicks zum Startknoten beibehalten werden. Die Information, ob der Kantenknick „links“ oder „rechts“ lag, ist nötig, um zu bestimmen, in welche Richtung der Kantenknick entlang seines Abstandes zur Straightline bewegt werden muss.

Die Funktionalität für diese Kantenknick-Animation wurde im Animations-Modul abgelegt, so dass sie von jedem Animations-Algorithmus verwendet werden kann, da ja jeder Animations-Algorithmus vom Animations-Modul erbt.

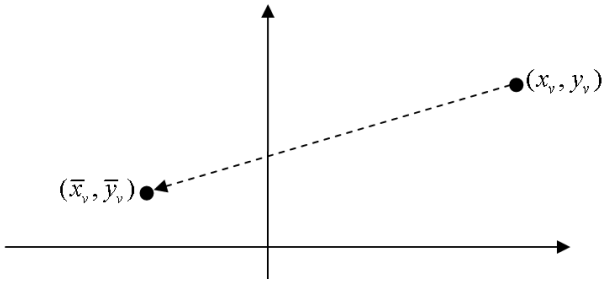

Abbildung 3.4: Kurve, entlang der ein Knoten v von der linearen Animation bewegt wird

3.5 Die Animationsalgorithmen

Im folgenden Abschnitt wird jeder implementierte Animationsalgorithmus mit einem Anwendungsbeispiel vorgestellt. Sei dabei $G = (V, E)$ der zugrundeliegende Graph. Es bezeichne $(x_v, y_v) \in \mathbb{R}^2$ stets die Koordinaten des Knotens $v \in V$ im Quell-Layout und $(\overline{x}_v, \overline{y}_v) \in \mathbb{R}^2$ die Koordinaten im Ziel-Layout. Falls die Zwischenkoordinaten der Knoten explizit beschreibbar sind, werden diese in Abhängigkeit der Zeit $t \in [0, 1]$ beschrieben. Z.B. für $t = 1/3$ gibt die Formel dann die Position des betrachteten Knotens nach 1/3 der Zeit an.

3.5.1 Die lineare Animation

Die lineare Animation ist die einfachste und sicherlich auch naheliegenste Art und Weise einer Graph-Animation. Die Koordinaten eines Knotens v lauten hier zum Zeitpunkt $t \in [0, 1]$

$$\begin{pmatrix} x_v \\ y_v \end{pmatrix} + t \begin{pmatrix} \overline{x}_v - x_v \\ \overline{y}_v - y_v \end{pmatrix}$$

Jeder Knoten wird also auf einer direkten Geraden zwischen Ausgangs- und Endpunkt bewegt. Schematisch wird diese Bewegung in Abbildung 3.4 dargestellt. Ein konkretes Beispiel für eine lineare Animation lässt sich in Abbildung 3.5 finden. Ein Nachteil dieser Animation ist, dass sich die Knoten massiv überlappen, wenn ein Großteil die Geraden, entlang derer sich die Knoten bewegen, den Mittelpunkt des Layouts kreuzen. In diesem Fall verliert der Benutzer den Überblick über die Knoten.

3.5.2 Die orthogonale Animation

Die orthogonale Animation entspricht einer modifizierten linearen Animation. Unterschied ist aber, dass Knoten zunächst entlang einer Geraden parallel zur X-Achse und dann entlang einer Geraden parallel zur Y-Achse verschoben werden, bis sie ihr Ziel erreicht haben. Wenn (x_v, y_v) erneut Ausgangspunkt und $(\overline{x}_v, \overline{y}_v)$ Endpunkt eines Knotens

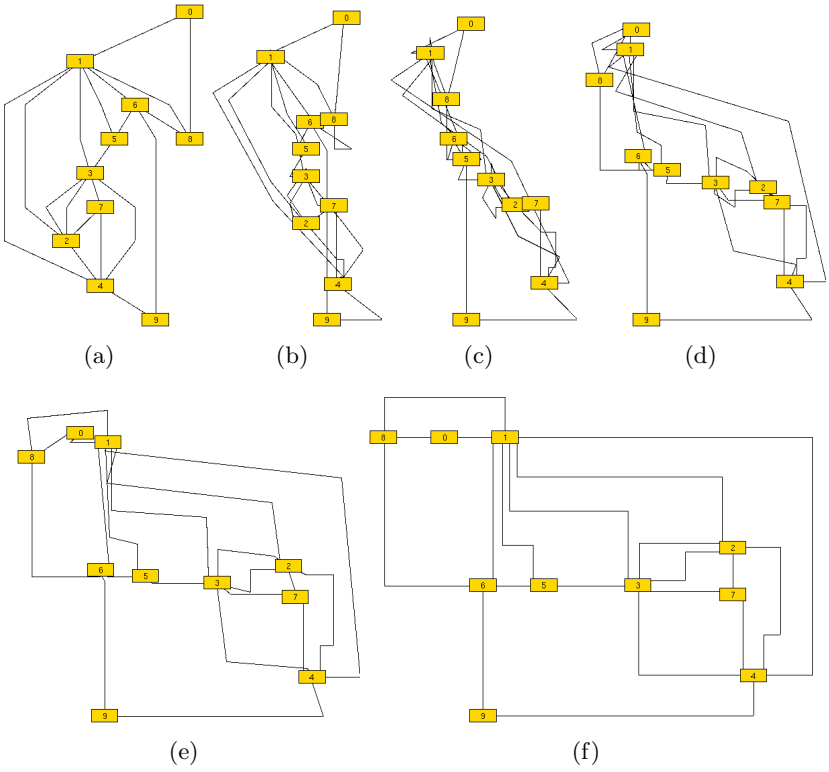

Abbildung 3.5: Anwendungsbeispiel für die lineare Animation

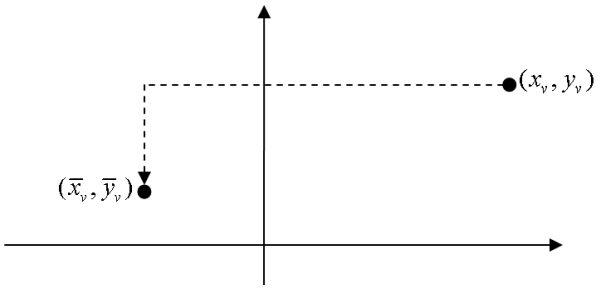

Abbildung 3.6: Kurve, entlang derer ein Knoten v von der orthogonalen Animation bewegt wird

v ist, so lauten seine Koordinaten zum Zeitpunkt $t \in [0,1]$

$$\begin{pmatrix} x_v \\ y_v \end{pmatrix} + 2t \begin{pmatrix} \overline{x}_v - x_v \\ 0 \end{pmatrix}$$

falls $t \leq \frac{1}{2}$ ist und

$$\begin{pmatrix} \overline{x}_v \\ y_v \end{pmatrix} + 2(t - \frac{1}{2}) \begin{pmatrix} 0 \\ \overline{y}_v - y_v \end{pmatrix}$$

wenn $t > \frac{1}{2}$. Eine schematische Darstellung dieser Bewegung ist in Abbildung 3.6 dargestellt. Ein konkretes Beispiel für eine orthogonale Animation findet sich in Abbildung 3.7.

3.5.3 Die Rotationsanimation

Die Idee bei dieser Animation liegt darin, alle Knoten um den Mittelpunkt des Graphen zu rotieren und sie auf diese Weise zu ihren Zielkoordinaten zu führen. Sei der Einfachheit halber der Mittelpunkt des Graphens als Urspung $(0,0)$ gewählt. Sei ϕ_v der Winkel, den Knoten v im Quell-Layout mit der X-Achse im Ursprung beschreibt und r_v die Entfernung von v zum Ursprung. Seien ferner $\overline{\phi}_v$ und $\overline{r}_v$ analog für das Ziel-Layout definiert. Dann lauten die Koordinaten von v zum Zeitpunkt $t \in [0,1]$

$$((1-t)r_v + t\overline{r}_v) \begin{pmatrix} \cos((1-t)\phi_v + t\overline{\phi}_v) \\ \sin((1-t)\phi_v + t\overline{\phi}_v) \end{pmatrix}$$

Abbildung 3.8 zeigt die Strecke, die ein Knoten zurücklegen würde. Eine konkrete Anwendung findet sich in Abbildung 3.9. Ein Vorteil dieser Art der Animation liegt darin, dass eine Überlappung von Knoten vermieden wird. Ein Nachteil besteht leider darin, dass die Knoten, die einen größeren Winkel zurücklegen eine Strecke zurücklegen, die sehr viel länger ist, als die direkte Verbindungslinie.

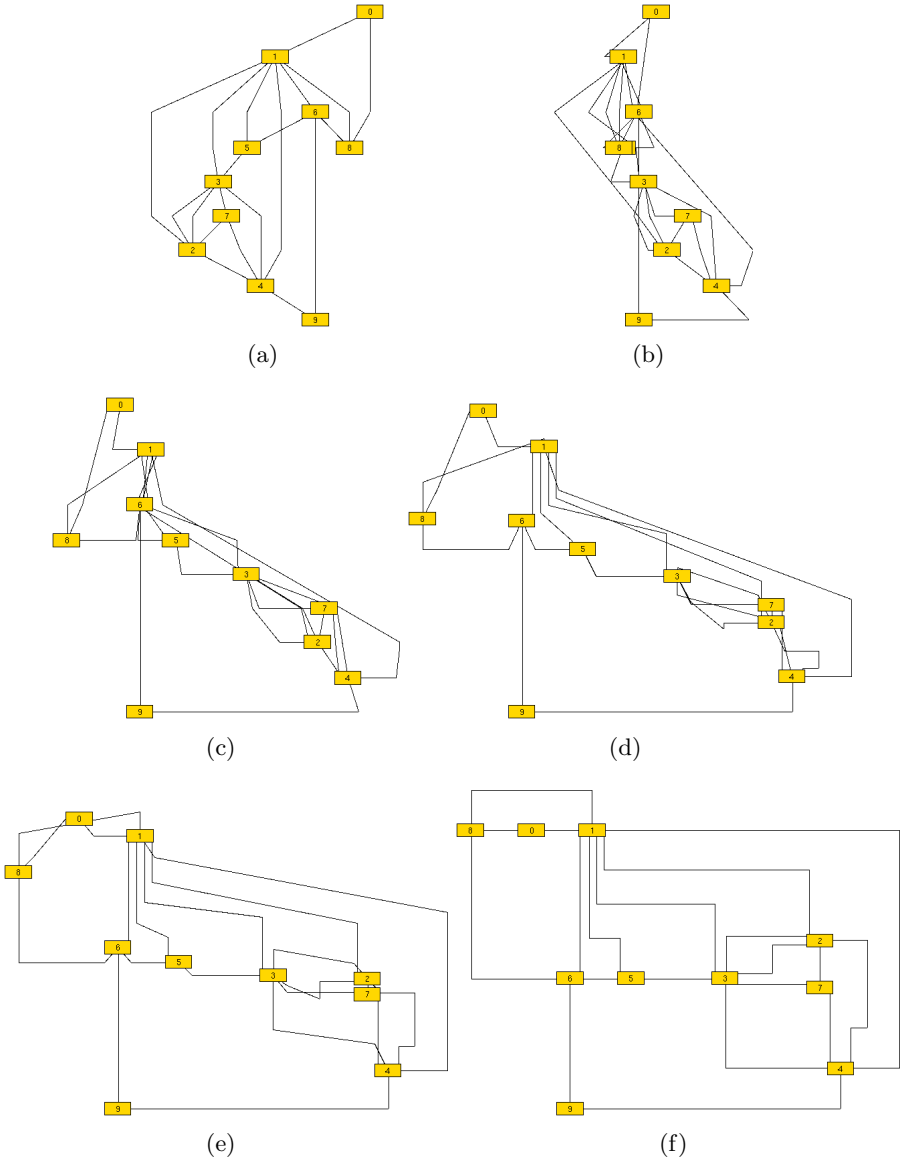

Abbildung 3.7: Anwendungsbeispiel für die orthogonale Animation

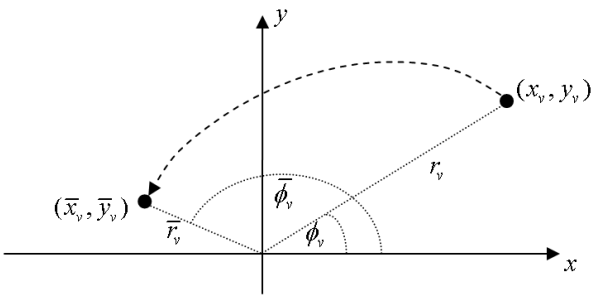

Abbildung 3.8: Kurve, entlang derer ein Knoten v von der Rotationsanimation bewegt wird

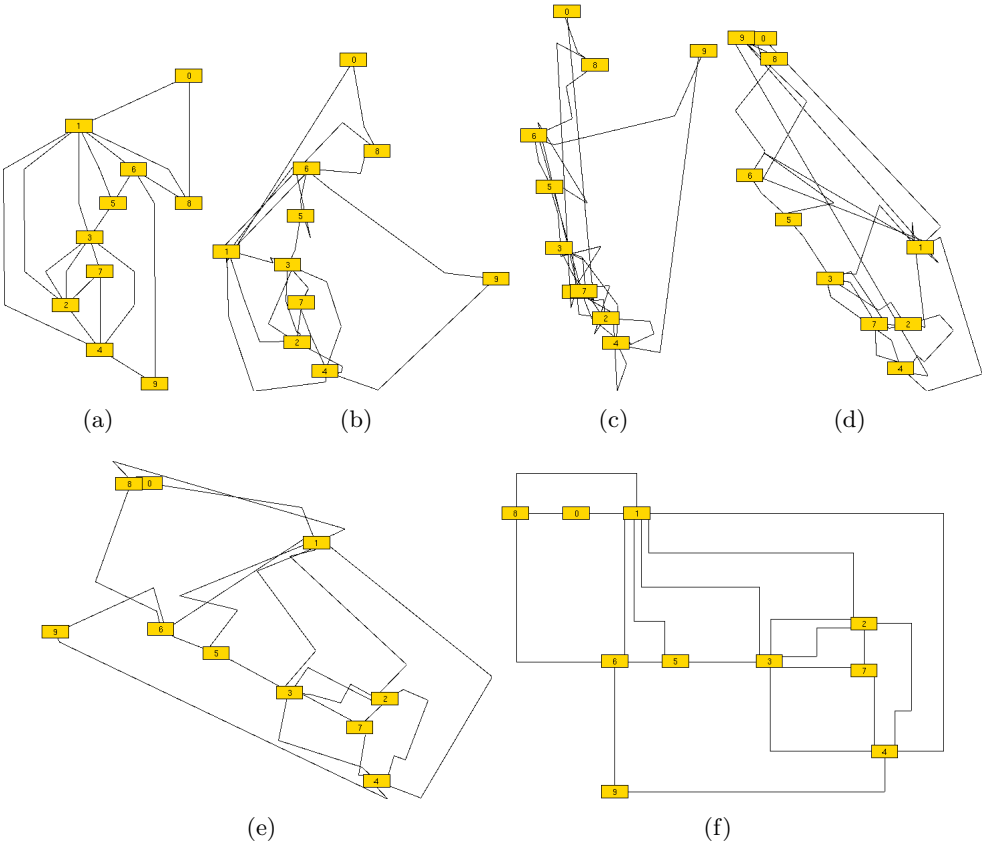

Abbildung 3.9: Anwendungsbeispiel für die Rotations-Animation. Der Algorithmus dreht das Layout in diesem Fall entgegen des Uhrzeigersinns.

3.5.4 Der Rigid-Motion-Stage

Der Rigid-Motion-Stage ist eine komplexere Methode der Animation, die auf [7] zurückgeht. Idee ist es dabei, alle Knoten gemäß einer gemeinsamen affinen Abbildung $f : \mathbb{R}^2 \to \mathbb{R}^2$ zu bewegen. Im Allgemeinen existiert eine derartige gemeinsame Abbildung, die alle Knoten an ihre exakten Zielkoordinaten bewegt, gar nicht. Daher ist es vielmehr das Ziel eine Abbildung zu finden, die alle Knoten möglichst nah an ihre Zielkoordinaten heranführt. Formal gesehen, soll eine Funktion der Form

$$f(x_v, y_v) = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x_v \\ y_v \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

gefunden werden, so dass

$$\sum_{v \in V} \| f(x_v, y_v) - (\overline{x}_v, \overline{y}_v) \|_2^2$$

minimiert wird.

Friedrich und Eades geben in [7] diejenigen Koeffizienten a_{ij}, b_i an, die die obige Zielfunktion minimieren. Die Abbildung f kann also leicht bestimmt werden.

Nun könnte man die Knoten linear von (x_v, y_v) nach $f(x_v, y_v)$ bewegen, doch dann sehen die Resultate in vielen Fällen nicht akzeptabel aus, da analog zur linearen Animation das Problem besteht, dass sich evtl. viele Knoten in der Mitte des Layouts überschneiden. Friedrich und Eades schlagen daher vor, die zu f gehörende Matrix wie folgt zu zerlegen

$$A := \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = Q \cdot S$$

wobei Q eine 2×2-Rotationsmatrix ist und S eine 2×2-Matrix ist, die die Skalierungsanteile der Bewegung enthält. Als Rotationsmatrix ist Q von der Form

$$Q = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}$$

Um den zunächst den Rotationsanteil Q zu berechnen, wird die in [17] angebebene Formel verwendet

$$Q = A + \begin{pmatrix} a_{22} & -a_{21} \\ -a_{12} & a_{11} \end{pmatrix}$$

Also kann auch ϕ bestimmt werden. Der Skalierungsanteil kann durch $S = Q^{-1}A$ ebenfalls berechnet werden. Nun lauten die Koordinaten des Knotens v zum Zeitpunkt $t \in [0, 1]$

$$\begin{pmatrix} \cos t\phi & -\sin t\phi \\ \sin t\phi & \cos t\phi \end{pmatrix} \cdot \left((1-t) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_v \\ y_v \end{pmatrix} + t \left(S \begin{pmatrix} x_v \\ y_v \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right) \right)$$

Nach Ablauf der Animation sind im Allgemeinen die Knoten nur in der Nähe ihrer Zielkoordinaten. Daher werden die Knoten dann noch per linearer Animation an ihre endgültigen Positionen bewegt. Eine Anwendung dieser Animation kann in Abbildung 3.10 gefunden werden.

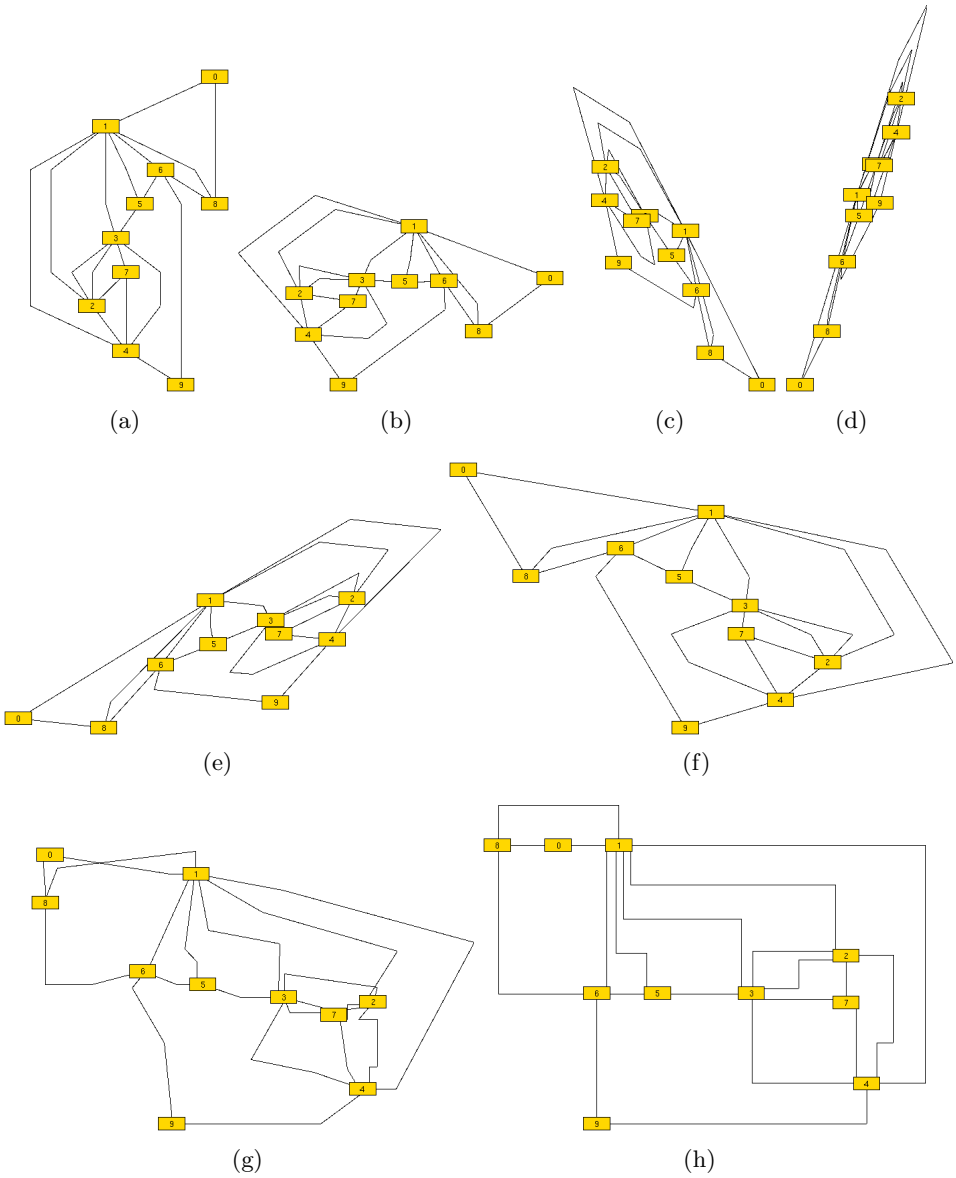

Abbildung 3.10: Anwendungsbeispiel für die Rigid-Motion-Stage-Animation. Von Bild (a) bis (f) wird eine gemeinsame affine Abbildung auf alle Knoten angewendet, deren maßgebender Bestandteil hier in einer Rotation im Uhrzeigersinn besteht. Von Bild (f) bis (h) wird schließlich eine lineare Animation durchgeführt, die alle Knoten zu ihren Endpositionen bewegt.

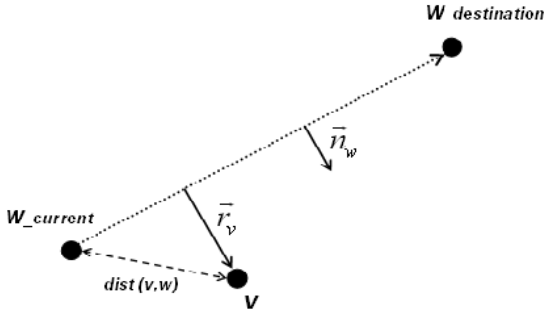

Abbildung 3.11: Schematische Skizze zur orthogonalen Ablenkung eines Knotens v von der Animations-Bahn eines Knotens w

3.5.5 Die Force-Directed Animation

Die von uns implementierte Force-Directed Animation könnte man auch als „advanced, linear Animation“ bezeichnen. Sie ziehlt in erster Linie darauf ab, die Knoten möglichst, wie bei der linearen Animation, entlang der geraden Verbindungslinie zwischen Start- und Zielposition des Knotens zu bewegen. Da der große Nachteil der linearen Animation allerdings in den dadurch entstehenden Knotenüberlappungen lag, versucht die Force-Directed Animation genau diese möglichst zu vermeiden.

Dazu wird in jeder Iteration für jeden Knoten v sein momentaner Abstand zu allen anderen Knoten w ermittelt. Unterschreitet die Distanz $dist(v, w)$ für einen Knoten w dabei einen bestimmten Wert, so liegt die Wahrscheinlichkeit nahe, dass sich die „Animations-Bahnen“ der beiden Knoten überlappen werden, die beiden Knoten sich also zu nahe kommen. Um dies zu vermeiden, wird der Knoten v von der Animations-Bahn des Knotens w orthogonal „abgelenkt“ (in Abbildung 3.11 ist eine schematische Skizze dazu abgebildet). Dazu wird in Abhängigkeit des Abstands eine Kraft berechnet, die die Stärke der orthogonalen Ablenkung bestimmt. Die Kraft ist natürlich umso größer, je kleiner der Anstand ist. Konkret ist dies wie folgt umgesetzt: Sei $f : \mathbb{R} \to \mathbb{R}$ die Funktion, die in Abhängigkeit des Abstands die abstoßende Kraft berechnet, $dist(v, w)$ der aktuelle Abstand zwischen v und w, $\vec{w}$ der Richtungsvektor zwischen aktueller und Zielposition des Knotens w und $\vec{n_w}$ ein normalisierter, zu $\vec{w}$ orthogonaler Vektor. Dann wird der Abstoßungs-Vektor $\vec{r}$ ermittelt durch $\vec{r} = \vec{n_w} * f(dist(v, w))$. Da dies für jeden Knoten w berechnet wird, ergibt sich die endgültige Ablenkung $\vec{r_v}$ für den Knoten v als:

$$\vec{r_v} = \sum_{w \neq v} f(dist(v, w)) * \vec{n_w}$$

Durch eine solche Ablenkung verändert sich natürlich auch die geradlinige Animations-Bahn des Knotens. Diese muss also nach jeder Iteration neu berechnet werden.

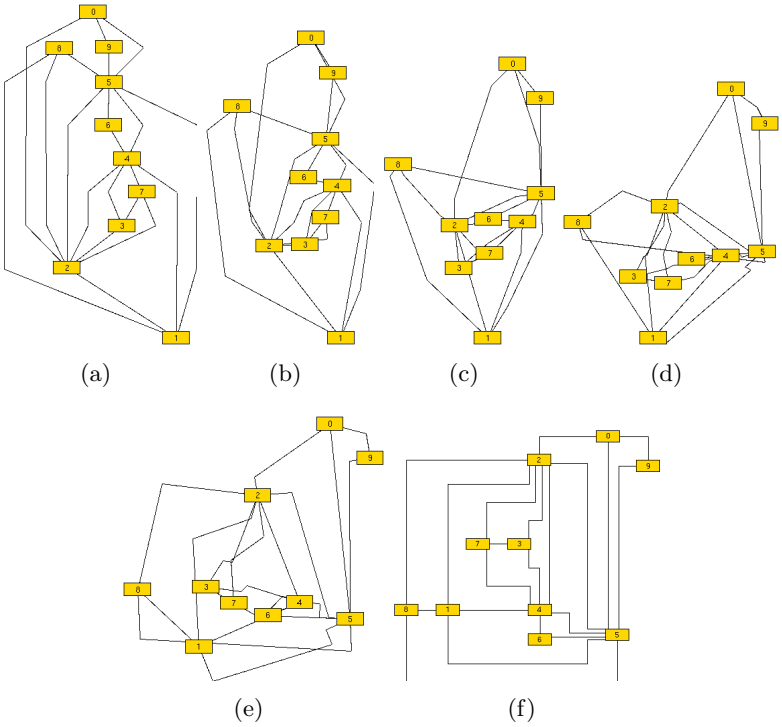

Abbildung 3.12: Anwendungsbeispiel für die Force-Directed-Animation.

Die Vorteile einer solchen Animation sind offensichtlich. Allerdings kann diese Methode zur Vermeidung von Knotenüberlappungen natürlich nur als heuristischer Ansatz betrachtet werden. Sie bietet keine Garantie, dass es niemals zu Überlappungen kommt. Insbesondere ist gerade die Wahl der Abstoßungs-Funktion eine sensible Stelle dieses Algorithmus. Es kann dazu kommen, dass sich Knoten gegenseitig blockieren, und sich dadurch zu weit von ihrer Zielposition wegbewegen, was die letztendlich zurückzulegende Strecke des Knotens unnötig erhöht. Auch die Animations-Geschwindigkeit eines Knotens kann sich dadurch gerade zum Ende der Animation hin erheblich erhöhen, was den hauptsächlichen Nachteil dieser Animation ausmacht. Positiv auswirken könnte sich vielleicht die Hinzunahme von randomisieten Elementen, um solche Blockierungen zu minimieren. Eine Anwendung dieser Animation kann in Abbildung 3.12 gefunden werden.

3.5.6 Die gradabhängige, lineare Animation

Hierbei handelt es sich um eine Adaption der gewöhnlichen linearen Animation. Die Motivation zu dieser Animation lag darin, dass die Tatsache, dass ein Knoten mit hohem Grad ein für den Graphen zentraler, „wichtiger“ Knoten ist durchaus auf viele Graphen zutrifft. Die Annahme war daher, dass es sinnvoll sein könnte, zunächst die Knoten mit hohem Grad zu animieren, um deren Bewegung in Ruhe nachvollziehen zu können. Es ergibt sich somit eine Animation, bei der die Knoten gruppenweise bzw. etappenweise animiert werden.

Konkret geht der Algorithmus so vor, dass er zunächst die Differenz zwischen maximalem und minimalem Knotengrad bestimmt und anhand dessen die Anzahl der Gruppen festlegt (maximal vier), und anschließend die Knoten abhängig von ihrem Grad in diese Gruppen einteilt. Die Einteilung erfolgt linear zur Anzahl der unterschiedlichen Knotengrade in absteigender, konsekutiver Folge.

Beispiel: Seien max und min die maximal bzw. minimal auftretenden Knotengrade und sei n die Differenz zwischen maximalem und minimalem Knotengrad. Es gebe p Gruppen und p sei Teiler von n. Die erste Gruppe erhält nun diejenigen Knoten, deren Grad im Bereich $[max; max - (\frac{n}{p} - 1)]$ liegt. Die zweite Gruppe enthält die Knoten, deren Grad im Bereich $[max - \frac{n}{p}; max - (2 * \frac{n}{p} - 1)]$ liegt, usw.

Denkbar gewesen wäre natürlich auch eine Einteilung, die nicht linear vorgeht, sondern bei der die Intervallgrößen der Gruppen abnehmen. Unter der Annahme, dass der gegebene Graph nur weinige Knoten mit hohem Grad und viele Knoten mit nierdrigerem Grad hat, würden dadurch meist Gruppen entstehen, die alle ungefähr gleich viele Knoten enthalten. Die gradabhängige, lineare Animation könnte in manchen Fällen bzw. Bereichen des automatischen Graphenzeichnens durchaus Anwendung finden.

3.6 Ausblick

Erweiterungsmöglichkeiten könnten z.B. darin bestehen, dass für die Animation von Compound-Graphen bisher gewöhnliche Animationsverfahren verwendet werden, die

nicht die gegebene hierarchische Struktur der Graphen berücksichtigen. Diesem Missstand könnte man mit speziellen Compound-Animationen abhelfen. Dabei wäre z.B. ein Algorithmus denkbar, der zunächst alle Compounds ausblendet, die nicht direkt unter der Wurzel hängen, dann die verbleibenden Compounds an ihre Zielpositionen verschiebt. Danach werden dann die Compounds auf der nächsten Ebene eingeblendet, welche nun animiert werden usw. Eine beispielhafte Darstellung dieses Verfahrens ist in Abbildung 3.13 zu finden.

GDE-seitig wäre auch noch eine Funktion wünschenswert, die es erlaubt die Animation eines Graphens selbst abzuspeichern, beispielsweise in einer Video-Datei.

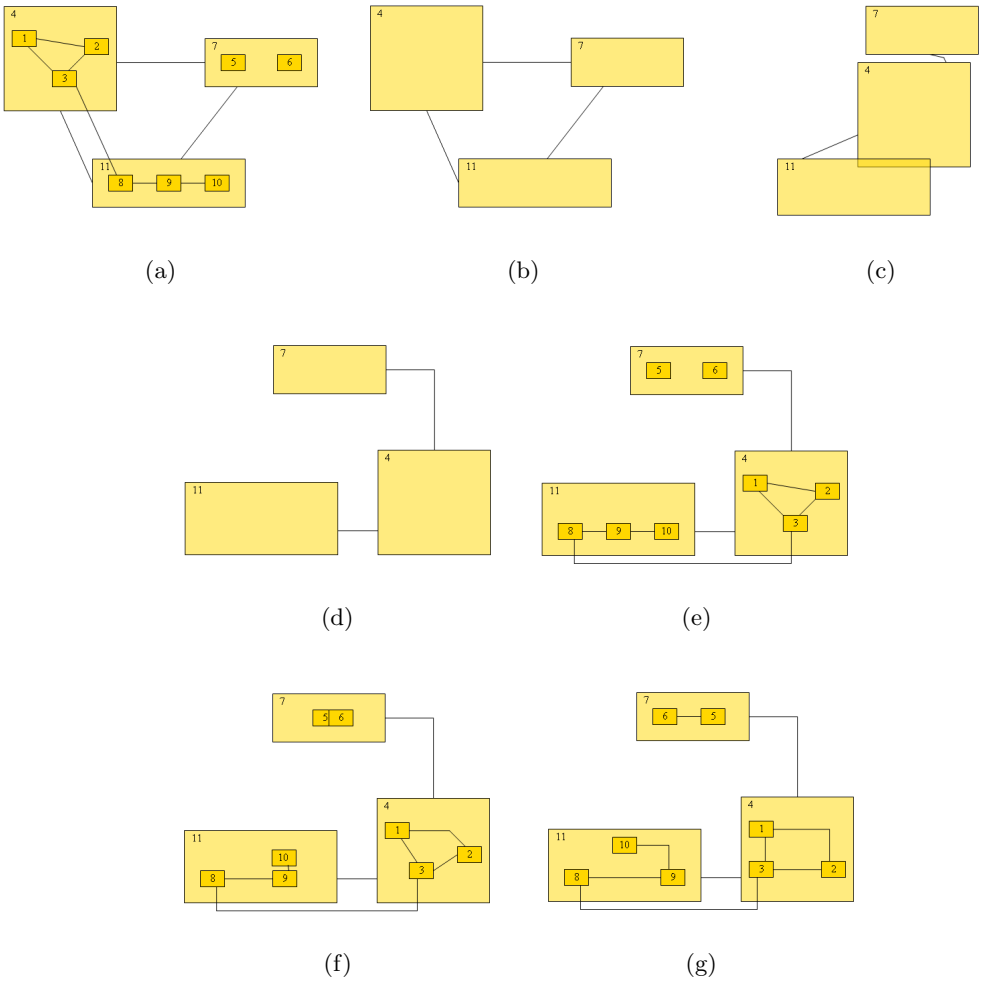

Abbildung 3.13: Darstellung einer möglichen Compound-Animation, die die hierarchische Struktur des Graphens berücksichtigt

4 Arbeitsgruppe Compounds

Man ist es gewohnt, mit Hilfe von Graphen Abhängigkeiten zwischen einzelnen Komponenten sehr anschaulich visualisieren zu können. Oft kommt es auch vor, dass einige dieser Komponenten zu Gruppen zusammengefasst werden können und wenn zusätzliche Abhängigkeiten zwischen diesen Gruppen untereinander oder auch mit anderen (einzelnen) Komponenten existieren, dann lassen sich diese Gruppen als „speziellen Knoten“ in einen Graphen integrieren. Diese „speziellen Knoten“ nennt man „Compounds“.
Zuerst werden im Folgenden genauer die Definition von Compounds und das daraus erarbeitete Konzept vorgestellt, um die durch Compounds induzierte Inklusionsstruktur verwalten zu können. Anschließend werden die entworfenen Layout-Verfahren für „CompoundGraphen“ erläutert.
Diese Tätigkeiten stellen die im ersten Semester absolvierten Aufgaben dar. Im zweiten Semester hingegen lag der Schwerpunkt auf der Realisierung einer graphischen Darstellung. Zu diesem Zweck wurde das zur Algorithmenbibliothek OGDF bestehende graphische Tool „GoVisual Diagramm Editor“ von uns überarbeitet, um Compound-Graphen darstellen zu können, worauf im letzten Kapitel eingegangen wird.

4.1 Einleitung

Unsere Aufgabe ist es, eine entsprechende Repräsentation im OGDF zu konstruieren und zu implementieren, um Compounds verwalten und editieren zu können und vor allem auch dafür entwickelte Layout-Algorithmen zur Verfügung zu stellen. Wir definieren kurz die dazu notwendigen Begriffe.

Definition eines Compound-Graphen

Ein Compound-Graph $C = (G, T)$ ist definiert als ein (gerichteter oder ungerichteter) Graph $G = (V, E_G)$ und ein Baum $T = (V, E_T)$, wobei beide eine gemeinsame Knotenmenge V besitzen.
Durch T, den Inklusionsbaum, wird also modelliert, welche Knoten in einem anderen Knoten enthalten sind, womit man die hierarchische Komponente erhält. In Abbildung 4.1 wird gezeigt, wie ein Compound-Graph dargestellt wird. Die Struktur des Baumes stellt die Inklusionsbeziehungen zwischen den Knoten dar. Der Knoten u steht in Inklusionsbeziehung zum Knoten v genau dann, wenn u ein Kind von v in T ist. Wenn die Endknoten u und v aller Kanten $u, v \in E_G \cup E_T$ zu verschiedenen Pfaden von der Wurzel aus gehören, so nennt man C einen „Simple Compound Graph“ (Abbildung 4.2). In einem solchen Graphen kann ein Knotenpaar (u, v) niemals gleichzeitig in Adjazenz- und Inklusionsbeziehung stehen.

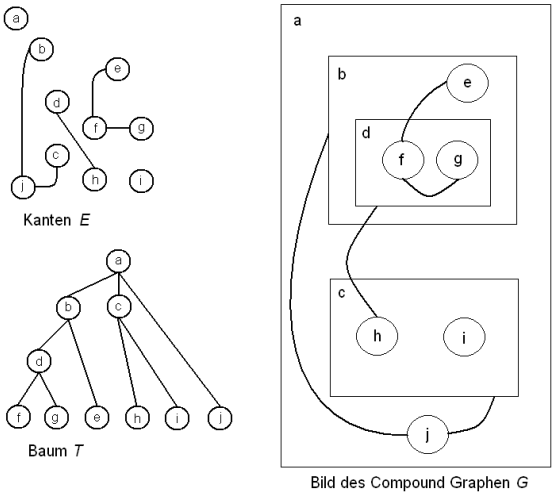

Abbildung 4.1: Graph, Inklusionsbaum, Compound-Graph

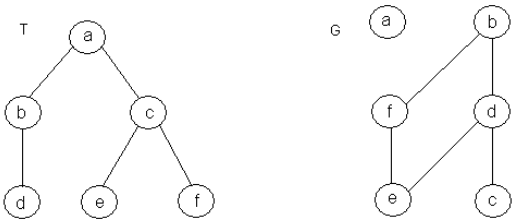

Abbildung 4.2: Simple Compound-Graph

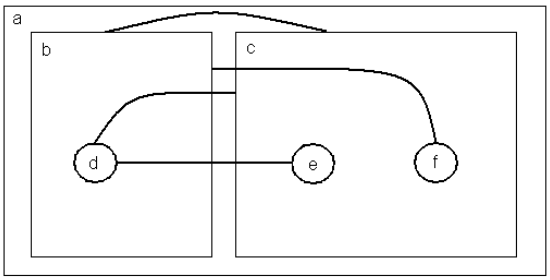

Abbildung 4.3: Zeichnen eines Compound-Graphen

Kanten zwischen Knoten auf verschiedenen Baumebenen heißen „Inter-Level Edges“. Ein Compound-Graph, welcher keine solche Kanten enthält, heißt „Nested Graph“. Eine weitere Einschränkung, welche ausschließlich Kanten zwischen den Blättern von T erlaubt, führt zu den sogenannten „Cluster-Graphen“ (welche bereits im OGDF implementiert sind).

Zeichnen eines Compound-Graphen

Beim Zeichnen eines Compound-Graphen $C = (G, T)$ werden die Knoten des Graphen G als geschlossene Regionen gezeichnet (Abbildung 4.3). Liegt ein Knoten v in einer Region, die den Knoten u repräsentiert, so ist v ein Kind von u in T. Die Kanten $e \in E_G$ werden als Kanten gezeichnet, welche die Regionen verbinden, die ihre Endpunkte darstellen.

4.2 Klassen `CompoundGraph` und `CompoundElement`

4.2.1 Modellierung

Da der Compound-Graph eine Erweiterung des Cluster-Graphen darstellt, entschlossen wir uns, als Analogon zu den Klassen `ClusterGraph` und `ClusterElement` die Klassen `CompoundGraph` (für die Modellierung des gesamten Graphen) und `CompoundElement` (um einen einzelnen Knoten des Graphens darzustellen) als Kern unseres Teilprojektes zu entwickeln. Eine andere Datenstruktur für das Verwalten von Compound-Graphen wäre wenig sinnvoll gewesen. Wir sahen es als zweckmäßig an, weitgehend dieselben Methoden für den `CompoundGraph` umzusetzen, wie für den `ClusterGraph` bereits zur Verfügung standen, allerdings zeigte sich schnell, dass die Struktur „unter der Oberfläche“ anders beschaffen werden musste. Beim `ClusterGraph` sind die Blätter bzgl. der Inklusionsstruktur als `NodeElements` modelliert, während die inneren Knoten als `ClusterElements` verwaltet werden. Diese getrennte Behandlung ist sinnvoll, da die inneren Knoten per Definition keine Kanten erhalten können. Beim `CompoundGraph` fällt aber diese Trennung zwischen Blättern und inneren Knoten weg, denn Kanten können sowohl von Blättern als auch von inneren Knoten ausgehen. Daher entschieden wir uns für alle „Sorten“ von Knoten stets `CompoundElements` einzusetzen. Lediglich ein Flag

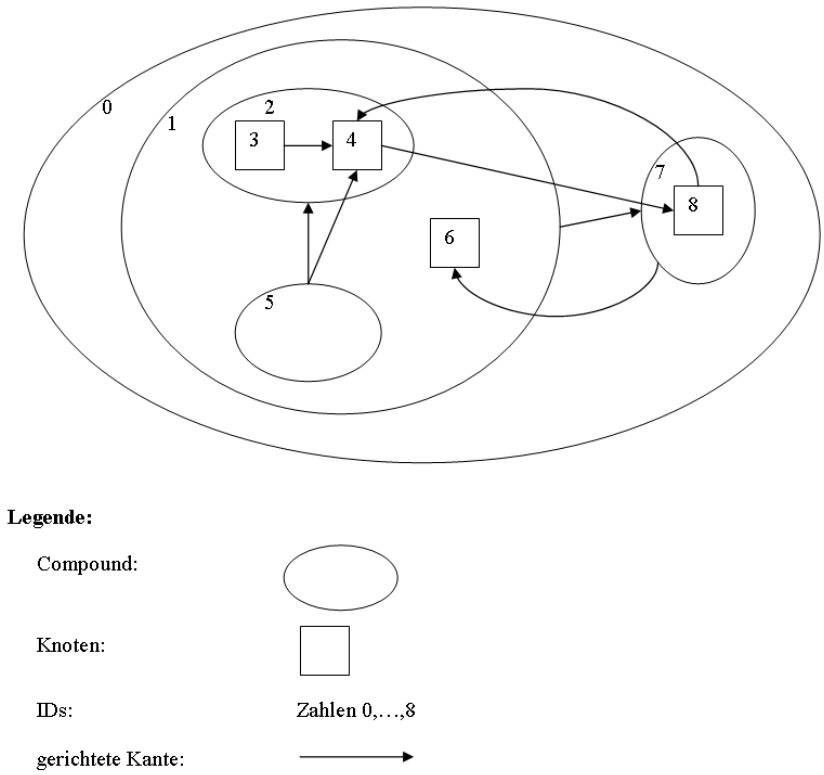

Abbildung 4.4: Beispiel eines Compound-Graphen. Der Wurzelcompound hat die ID 0, die Objekte mit den IDs 3,4,6,8 werden als Knoten dargestellt. Intern werden sie jedoch ebenfalls als Objekte vom Typ `CompoundElement` verwaltet.

kann ggf. gesetzt werden, mit dem der Benutzer für einen Compound angeben kann, ob dieser in der graphischen Darstellung als Knoten oder als Compound dargestellt werden soll. Dieses Feature wurde insbesondere im Hinblick auf den Einsatz in GDE eingebaut. Ein Beispiel eines Compound-Graphen findet sich in Abbildung 4.4.

Da die Behandlung von Kanten bereits in der Klasse `Graph` implementiert war, waren wir uns von Anfang an einig, diese für die Verwaltung der Kanten im `CompoundGraph` zu verwenden, um unnütze Doppelimplementierungen zu vermeiden. Zunächst planten wir die Klasse `CompoundElement` von `NodeElement` erben zu lassen, so dass dann `CompoundElements` anstatt `NodeElements` in den Graph eingefügt werden. Auf Grund von Komplikationen, die dieser Eingriff in die komplexeren Mechanismen der Klasse `Graph` wohl nach sich gezogen hätte, änderten wir nach einer Beratung mit den Betreuern unsere Pläne. Nun sahen wir es vor, die Klasse `CompoundElement` aus der direkten Verwaltung der Kanten herauszunehmen und stattdessen jedem Compound bijektiv ein `NodeElement` zuzuweisen. Auf diese Weise haben wir gemäß nach Definition einen Graphen G der Klasse `Graph`, der die Beziehungen der Knoten untereinander über die Kanten

festlegt und einen Inklusionsbaum T, in unserer Implementierung der `CompoundGraph`, der die Hierachie der Knoten festhält. Abbildung 4.5 zeigt die interne Struktur des Beispiels aus Abbildung 4.4.

Es fällt auf, dass dem Wurzel-Compound kein `NodeElement` zugewiesen ist. Dies wäre auch nicht notwendig, da in einem Compound-Graphen per Definition keine Kante zum Wurzel-Compound inzident sein kann (da der Wurzel-Compound Vorfahre jedes anderen Knotens ist). Der Hauptgrund, warum dieses Detail praktisch in letzter Minute geändert wurde, ist der Folgende: Im Falle, dass alle Compounds direkt unter der Wurzel angeordnet sind (also abgesehen vom Wurzel-Compound ein „normaler“ Graph vorliegt, (siehe Abbildung 4.6), können sehr einfach Layout-Algorithmen, die für einen gewöhnlichen Graphen konzipiert sind, auf den Compound-Graphen angewendet werden. Wenn ein zusätzliches `NodeElement` für die Wurzel vorhanden gewesen wäre, würde dieses die Layouts unnötig stören.

4.2.2 Umfang der Methoden

Der Umfang der Methoden umfasst bisher unter anderm die Aufgabenbereiche

- Abfragen von Kanten
- Änderungen der Inklusionsstruktur (z.B. „Umhängen“ von Compounds)
- Anfragen zur Inklusionsstruktur (ist ein Compound Vorfahr von einem anderen Compound, wie viele Kinder hat ein Compound, weches ist der früheste gemeinsame Vorfahr, etc.)
- Berechnung von Tiefe und Rang[1] von Compounds bzgl. der Inklusionsstruktur
- Erstellen einer tiefen Kopie des Compound-Graphen
- Umwandeln in einen Cluster-Graphen und zurück
- Debug-Methoden, um den Compound-Graphen ausgeben zu lassen bzw. seine Konsistenz zu überprüfen

4.3 Layout-Algorithmen für Compound-Graphen

4.3.1 Vorausgehende Entscheidungsfindung

Die nächste Aufgabe bestand darin, mindestens einen Layout-Algorithmus für Compound-Graphen umzusetzen. Zur Auswahl standen ein vergleichsweise einfacher Force-Directed-Ansatz nach [2] und das deutlich komplexere Verfahren von Sugiyama. Wir entschlossen uns den Force-Directed Ansatz umzusetzen.

[1] Der Rang eine Knotens ist hier definiert als Länge des Weges vom Knoten zur Wurzel.

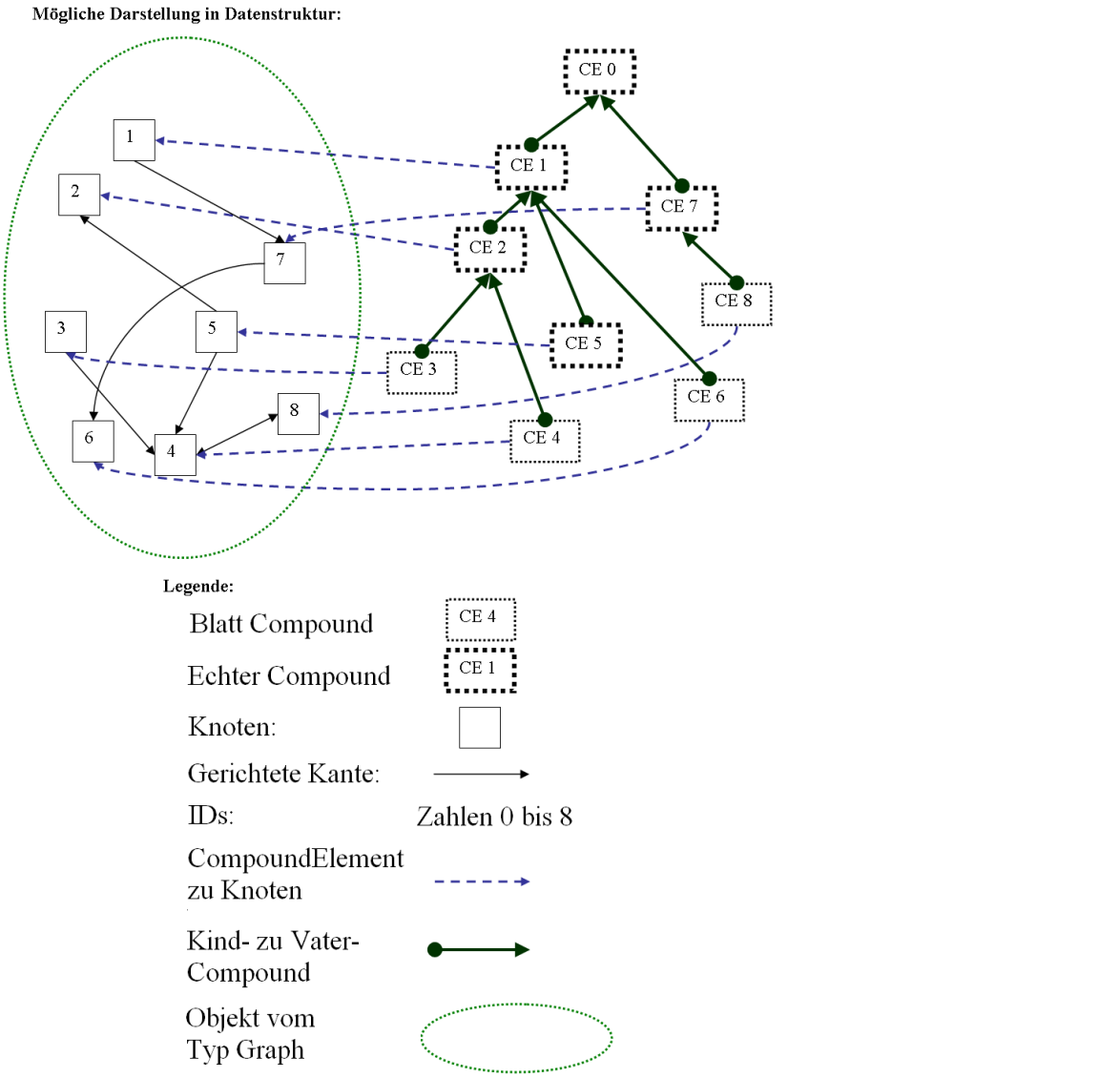

Abbildung 4.5: Darstellung der internen Datenstrukturen zum Beispiel aus Abbildung 4.4

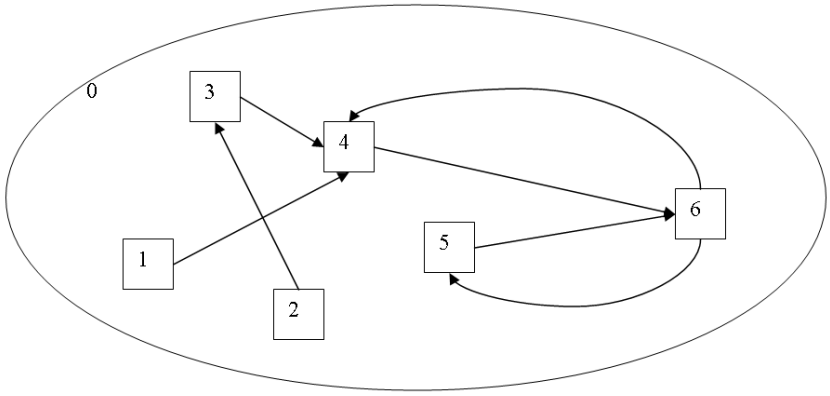

Abbildung 4.6: Compound-Graph ohne echte Inklusionsstruktur, auf den auch gewöhnliche Graph-Layouts angewendet werden können.

4.3.2 Der Force-Directed-Ansatz

Die Idee des Force-Directed-Ansatzes liegt darin, dass beginnend von der untersten Ebene (also der Ebene der Compounds, die keine Nachfolger haben) für jeden Compound einmal ein Force-Directed-Algorithmus für alle seine Kinder aufgerufen wird (wobei deren Kinder allerdings ignoriert werden). Als großer Vorteil dieses Verfahrens stellte sich heraus, dass zunächst der bereits implementierte Algorithmus von Fruchtermann und Reingold als Blackbox verwendet werden konnte. Bis zum jetzigen Zeitpunkt wird leider noch keine zusätzliche, nachträgliche Optimierung angewendet, um das Layout weiter zu verbessern und die existierenden, gravierenden Nachteile auszubessern.

Der Algorithmus ruft die Methode MainStep (hier in Pseudo-Code) für den Wurzel-Compound auf:

```
PROC MainStep (Compound C)
   1. IF C hat keine Kinder THEN RETURN
   2. FOR EACH Kind C' von C DO
        CALL Mainstep (C')
      END FOR
   3. erstelle Teilgraph G' mit Kindern von C als Knoten und Kanten
        zwischen den Compounds, deren Nachfolger adjazent sind
   4. setze Knotengrößen in G' auf berechnete Compoundgrößen
   5. initialisiere Positionen der Knoten in G' mit zufälligen Werten
   6. rufe Force-Directed Algo für G' auf
   7. übertrage Koordinaten von G' auf Kinder von C
   8. setze Größe von C auf Maße der Boundingbox von G'
END
```

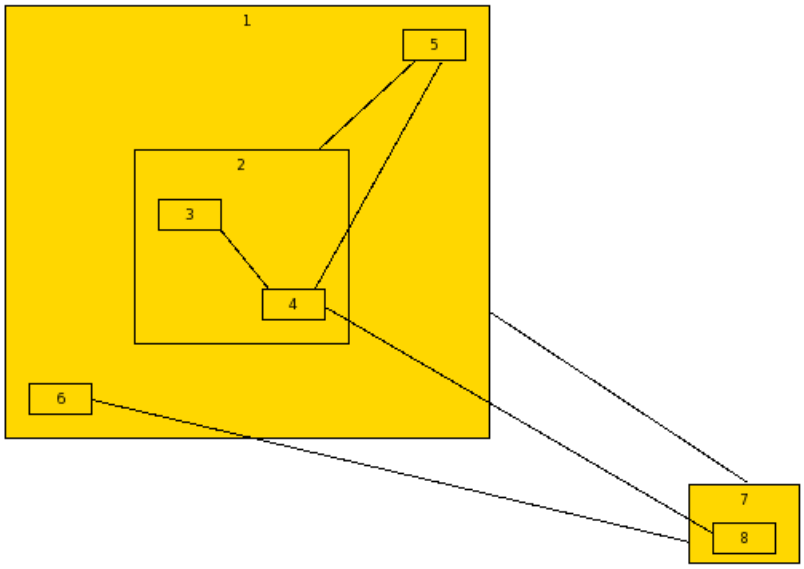

Abbildung 4.7: Von der Klasse `SimpleCompoundLayout` erzeugtes Layout des Beispielgraphs aus Abbildung 4.4

4.3.3 Layout-Beispiele

Um dem Leser ein besseres Gefühl für die produzierten Layouts inklusive Vor- und Nachteilen zu geben, zeigen wir nun zwei Beispiele.

Für den aus Kapitel 4.2.1 bekannten Compound-Graphen (siehe Abbildung 4.4) wird beispielsweise das Layout, wie in Abbildung 4.7 zu sehen ist, erzeugt.

Auf den ersten Blick ein passables Ergebnis, aber das Beispiel zeigt nicht die Probleme auf, die der Algorithmus zum jetzigen Zeitpunkt noch aufweist. Betrachten wir daher den zufällig erzeugten Graphen mit 10 Compounds und 10 Kanten in Abbildung 4.8.

Da der Force-Directed-Algorithmus Kantenkreuzungen nicht verhindert, überrascht es auch nicht, dass unnötige Kantenkreuzungen wie zwischen Kanten $(4, 8)$ und $(5, 6)$ auftreten, die beseitigt werden könnten, wenn man Compound 6 weiter nach unten zieht. Ein weiterer überaus unschöner Punkt ist, dass Kanten quer durch Compounds verlaufen können, so wie es bei den Compounds 4 und 6 der Fall ist. Auch dies ließe sich mit einer geänderten Positionierung verhindern.

4.3.4 Optimierung des Force-Directed-Ansatz

Eine Möglichkeit, um die soeben beschriebenen Probleme (zumindest teilweise) zu beseitigen, ist es, das erhaltene Layout durch lokale Veränderungen wie z.B. Vertauschen von Knoten und Verändern der Position von Knoten schrittweise zu verbessern. Unsere Idee war es nun, diese lokalen Veränderungen im Rahmen einer randomisierten lokalen Suchheuristik umzusetzen. Bekanntlich haben derartige Heuristiken die Eigenschaft, recht

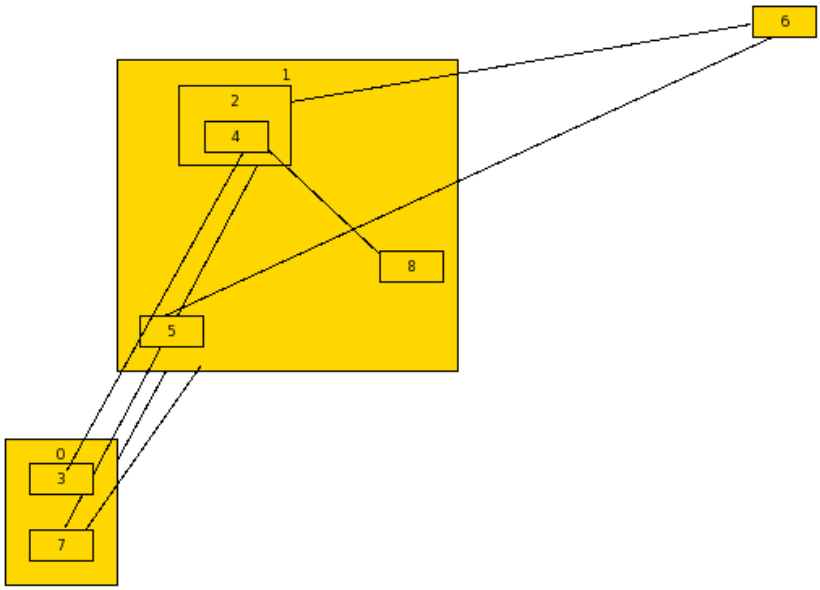

Abbildung 4.8: Beispiel für ein ungünstiges, von SimpleCompoundLayout erzeugtes Layout. Dieser Graph mit 10 Compounds und 10 Kanten wurde per Zufallsgenerator erzeugt.

zügig zumindest lokale Optima zu erreichen. Unserer Meinung nach, bietet ein solches, lokales Optimum in vielen Fällen ein ansprechendes Layout.

Die beiden Hauptbestandteile bei der Adaption einer Suchheuristik sind die Definition des Umgebungsbegriffs sowie der Zielfunktion, an Hand welcher sich entscheidet, ob ein erreichter Nachbar übernommen wird oder nicht. Die Umgebung unseres Layouts enthält alle Layouts, die erreicht werden können durch:

- Vertauschen zweier Compounds
- Setzen der Koordinaten eines Compounds auf einen zufälligen Wert
- Drehen eines Compounds (mit seinen Nachfolgern) um einen zufälligen Winkel

Die Zielfunktion enthält dabei folgende Faktoren:

- Minimieren der Kantenkreuzungen
- Minimierung der Kreuzung von Kanten und Compounds [2]
- Minimieren der Kantenlängen
- Minimieren der Winkel aller zu einem Compound adjazenten Kanten

[2] wie z.B. bei Compounds 4 und 6 in Abbildung 4.8 notwendig

- Erhalt der Inklusionsstruktur (ein Compound muss auch innerhalb seines Vater-Compound angeordnet sein)
- Verbieten des Überlappens von Compounds

Welche Anzahl von Iterationen bzw. welche Gewichte in der Zielfunktion verwendet werden, kann vom Benutzer manuell eingestellt werden. Es ist natürlich auch möglich, einen vordefinierten Satz aus Gewichten und Iterationenzahl zu verwenden. Außerdem enthält der Layout-Algorithmus nun eine Methode um Compounds und Knoten vor der Berechnung der Kreuzungszahl temporär aufzublähen, das heißt alle Compounds/Knoten werden rekursiv um einige Pixel vergrößert. Dadurch enstehen Kreuzungen natürlich bereits, obwohl die betreffende Kante sonst äußerst knapp an einem Knoten vorbeiführen würde. Genau dies ist beabsichtigt, da sehr knapp an Knoten vorbeiführende Kanten optisch sehr unschön wirken.

Nun betrachten wir beispielhaft die praktische Wirkung des randomisierten Layout-Algorithmus. Zu diesem Zweck wollen wir das suboptimale Ergebnis des SimpleLayout aus Abbildung 4.8 verbessern. Wir führen 500 Iterationen des RLS durch. In Abbildung 4.9 sind durch zufällige Vertauschungen bereits deutliche Veränderungen und Verbesserungen festzustellen. Anfänglich 9 Kreuzungen konnten wir so bereits auf 2 Kreuzungen reduzieren, einfach durch die Vertauschung der Knoten 5 und 8, sowie der Verschiebung des Compound 0. Nach etwas ber 200 weiteren Iterationen erreichen wir bereits ein lokales Optimum, in welchem es tatsächlich keinerlei Kreuzungen mehr gibt, wie in Abbildung 4.10 ersichtlich ist. Dies wurde durch einen Positionstausch der Knoten 3 und 7, sowie einer erneuten Verschiebung des Compound 0 erreicht.

4.4 GDE - GoVisual Diagramm Editor

Um dem Nutzer der Graph-Algorithmen, die im OGDF implementiert wurden, eine Möglichkeit zu bieten entsprechende Ergebnisse graphisch präsentiert zu bekommen, existiert als Parallel-Projekt der „GoVisual Diagram Editor“ - kurz: GDE.

Dieser bietet nicht nur die Möglichkeit einen Graphen zu betrachten, sondern auch diverse Werkzeuge, um beliebige Graphen selbst erzeugen zu können. Außerdem können ausgewählte Algorithmen der OGDF Bibliothek auf jenen angewendet und das dementsprechende Ergebnis betrachtet werden.

Die Oberfläche des GDE v1.3 bietet drei Bereiche:

- das Popup-Menü und optionale Werkzeugleisten
- das Hauptfenster, welches den Graphen abbildet
- den Cluster-Explorer (optional), der den aktuellen Cluster-Baum darstellt

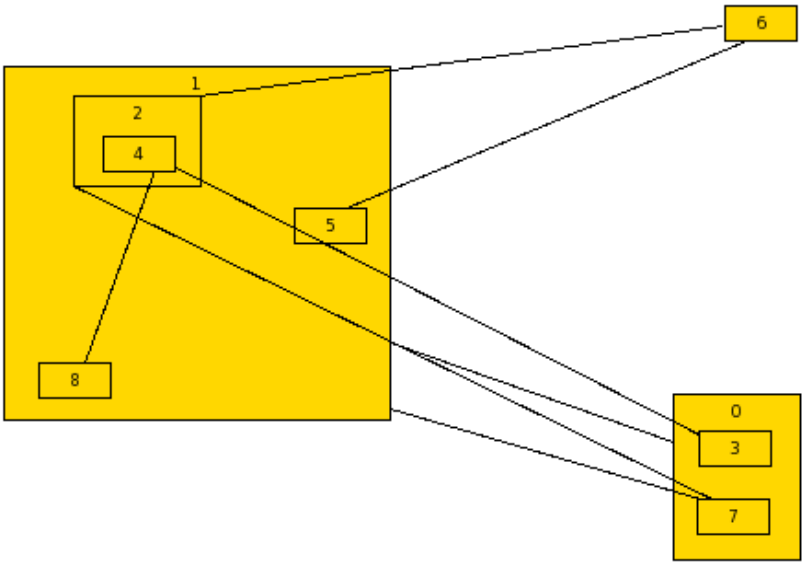

Abbildung 4.9: Von `RLSLayout` erzeugtes Layout des Beispielgraphs aus Abbildung 4.8 nach 41 Iterationen

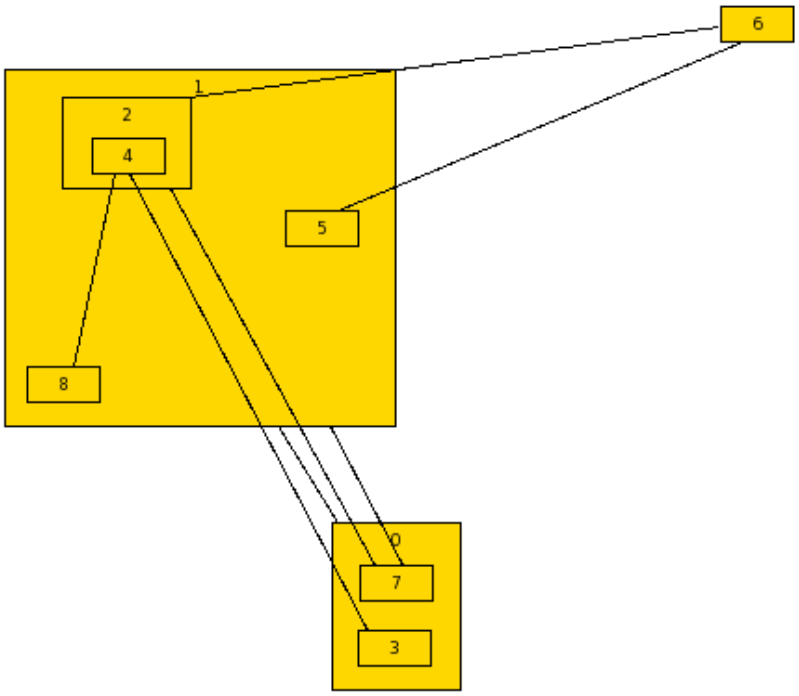

Abbildung 4.10: Von `RLSLayout` erzeugtes Layout des Beispielgraphs aus Abbildung 4.8 nach 274 Iterationen

Die Oberfläche stellt dem Benutzer folgende Möglichkeiten zur Verfügung:

- Laden und Speichern von (Cluster-) Graphen
- Erzeugen und Löschen von Kanten, Knoten und Clustern
- Verschieben von Kanten, Knoten und Clustern
- Änderungen des Layouts (Größe, Farbe, Füllung, Sichtbarkeit)
- Cluster-Verwaltung im Explorer-Stil
- Zoomen und Scrollen
- Erzeugen von Zufallsgraphen
- Prüfung des Graphen auf (2-, 3-) Zusammenhangskomponenten
- Automatische Layout-Berechnung in verschiedenen Stilen
- Animation
- und weitere...

Somit stellte der bisherige GDE (Abbildung 4.11) eine umfangreiche, graphische Benutzerschnittstelle zur OGDF-Bibliothek dar. Unser Anliegen war es nun, den GDE dahingehend zu erweitern, dass auch Compounds erzeugt, dargestellt und von uns speziell dafür implementierte Algorithmen für automatisch generierte Layouts angewandt werden können.

4.4.1 Der Editor im Detail

Um diese Erweiterungen bewerkstelligen zu können, haben wir uns zuerst mit der bestehenden Implementierung befasst. Es handelt sich dabei um drei untereinander agierende Teilsysteme:

- die graphische Oberfläche inklusive der Dialoge
- die Anbindung an die OGDF-Bibliothek
- die Verwaltung der internen Objekte und das Event-Handling

Das Projekt wurde in C++ unter Zuhilfenahme der Qt-Bibliothek programmiert, um eine größ tmögliche Portabilität und schnelle Reaktionszeiten zu erreichen. Intern werden für alle Objekte eines gegebenen Graphen (Kanten, Knoten, Cluster) eigene Objekte erzeugt und verwaltet. So kann der GDE zwischen den zu zeichnenden Objekten, die für ihn eine feste Position im Hauptfenster haben und vom Benutzer beliebig hin- und hergeschoben werden können, und den Objekten des Graphen, die der Graph selber verwaltet, unterscheiden. Auf diese Art und Weise bleibt der Graph konsistent, denn nur

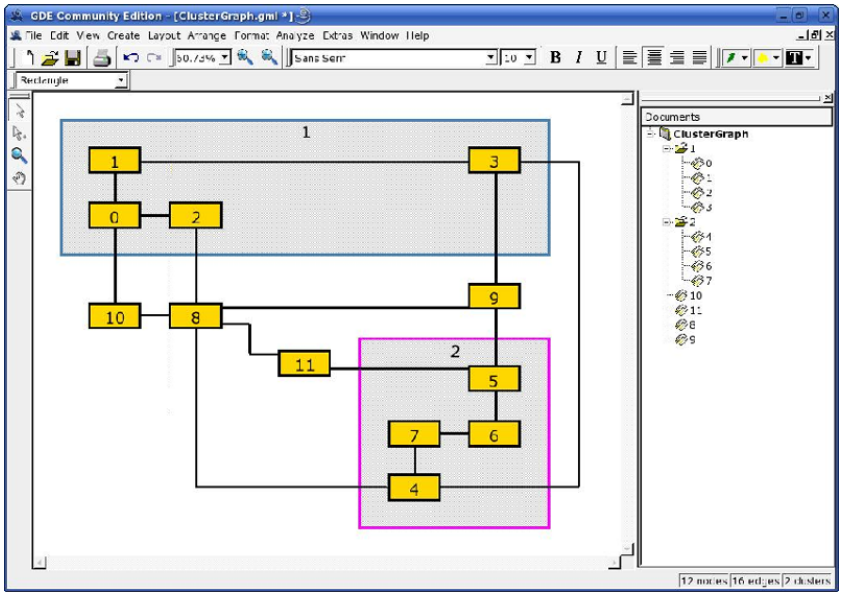

Abbildung 4.11: Das GDE v1.3: Unser Ausgangspunkt vor der Erweiterung um CompoundGraphen

über ihn können Methoden aufgerufen werden, um neue Knoten und Kanten zu erzeugen, für welche wiederum erst dann neue Objekte in der Klasse `GraphDoc` instanziiert werden können, welche anschließend dem Hauptfenster zum Darstellen durchgereicht werden.

Neben Knoten und Kanten können, wie oben schon erwähnt, auch Cluster erzeugt werden. Damit innerhalb des GDE nicht verschiedene Graph-Typen („normaler“ Graph und Cluster-Graph) berücksichtigt werden müssen, wird intern jeder Graph als Cluster-Graph behandelt. Dafür wird einfach um jeden Graphen, egal ob Cluster-Graph oder nicht, ein „unsichtbares“ Cluster herumgezogen. Somit besitzt jeder einfache Graph genau ein ihn umgebendes Cluster, und der dazugehörige Cluster-Baum besteht nur aus einem einzigen Knoten. Dadurch kann man auf Fallunterscheidungen verzichten und zu jeder Zeit ein Cluster einem „normalen“ Graphen hinzufügen.

Die ausschließliche Behandlung von Cluster-Graphen im GDE stellte für unsere Erweiterung um Compounds ein konzeptionelles Problem dar. Im OGDF stellt der Compound-Graph im Grunde nichts anderes als den Inklusionsbaum nach und merkt sich einfach zu jedem Compound, welchem Knoten er entspricht. Ein unbedachter Programmierer könnte somit nun zu einem erzeugten Graphen sowohl einen Cluster- als auch einen Compound-Graphen erstellen, die sich jeweils gegenseitig beinhalten können. Das wäre aber nicht konform zur Definition von Compound-/Cluster-Graphen, denn ein Compound-Graph kann durch fehlende Kanten zwischen Compounds einen Cluster-Graphen darstellen, umgekehrt aber nicht. Somit sollte auch ein Cluster-Graph keine Compound-Graphen enthalten können. Was in einer Algorithmen-Bibliothek vielleicht noch nebeneinander existieren kann (um einen Cluster-Graphen oder einen Compound-

Graphen zu ermöglichen), das darf im GDE also nicht gleichzeitig auftreten. Dies wäre allerdings unausweichlich, weil bisher *jeder* Graph im GDE, somit also auch ein späterer Compound-Graph, in einem Cluster-Graphen liegt.

4.4.2 Vom Cluster zum Compound

Aus diesem Grund (und im Hinblick auf eine eventuelle, spätere Ersetzung von Clustern durch Compounds im OGDF) vereinbarten wir, den Cluster-Graphen im GDE komplett durch einen Compound-Graphen zu ersetzen. Diese komplette Ersetzung von Clustern im GDE kann aber nicht auf einmal umgesetzt werden, weil viele Algorithmen im ODGF noch auf Cluster-Graphen arbeiten, und weil durch ein sofortiges Umsetzen aller Änderungen im GDE Unübersichtlichkeit entstehen würde. Deshalb haben wir uns zu drei Schritten entschlossen:

1. Umstellung von Cluster- auf Compound-Graph
2. Darstellung eines Compounds
3. Anbindung von (Layout-)Algorithmen und entsprechende Ergänzung von Dialogen

Umstellung von Cluster- auf Compound-Graph

Damit der GDE neben dem Graphen G, der nur aus Knoten und Kanten besteht, auch Cluster darstellen und manipulieren kann, besitzt jede Instanz der Klasse `GraphDoc` eine Referenz auf seinen Graphen G und eine konstante Referenz auf einen Cluster-Graphen, der wiederum selber eine konstante Referenz auf jenen Graphen G besitzt. Wir haben bei der Ersetzung des Cluster-Graphen im GDE durch einen Compound-Graphen darauf geachtet, dies genauso zu übernehmen, denn auf diese Weise wird die Konsistenz innerhalb der Objektverwaltung gewahrt.

Allerdings existieren neben den verschiedenen Algorithmen, die für einen „bloßen“ Graphen ein geeignetes Layout berechnen, auch spezielle Layoutalgorithmen für Cluster-Graphen. Um auch diese Algorithmen nach der Umstellung auf Compound-Graphen benutzen zu können, haben wir zuerst Methoden geschrieben, um Cluster-Graphen in Compound-Graphen zu wandeln und umgekehrt. Somit mussten nicht die Algorithmen für Cluster in der OGDF umgeschrieben werden, sondern ein Compound-Graph kann nun (sofern er keine Kanten zwischen Compounds besitzt, denn dies würde der Definition von Clustern widersprechen) ein Ebenbild von sich als Cluster-Graph erzeugen und diesen dann an die entsprechenden Cluster-Algorithmen durchreichen. Anschließend holt sich der Compound-Graph die Größen und Positionen der Cluster und Knoten aus den Attributen des Cluster-Graphen und übergibt diese seinen eigenen Knoten.

Eine weitere Aufgabe ergab sich durch die Benennung von Methoden und Objekten im GDE-Code sowie auch durch die Bezeichnungen in verschieden Dialogen: Da nun auch vom Anwender generell nur noch mit Compounds statt mit Clustern gearbeitet werden sollte, mussten Dialogeinträge umbenannt werden. Diese einfache, rein textuelle Umbennennung vollzogen wir entsprechend auch für Methodennamen, um den Quellcode

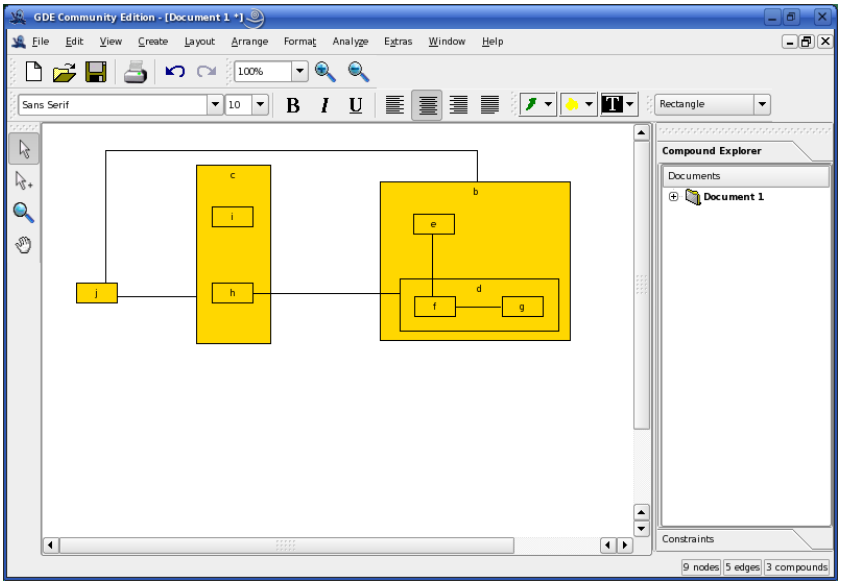

Abbildung 4.12: Beispiel für die Darstellung eines Compound-Graphen im GDE

einheitlich zu halten. ZumBeispiel wurde so `void viewShowClusters(bool clustered)` zu `void viewShowCompounds(bool compounded)`. Dieses ließ sich ohne Probleme umsetzten, nahm dafür aber wesentlich mehr Zeitin Anspruch als erwartet.

Desweiteren prüfen wir nun vor der Ausführung der bereits oben erwähnten Cluster-Algorithmen, ob es Kanten zwischen Compounds gibt. Ist dies nicht gegeben oder akzeptiert der Benutzer die Löschung dieser Kanten, dann lassen sich auch weiterhin alle Algorithmen benutzen.

Somit wäre die Umstellung auf einen Compound-Graphen geglückt ohne bisherige Funktionalitäten zu verlieren. Leider ergaben sich zuerst einige Unstimmigkeiten mit unserer entworfenen Compound-Klasse, welche später noch erläutert werden.

Darstellung eines Compounds

Nachdem der erste Teil erfolgreich umgesetzt wurde, hatte der GDE in erster Linie genau dieselben Funktionalitäten wie vorher. Die graphische Darstellung der Compounds entsprach genau der damaligen Darstellung von Clustern. Nur existierte für jeden der vom Benutzer erstellten Compounds (ehemals: Cluster) nun jeweils ein weiterer Knoten, der aber noch unabhängig vom Compound existierte und deshalb zusätzlich abgebildet wurde. Deshalb sollte als nächster Schritt ein in sich stimmiges Konzept für die Erstellung von Compounds und die Implementierung der darzustellenden Compound-Objekte umgesetzt werden. Das Ergebnis ist in Abbildung 4.12 dargestellt.

Erstellen eines Compounds Bisher wurden neue Cluster erzeugt, indem man im Hauptfenster oder im Cluster-Explorer einen oder mehrere Knoten markierte und mit einem Rechtsklick „Create Cluster“ im Kontextmenü auswählte. Wir sind bei die-

sem Verfahren geblieben, da es von uns für geeignet intuitiv bewertet wurde. Wird desweiteren nun ein Compound erstellt, so wird die gemeinsame BoundingBox der markierten Knoten berechnet und der neue Knoten, der den neuen Compound darstellt, wird um die ausgewählten Knoten herumgelegt. Gleichzeitig werden im Compound-Explorer die Hierarchien entsprechend geändert, so dass dort die zuvor markierten Knoten unterhalb des neuen (Compound-)Knotens liegen. Der neue Compound selber liegt dann unter dem Compound, der als Erster (von unten betrachtet) alle zuvor markierten Knoten als direkte Kinder besitzt oder auf einem Pfad über seine Kinder erreichen kann.

Implementierung In der Version 1.3 des GDE wurde die Darstellung von Knoten und Clustern durch die Implemetierung der jeweiligen Klassen `NodeObject` und `ClusterObject` bewerkstelligt. Schon früh wurde uns bewußt, dass ein direktes Adaptieren von `ClusterObject` als Vorlage für `CompoundObject` zu einer Reihe von Problemen führen würde. Durch eigene Attribute wurden Cluster bisher anders dargestellt als Knoten, was generell auch für Compounds umsetzbar gewesen wäre. Da es sich aber bei Compounds auch um Knoten handelt, so müsste beim Entfernen der Kinder eines Compounds auch das zugehörige `CompoundObject` wieder wie ein Knoten dargestellt werden und umgekehrt. Und abgesehen von einer nicht-trivialen Umwandlung der Attribute wenn ein `NodeObject` zu einem `CompoundObject` (oder umgekehrt) umgewandelt werden muss (denn die `CompoundAttributes` stellen eine Erweiterung der `GraphAttributes dar` und haben somit mehr Eigenschaften), so stellte sich dieser Umsetzung ein weiterer, weitaus problematischerer Punkt in den Weg. Einem `CompoundObject` müssten Möglichkeiten gegeben werden, Kanten zu haben (da es sich ja um einen Knoten handelt). Dies blieb dem `ClusterObject` zuvor verwehrt und würde sich auch nicht so einfach bewerkstelligen lassen. Um nur ein Beispiel zu nennen: Man hätte im weiteren Verlauf auch bei den Kanten unterscheiden müssen, ob diese von einem Knoten zu einem Knoten oder von einem Knoten zu einem Compound verläuft (oder umgekehrt). Da aber mit der Klasse `NodeObject` bereits ein einheitliches und bewährtes Konzept für die Darstellung von Knoten (inkl. Kantenanbindung) vorhanden war, erschien es uns sinnvoller dieses uns auch für Compound von Nutzen zu machen. An einer Stelle ließ sich das Casten zwischen verschiedenen Objekten allerdings nicht verhindern:

Im Compound-Explorer (ehemals: Cluster-Explorer) werden „normale“ Knoten und Knoten, die einem Compound entsprechen, unterschiedlich dargestellt, weil Letztere weitere Knoten und Compounds beinhalten können und sich dies in der Verzeichnisstruktur des Compound-Explorers bemerkbar machen muss. Dafür existieren für jeden Knoten oder Compound ein `NodeItem` oder ein `CompoundItem`. Desweiteren ist es nach der Umstellungvon Clustern zu Compounds möglich, durch das Anhängen von Knoten oder Compounds an bereits existierende Knoten, diese nun zu Compounds zu wandeln. Das muss sich natürlich auch im Compound-Explorer wiederspiegeln, was eben genanntes Casten von `NodeItem` zu `CompoundItem` notwendig macht. Analog dazu verhält es sich im umgekehrten Fall, wenn man einem Compound seine Kinder entfernt.

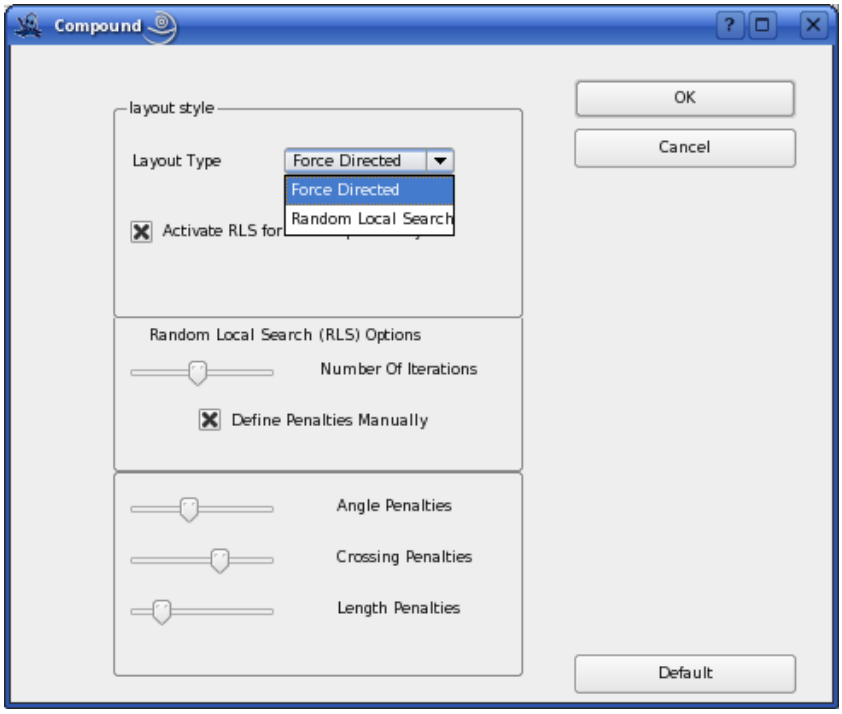

Abbildung 4.13: Das Optionenfenster der Compound-Layout-Algorithmen

Anbindung von (Layout-)Algorithmen

Um unsere in Kapitel Layout-Algorithmen für Compound-Graphen vorgestellten Algorithmen anzubinden, war es zunächst notwendig ein entsprechendes neues Fenster im GDE unter dem Menüpunkt `Layout` zu erstellen. Leider ist dies mit dem QT-Designer nicht möglich, daher mussten alle notwendigen Änderungen von Hand geschehen. Analog zu anderen Layout-Algorithmen können auch das `SimpleLayout` und `RLSLayout` im OGDF direkt aus der Datei `GDEAppWindow-layout.cpp` aufgerufen werden, wobei benutzerdefinierte Parameter aus dem `CompoundLayoutDialog` verwendet werden können. Die Einstellung dieser Parameter kann im GDE direkt bei Aufruf des Layout-Algorithmus geschehen, wie in Abbildung 4.13 zu sehen ist.

4.4.3 Erweiterungen im OGDF

Leider wurden im Verlauf der GDE-Anpassung noch einige Erweiterungen der Compound-Klassen notwendig. Auch wenn die Klassen analog zu den Klassen `ClusterElement` und `ClusterGraph` implementiert wurden, so haben wir alles, was uns nicht notwendig erschien, erst einmal außen vor gelassen. Vieles wurde dann im Laufe der Anpassung wieder notwendig, um nicht tiefgreifende Veränderungen in der bestehenden Objektverwaltung des GDE machen zu müssen. Zum Beispiel musste ein Compound sein eigenes Vorkommen in den Speicherlisten/-arrays anderer Objekte verwalten und dafür auch von einer Observer-Klasse erben. Desweiteren mussten Methoden, die Änderungen

im Graphen vornahmen, komplett ausgelagert werden, um diese Graphveränderungen dem Compound-Graphen nicht mehr zu ermöglichen. Diese nachträgliche und teilweise mühselige Aufbereitung der bereits im ersten Semester implementierten Compound-Klassen führten allerdings dazu, dass ein Compound-Graph nun ohne Einschränkungen mit dem gesamten OGDF kompatibel ist, und dass er nun in das Konzept des GDE's passt.

4.5 Fazit

Im Verlaufe der letzten zwei Semester entwickelten wir zunächst ein Konzept zur Integration von Compound-Graphen in das OGDF, welches wir recht geradlinig umsetzen konnten. Danach implementierten wir einen einfachen Layout-Algorithmus, welchen wir zunächst mit einem kleinen, selbstgeschriebenen Grafikfenster testeten und für gut befanden. Schließlich gelang es uns einen randomisierten Algorithmus anzuschließen, welcher das Layout weiter verbesserte. Parallel dazu gelang es unter einigem Aufwand, Compound-Graphen ins GDE zu integrieren. Dort lösten sie als mächtigere Struktur die Cluster Graphen ab. Schlußendlich waren wir auch in der Lage, Compound-Graphen betreffend, den Schulterschluß zwischen GDE und OGDF herzustellen.

5 Arbeitsgruppe Constraints

Constraints sind in der Welt der Graphenzeichnung Bedingungen oder Beschränkungen, welche bei der Darstellung des Graphen oder dessen Verarbeitung durch einen Layoutalgorithmus zu berücksichtigen sind. Häufig sind solche Bedingungen von geometrischer Natur, wie z. B. die Fixierung von Positionen von Graph-Elementen oder dem Verbot von Kreuzungen bestimmter Kanten. Es handelt sich hierbei um ein aktuelles Forschungsthema.

Die Arbeit der Constraint-Subgruppe innerhalb der PG478 gliederte sich in zwei große Zeit- und Arbeitsphasen. Im ersten Semester (WS05/06) beschäftigte man sich mit der Einbindung einer Auswahl von Constraints in die bestehende Struktur des OGDF sowie der Realisierung eines modifizierten Force-Directed Layout-Algorithmus, welcher diese Constraints berücksichtigt. Das zweite Semester (SS06) diente der Entwicklung einer grafischen Benutzerschnittstelle für diese Constraints im *GoVisual Diagram Editor*. Diesen großen Arbeitsphasen entsprechen die Kapitel 5.1 und 5.2.

5.1 Auswahl und Integration von Constraints in das OGDF

Kapitel 5.1.1 bietet einen ersten Zugang zum Thema Constraints in Graphen. Dort wird eine Menge von Constraints kurz vorgstellt. Im Wesentlichen beschreiben wir dort, wie wir in den ersten Sitzungen vorgegangen sind und warum wir uns letztlich für die in Kapitel 5.1.2 beschriebenen Constraints entschieden haben. In Kapitel 5.1.3 stellen wir unser entwickeltes Modell für das OGDF vor und erläutern, wieso wir uns für diese Struktur entschieden haben. Hier werden auch die implementierten Klassen näher erläutert. Anschließend, in Kapitel 5.1.5, wird der Layout-Algorithmus vorgestellt. Eine Bewertung der Resultate der ersten Arbeitsphase wird in Kapitel 7.3 gegeben.

5.1.1 Auswahl der Constraints

In den ersten Sitzungen haben wir uns damit beschäftigt, uns einen Überblick über die Vielzahl der Anwendungsgebiete im Bereich des Graphenzeichnens zu verschaffen. Als Recherche- und Informationsquelle diente dazu in erster Linie das WWW. Auf Grundlage dessen haben wir dann zusammengetragen, was es überhaupt für mögliche Constraints gibt. Dabei stand zunächst nicht unbedingt im Vordergrund, wie sinnvoll oder zweckmäßig ein bestimmter Constraint wohl ist. Primär ging es uns erstmal darum, eine möglichst umfangreiche Liste möglicher Constraints zusammenzutragen:

- *Node-Node-Overlap:* Knoten sollen sich nicht überlappen.
- *Node-Edge-Overlap:* Knoten und Kanten sollen sich nicht überlappen.

- *Alignment:* Ausgewählte Knoten sollen auf einer Geraden liegen.
- *Even Spacing:* Knoten sollen den gleichen Abstand zueinander haben.
- *Sequence:* Knotenmengen sollen in einer festen Reihenfolge angeordnet werden.
- *Cluster:* Semantisch zusammengehörige Knoten sollen nah beieinander liegen.
- *T-Shape:* Der Graph soll als Baum dargestellt werden.
- *Symmetry:* Knoten sollen symmetrisch bezüglich ihres Schwerpunkts angeordnet werden.
- *Zone:* Ein ausgewählter Bereich enthält ausschließlich ausgewählte Knoten.
- *Hub-Shape:* Knoten werden kreisförmig um einen Mittelpunkt platziert.
- *Anchor:* Ein Knoten soll seine Koordinaten beibehalten.
- *Solid Adjacency List:* Die Reihenfolge der adjazenten Kanten eines Knotens ist festgelegt.
- *Node Input/Output:* Die Position einer ein-/ausgehenden Kante an einem Knoten ist festgelegt.
- *Absolute Edge Length:* Die Kanten sollen eine absolute Länge haben.
- *Relative Edge Length:* Die Kanten sollen eine relative Länge haben.
- *Minimal/Maximal Edge Length:* Die Kanten sollen eine bestimmte Länge nicht über-/unterschreiten.
- *No Edge Bend:* Eine Kante soll keine Knicke haben.
- *No Edge Crossing:* Bestimmte Kanten sollen sich nicht schneiden.
- *Horizontal/Vertical Edges:* Es soll nur horizontale/vertikale Kanten geben.
- *Parallel Edges:* Bestimmte Kanten sollen parallel zueinander verlaufen.

Als nächstes haben wir uns dann entschieden, welche Constraints wir implementieren möchten. Dabei haben wir auch berücksichtigt, dass diese auch möglichst von einem Force-Directed-Layout-Algorithmus umgesetzt werden können, bzw. überhaupt in einem Force-Directed-Layout Sinn machen, da eben dies eine unserer weiteren Aufgaben war. Wir haben uns dafür entschieden, die Constraints Anchor, Alignment und Sequence zu implementieren.Die Implementierung soll unabhängig von einem bestimmten Layout-Algorithmus sein. Unsere Kleingruppe erweitert den Force-Directed-Layout-Algorithmus so, dass er die drei Constraints so gut wie möglich erfüllt. Hierbei soll der Layout-Algorithmus und die Menge der Constraints problemlos erweiterbar sein.

5.1.2 Implementierte Constraints

Anchor-Constraint

Der Anchor-Constraint besteht darin, einen oder mehrere Knoten, Cluster oder Compounds auf der Zeichenfläche zu verankern. Das bedeutet, dass die momentane Position nach der Ausführung eines Layout-Algorithmus möglichst nicht verändert werden soll. Dies ist im Hinblick auf „preserving the mental map“ ein sinnvoller, häufig gebrauchter/wünschenswerter Constraint. Außerdem gibt es durchaus Anwendungen im Bereich des Graph-Drawing, bei denen bestimmte Knoten eines Graphen auch nur bestimmte Positionen im Graphen einnehmen sollten. So können zum Beispiel syntaktische oder semantische Konventionen eingehalten werden.

Da dieser Constraint auch nicht zuviel Aufwand erfordert, haben wir beschlossen, ihn zuerst zu implementieren.

Zwei Varianten des Anchors wurden angedacht:

- *Hard Anchor:* Die Koordinaten sind absolut bezogen auf die Bounding-Box der Zeichnung. Das Objekt soll also gar nicht bewegt werden dürfen.
- *Soft Anchor:* Die aktuelle Position soll möglichst eingehalten werden. Abweichungen sind aber möglich.

Für die Implementierung im Force-Directed macht nur der Soft Anchor Sinn, da der Hard Anchor den Algorithmus zu sehr einschränken würde. Bei einem Force-Directed-Algorithmus sollte kein Knoten auf einen absoluten Wert gesetzt werden. Es würden keine Kräfte auf ihn wirken können, die andere wichtige Constraints oder Layout-Bedingungen repräsentieren.

Da die Implementierung allerdings unabhängig vom Layoutalgorithmus sein soll, kann man zwischen beiden Varianten wählen.

Alignment-Constraint

In vielen Anwendungen ist es wünschenswert für eine bestimmte Menge von Knoten festzulegen, dass sie von einem Layoutalgorithmus grundsätzlich linear angeordnet werden sollen. Die wahrscheinlich häufigste/sinnvollste Anforderung stellt die Platzierung bestimmter Knoten auf einer horizontalen, vertikalen oder diagonalen (virtuellen) Geraden dar. Dies soll dem Benutzer durch diesen Constraint ermöglicht werden.

Es wurden mehrere Varianten angedacht:

- Entweder hat der Nutzer die Auswahl zwischen horizontalem/vertikalem Alignment, oder er kann eine Gerade mit beliebiger Steigung zeichnen.
- Bei gleichbleibender Steigung der Geraden kann sich die genaue Position ändern (relatives Alignment), oder die Position der Geraden ist fixiert (absolutes Alignment).

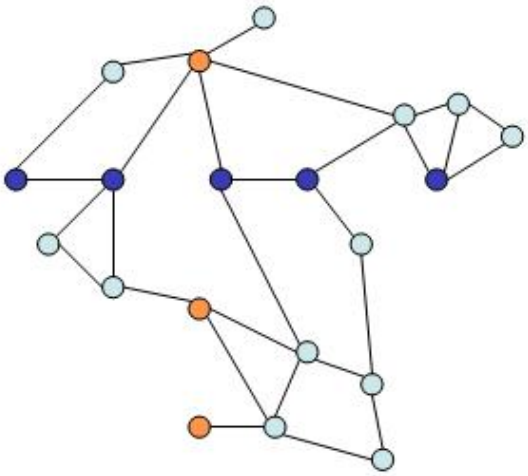

Abbildung 5.1: Beispiel Alignment-Constraint

Wir finden es sinnvoll, nicht nur die Wahl zwischen einer horizontalen oder vertikalen Geraden zu haben. Der Benutzer sollte die Gerade frei auswählen können. Eine Fixierung der Geraden wäre eine sehr starke Einschränkung des Layoutalgorithmus, und eventuell wünscht der Nutzer nur eine relative Anordnung der Knoten. Falls eine Fixierung gewünscht ist, kann der Alignment-Constraint mit einem Anchor-Constraint kombiniert werden.

Deswegen haben wir uns für eine bewegliche Gerade mit beliebiger Steigung entschieden. Die Funktion eines Alignment-Constraints soll hier noch kurz visualisiert werden. In Abbildung 5.1 sieht man einen Graphen, bei dem die orange bzw. hell markierten Knoten zu einem vertikalen Alignment-Constraint gehören, und die blauen bzw. dunklen Knoten zu einem horizontalen.

Sequence-Constraint

Der Benutzer kann mehrere Mengen von Knoten auswählen und diese relativ zueinander in Beziehung stellen. Die Möglichkeiten zur Auswahl der relativen Lage beschränken sich dabei auf die beiden Arten „Menge A links von Menge B“ und „Menge A oberhalb von Menge B“. Der Constraint ist unter anderem sinnvoll um zeitliche Abläufe hervorzuheben.

Es galt hierbei zu entscheiden, ob wir es nur ermöglichen, jeweils zwei Knoten miteinander in Beziehung zu setzen, oder jeweils zwei Mengen von Knoten. Wahrscheinlich wird es häufig so sein, dass ein User nur zwei Knoten (oder zumindest sehr vereinzelte) miteinander topologisch in Beziehung setzen will. Das Mengen-Design stellt aber keine Einschränkung oder überflüssigen Overhead dar, da zwei einzelne Knoten als einelementige Mengen angesehen werden können. Die Wahl ist auf die Mengen-Repräsentation

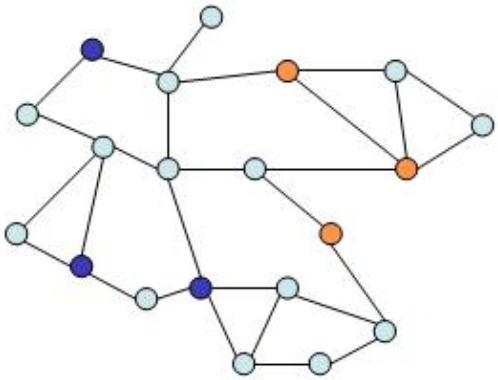

Abbildung 5.2: Beispiel Sequence-Constraint

gefallen, aus folgenden Gründen:

- Die Zweier-Beziehungen bieten Vorteile bei der Widerspruchsprüfung. Allerdings lässt sich die Mengen-Darstellung ohne Weiteres in die Zweier-Beziehungen umrechnen.
- Der Benutzer hat es auf diese Weise einfacher, wenn er zwei Gruppen von Knoten zueinander in Beziehung setzen will, es ihm aber egal ist, wie die Knoten innerhalb einer Gruppe angeordnet sind. Dies ist dann somit nur eine Constraint-Instanz.

In Abbildung 5.2 ist ein Graph dargestellt, der ein Sequence-Constraint vom Typ „blaue bzw. dunkle Knoten links von orangen bzw. hellen Knoten“ enthält.

5.1.3 Vorüberlegungen und Anforderungen an das Modell

Bei dem Entwurf eines Konzepts, Constraints im OGDF und GDE zu unterstützen und schließlich auch einen Force-Directed-Layoutalgorithmus zu implementieren, der diese umsetzen kann, standen folgende Anforderungen und Wünsche im Vordergrund:

- Erweiterbarkeit:

  Aufgrund der Fülle von Anwendungen im Bereich des Graphenzeichnens und der großen Menge verschiedener Constraints, soll sich das Modell leicht um neue Constraints erweitern lassen. Leicht bedeutet in diesem Sinne, dass dies möglichst mit keinen oder zumindest nur sehr wenigen Änderungen im Quellcode zu verwirklichen ist. Besonders im Hinblick darauf, dass es später auch möglich sein soll, den OGDF als Algorithmen-Bibliothek zu nutzen, muss ein Modell geschaffen werden, welches keine Änderung an Dateien oder Klassen erfordert, die nicht unmittelbar mit dem neu implementierten Constraint in Zusammenhang stehen.

- Unabhängigkeit von Layout-Algorithmen:

  Sowohl die Constraint-Implementierungen als auch das Modell selber sollen unabhängig von Layout-Algorithmen sein und somit keine diesbezüglichen Informationen oder Abhängigkeiten beinhalten. Die layout-spezifische Interpretation und Umsetzung der übergebenen Constraints soll allein dem ausgewählten Algorithmus vorbehalten sein.

- Weitgehende Unabhängigkeit vom Dateiformat für Graphen:

  Um oben genannte Erweiterbarkeit auch umsetzen zu können, müssen Abstriche beim Dateiformat bzw. der Schema-Definition zur Speicherung der Constraints eines Graphen gemacht werden. Einzelne Constraint-Typen dürfen nicht fest in der Schema-Definition integriert sein, da dies sonst zusätzlich eine Modifikation des Schemas und des Parsers erfordern würde, wenn ein neuer Constraint-Typ implementiert wird. Es muss nämlich davon ausgegangen werden, dass dem End-Anwender nicht grundsätzlich der vollständige Quellcode zur Verfügung steht, und solche Modifikationen somit nicht durchzuführen sind. Abgesehen davon ist das angestrebte Ziel sowieso, dass der Erweiterungsmechanismus möglichst komfortabel und einfach umzusetzen sein soll.

  Dies erfordert dann zwangsläufig, dass die Schema-Definition im Bereich der Constraint-Speicherung sehr „locker“ gestaltet werden muss, und seine Serialisierung dem Constraint weitgehend selbst überlassen wird. Näheres zu diesem Thema findet sich im Kapitel 6.2.4: „Arbeitsgruppe Dateiformat: Constraints“ .

5.1.4 Entwurf

Im Hinblick auf die zuvor genannten Anforderungen an ein Konzept zur Unterstützung und Umsetzung von benutzerdefinierten Constraints, hat sich unsere Kleingruppe letztendlich unter Absprache mit den Betreuern zur Umsetzung des Modells in Abbildung 5.3 entschieden.

Das oben abgebildete Klassendiagramm stellt die implementierten, zur Umsetzung unseres Konzepts benötigten Klassen in Beziehung. Im Folgenden sollen die Aufgaben und Funktionen der einzelnen Klassen sowie ihre wichtigsten Attribute, Methoden und Abhängigkeiten näher beschrieben werden.

Klasse GraphConstraints

In Analogie zum bereits bestehenden Konzept der Klasse `GraphAttributes`, die layoutspezifische Informationen zu einem Graphen enthält, steht die Klasse `GraphConstraints`. Eine Instanz dieser Klasse beinhaltet eine Liste aller zu dem korrespondierenden Graphen definierten Constraints. Es sind demnach Methoden zum Hinzufügen und Löschen von Constraints `addConstraint(Constraint c)` und `removeConstraint(Constraint c)` vorhanden. Nach dem bisherigen Planungsstand wird einem Constraint unterstützenden Layoutalgorithmus eine Instanz der Klasse `GraphConstraints` übergeben. Es steht also in der Verantwortung des Algorithmus, welche der übergebenen Constraints er

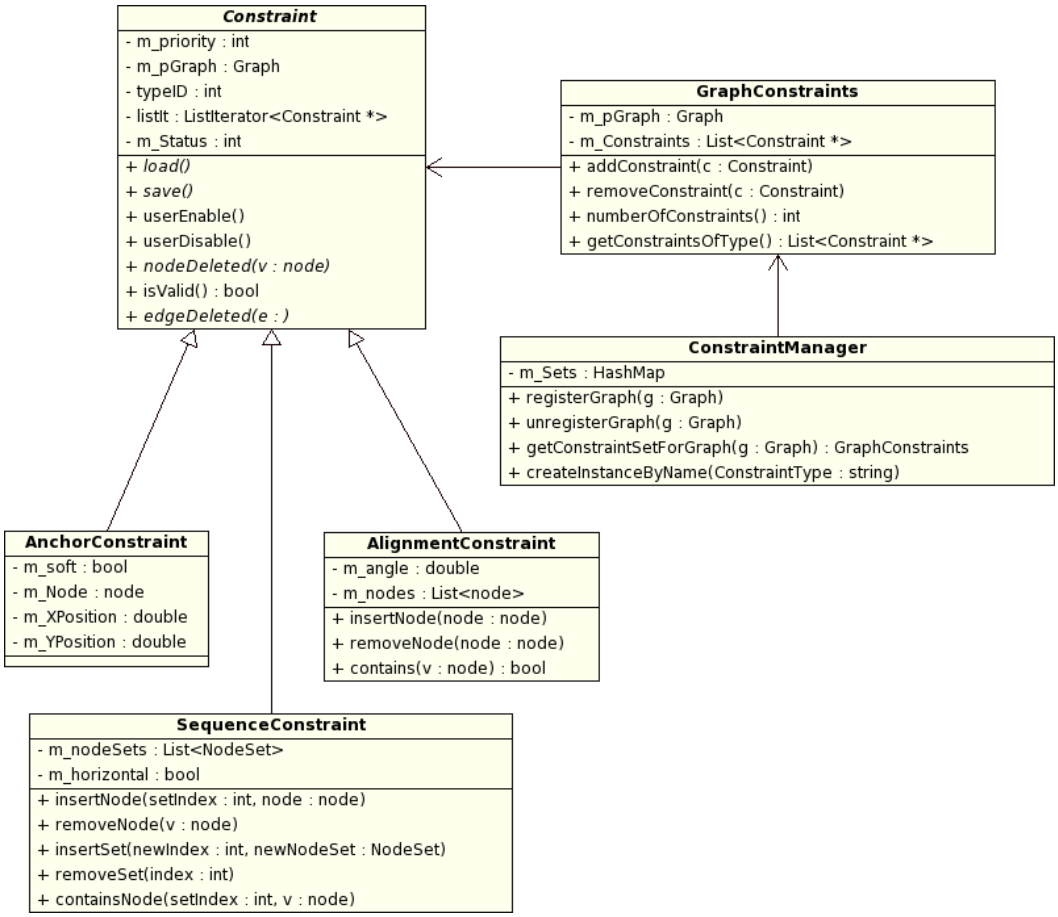

Abbildung 5.3: Klassendiagramm zur Constraint-Umsetzung

bei der Berechnung des Layouts berücksichtigt. Da nicht alle definierten Constraints zwangsläufig auch von diesem umzusetzen sind, weil sie z.B. von dem Layoutalgorithmus nicht unterstützt werden, muss es dem Algorithmus möglich sein, die für ihn relevanten Constraints herauszufiltern. Dazu dient die Methode `getConstraintsOfType(int ConstraintType)`, die eine Liste aller Constraints eines bestimmten Typs (spezifiziert durch `ConstraintType`) zurückgibt. Von den so herausgefilterten Constraints lässt sich dann auch noch seine Gültigkeit abfragen mit der entsprechenden in der Klasse `Constraint` definierten Methode `isValid()`.

Des Weiteren müssen Constraints Änderungen im korrespondierenden Graphen mitverfolgen. Denn wenn zum Beispiel Knoten oder Kanten im Graphen gelöscht werden, für die ein oder auch mehrere Constraints definiert sind, müssen diese dann natrlich darüber informiert werden und sich entsprechend aktualisieren. Um solche Ereignisse mitzubekommen, erbt die Klasse `GraphConstraints` von der Klasse `GraphObserver`, die Veränderungen des Graphen registriert. So kann im Falle einer Knoten- oder auch Kantenlöschung mittels der vererbten Methoden `nodeDeleted(node v)` und `edgeDeleted (edge e)` diese Information an die Constraints weitergereicht werden.

Klasse ConstraintManager

Der ConstraintManager stellt die zentrale Verwaltungs-Instanz in unserem Modell dar. Er verwaltet die Constraints aller existierenden Graphen und ist für die eindeutige Zuordnung, sowie die konsistente Erzeugung der Constaint- und GraphConstraint-Objekte beim Laden eines Graphen verantwortlich. Durch ihn werden demnach auch bereits gespeicherte Constraints und neu definierte Constraint-Objekte instanziiert. Der ConstraintManager muss demnach Methoden zum Erzeugen neuer Constraints implementieren. Wird ein neuer Constraint-Typ implementiert, so muss der ConstraintManager entsprechend modifiziert werden.

Klasse Constraint

Um eine einheitliche Behandlung aller implementierten Constraints sicherzustellen und im Hinblick auf die Erweiterbarkeit wurde eine abstrakte Superklasse `Constraint` implementiert, von der jeder neue Constraint-Typ erbt.

Das Attribut `typeID` wird fortlaufend an neu implementierte Constraint-Typen vergeben und dient somit der eindeutigen Identifikation eines Constraint-Typs.

Im Zuge unserer Überlegungen und unter Absprache mit den Betreuern sind wir in Bezug auf die Gültigkeit eines Constraint-Objekts zu folgender Entscheidung gekommen: ein vom Benutzer definierter Constraint kann unter Umständen syntaktisch ungültig werden, wenn der Graph geändert wird. So könnte zum Beispiel ein Knoten gelöscht werden, zu dem ein Anchor-Constraint definiert ist, oder der zu einem Alignment-Constraint gehört, welches durch die Löschung des Knotens vielleicht keinen Sinn mehr macht. Grundsätzlich sollte aber kein Constraint aufgrund solcher Ereignisse automatisch gelöscht werden, denn:

- Evtl. werden sehr komplexe Constraint-Typen implementiert, zu denen viel mehr

Attribute existieren, als es zum Beispiel bei unserer Implementierung des Anchor-Constraints der Fall ist. Es wäre daher nicht sinnvoll das ganze Constraint-Objekt aufgrund eines solchen Ereignisses zu löschen, wenn es nicht explizit vom Anwender gewollt ist.

- Die Löschung kann lediglich temporärer Natur sein. Der User möchte vielleicht den entsprechenden Knoten eines Anchor-Constraints nur durch einen anderen austauschen, um verschiedene Dinge auszuprobieren.

- Es stellt kein Problem dar, ungültige Constraints bei der Erzeugung eines Layouts zu ignorieren.

Der Anwender sollte außerdem die Möglichkeit haben, selber zu entscheiden, ob bestimmte Constraints bei der Ausführung eines Layout-Algorithmus überhaupt berücksichtigt werden sollen, ohne dass diese dazu gelöscht werden müssten. Denn evtl. sind einige definierte Constraints in manchen Graphenlayouts, die der Anwender ausprobieren möchte, nicht sinnvoll. Deshalb sollen Constraints vom Anwender deaktivierbar und aktivierbar sein. Dazu dienen die Methoden `userEnable()` und `userDisable()`, die das Attribut `mStatus` entsprechend setzen. Ein Layoutalgorithmus kann dann durch die Methode `isValid()` den Status eines Constraints abfragen und daran entscheiden, ob dieser bei der Layout-Berechnung mit berücksichtigt wird. Als Status eines Constraints sind mehr als nur die beiden Möglichkeiten „gültig“ und „ungültig“ vorgesehen. Es muss nämlich unterschieden werden können, ob zum Beispiel ein Constraint-Objekt aufgrund syntaktischer Inkorrektheit deaktiviert ist, ob es explizit vom Anwender deaktiviert wurde oder sein Status anderer Herkunft ist. Ein Beispiel für „anderer Herkunft“ wird im Folgenden gegeben:

Ein kürzlich erst neu angedachtes Feature im Bereich des Zusammenspiels von verschiedenen Constraints ist der `ConstraintChecker`. Dies ist ein erweiterbares Konzept um für einen Graphen definierte Constraints auf gegenseitige Widersprüchlichkeit zu überprüfen. Erweiterbar heißt hier, dass die Klasse `ConstraintChecker` als Interface dient, und mehr oder weniger beliebige Checker zum Abgleich bestimmter Constraint-Typen implementiert werden können. Möchte zum Beispiel ein Anwender wissen, ob sich von ihm definierte Constraints vom Typ Anchor mit Constraints vom Typ Sequence widersprechen, so könnte er mit diesem Konzept einen entsprechenden ConstraintChecker „AnchorSequenceChecker“ implementieren. Wenn von diesem dann sich gegenseitig ausschließende oder zueinander inkonsistente Constraints gefunden wurden, so könnten diese durch entsprechendes Setzen des Attributs `mStatus` markiert werden.

Für einen Layout-Algorithmus ist es nun wichtig, diesen Fall von z.B. einer Deaktivierung des Constraints durch den Anwender unterscheiden zu können. Denn ob durch einen ConstraintChecker markierte Constraints auch tatsächlich als „ungültig“ anzusehen sind, soll dem Layoutalgorithmus selbst überlassen bleiben.

Zuletzt gibt es noch die Methoden `load()` und `save()`, die von jedem neuen Constraint-Typ implementiert werden müssen. Sie dienen folglich dem Serialisieren und Deserialisieren eines Constraint-Objekts dieses Typs.

Mit dem Attribut `priority` war angedacht, dass der Anwender die Constraints mit Prioritäten versehen kann. Im Fall, dass ein Layout-Algorithmus bestimmte Constraints definitiv nicht gleichzeitig umsetzen kann, sondern jeweils nur eine Teilmenge (weil diese sich z.B. gegenseitig widersprechen), so könnte dem Algorithmus mit diesem Attribut die „Wichtigkeit“ eines bestimmten Constraints bekannt gemacht werden. In dem von uns implementierten Force-Directed-Layoutalgorithmus wird dies jedoch nicht umgesetzt werden.

Klasse AnchorConstraint

Die Klasse `AnchorConstraint` erbt von der Klasse `Constraint`. Sie hat vier Attribute. Die boolsche Variable `m_soft` wird auf `true` gesetzt, falls der Benutzer den Soft-Anchor wählt. Sonst `false`. Das Attribut `m_Node` ist vom Typ `node` und speichert den betroffenen Knoten. Die Attribute `m_XPosition` und `m_YPosition` vom Typ `double` legen die gewünschten Koordinaten fest.

Die Methoden beschränken sich auf set- und get-Methoden, da der Constraint unabhängig vom Layout-Algorithmus implementiert werden soll.

Klasse AlignmentConstraint

Die Klasse `AlignmentConstraint` erbt von der Klasse `Constraint`. Das Attribut `m_angle` vom Typ `double` repräsentiert den Winkel der Geraden, auf der die zugehörigen Knoten `m_nodes` vom Typ `NodeSet` angeordnet werden sollen. Da der Benutzer den Constraint jederzeit ändern kann, gibt es die Methoden `insertNode(node v)`, die den Knoten `node` der Menge hinzufügt, `removeNode(node v)`, die den Knoten `node` aus der Menge entfernt, und `bool contains(node v)`, die überprüft, ob ein Knoten `v` zum Constraint gehört.

Klasse SequenceConstraint

Die Klasse `SequenceConstraint` erbt von der Klasse `Constraint`. Sie besitzt als Attribut eine boolsche Variable `m_horizontal`, die auf `true` gesetzt wird, wenn der Benutzer die betroffenen Knoten horizontal anordnet. Das heißt, die erste Menge liegt links von der zweiten, die zweite links von der dritten usw. Vertikale Anordnung (`m_horizontal=false`) bedeutet in dem Zusammenhang eine Anordnung von oben nach unten. Ein weiteres Attribut ist `m_nodeSets`, eine Liste von Knotenmengen. Wichtig hierbei ist, dass die Liste die Reihenfolge der Knotenmengen wiedergibt. Auch hier gibt es die Methoden `insertNode(int setIndex, node v)`, die den Knoten `node` der Menge `setIndex` hinzufügt, und `removeNode(node v)`, die den Knoten `v` aus der Menge entfernt. Zusätzlich gibt es die Methoden `insertSet(int newIndex, NodeSet newNodeSet)`, die an der Position `newIndex` die Knotenmenge `newNodeSet` einfügt, und `removeSet(int index)`, die die Knotenmenge an der Position `index` entfernt. Die Methode `bool containsNode(int setIndex, node v)` liefert `true`, falls der Knoten `node` in der Knotenmenge `setIndex` enthalten ist, `false` sonst.

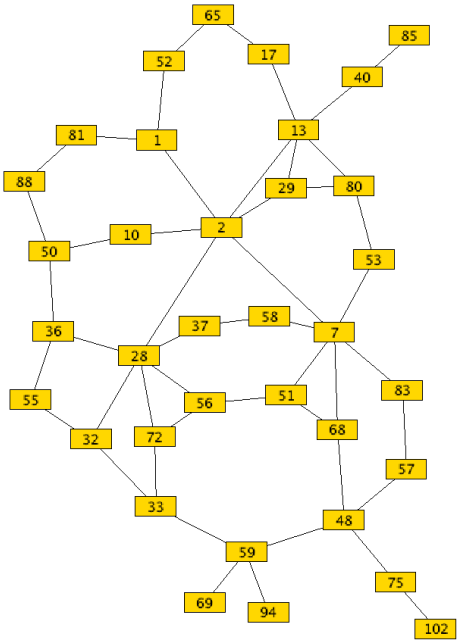

Abbildung 5.4: Ausgabe des aktuellen Algorithmus im GDE

5.1.5 Implementierung des Layoutalgorithmus

Der Layoutalgorithmus

Für die Arbeitsgruppe Constraints wurde als Ziel festgelegt, dass die von den Teilnehmern gewählten Constraints auch in einen Layoutalgorithmus eingebaut werden. Hierfür sollte das Konzept des Force-Directed-Layouts dienen. Das Force-Directed-Layout beruht auf dem Prinzip der anziehenden und abstoßenden Kräfte zwischen Knoten. Kanten werden hierbei als Federn betrachtet, die eine gegebene Restlänge haben. Unterhalb der Restlänge entsteht eine abstoßende, oberhalb eine anziehende Kraft. Um Knoten, die nicht durch eine Kante (hier Feder) verbunden sind, zu verteilen, existiert eine abstoßende Kraft zwischen jedem Knotenpaar. Die von uns verwendete Kraftfunktion für die Gesamtkraft $f(v)$ des Knotens v für einen ungerichteten Graphen $G = (V, E)$ im Simulationsschritt i lautet:

$$f_i(v) = \sum_{(v,w)\in E} \frac{\vec{w} - \vec{v}}{2} - \sum_{u \in V} \frac{\vec{u} - \vec{v}}{|\vec{u} - \vec{v}|} \cdot \frac{1}{|\vec{u} - \vec{v}|^2} + f_c(v)$$

In Worten: Anziehende Kräfte - Abstoßende Kräfte + Kräfte der Constraints. In dem Teil der Funktion ist kein Einfluss der Restlänge zu erkennen. Dies wird aktuell durch Skalieren der Knotenpositionen nach der Simulation erreicht. Für eine Realisierung der Constraints wurde folgendes allgemeingültiges Konzept entwickelt:

1. Aufbereitungsphase: Die Constraint-Information muss in eine für den Force-Directed-Algorithmus brauchbare Form gebracht werden. Die benutzten Datenstrukturen

der Constraints sind eher mengenorientiert, wohingegen der FD-Algorithmus knotenorientiert ist.

2. Zielbestimmung: Für jeden Knoten werden mehrere Ziele erstellt.

3. Approximation: Für jedes Ziel wird eine Anziehungskraft berechnet. Die Summe ergibt die Constraint-Kraftfunktion: $f_c(v)$.

Für die einzelnen Constraints wurden die folgenden Überlegungen angestellt:

Anchor

Der Constraint-Typ Anchor wird mit zwei Optionen integriert:

1. Hard Anchor: Am Ende jedes Simulationsschrittes wird die berechnete Kraft für den korrespondierenden Knoten wieder auf 0 gesetzt, sodass sich insgesamt keine Veränderung der Knotenposition ergibt.

2. Soft Anchor: Die Anchorposition ersetzt die Zielbestimmung, d.h. sie dient als Ziel. Die Anchorposition wird dazu wie ein virtueller Knoten mit Hard-Anchor-Constraint behandelt.

Alignment

Für die Integration des Alignment-Constraints wurde als erstes eine absolute Methode entwickelt, die sich durch zwei Schritte kennzeichnet:

1. Initialisierung: Die Knoten einer Alignment-Instanz werden auf einer Geraden mit der gegebenen Steigung angeordnet bzw. auf sie projiziert.

2. Iterationsschritt: Während jeder Iteration in der Simulation werden die Kräfte aller beteiligten Knoten gemittelt und als Gesamtkraft der Geraden gewertet. Zusätzlich werden die Kraftvektoren auf die Gerade projiziert.

Ein erstes Ergebnis ist in Abbildung 5.5 zu sehen. Dieses Konzept wurde allerdings wieder verworfen, da es aufgrund der Initialisierung und der eingeschränkten Bewegungsfreiheit der einzelnen Knoten nicht zu dem Konzept des Force-Directed-Layouts passt. Als endgültige Lösung wurde das vorhergehende Verfahren so geändert, dass die betroffenen Knoten ein auf der Gerade liegendes Ziel anstreben. Dies lässt sich leicht durch die Berechnung der Anziehungskräfte verdeutlichen. Sei $\vec{v}$ Position eines Knotens, der zur Geraden g mit Normalvektor $\vec{n}_g$ und Position $\vec{g}$ hingezogen werden soll.

$$\text{Anziehungskraft} = \text{Kürzester Weg zur Geraden}$$

Um den kürzesten Weg zur Geraden zu berechnen, muss der Lotpunkt (das Ziel) auf der Geraden gefunden werden:

$$\text{Abstand zum Lotpunkt} = (\vec{v} - \vec{g})\vec{n}_g$$

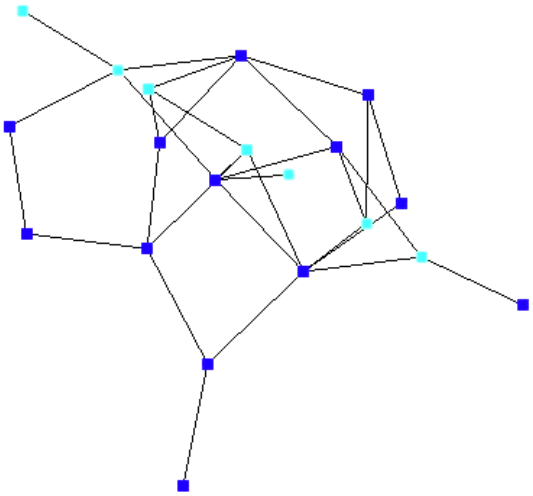

Abbildung 5.5: Erste experimentelle Ergebnisse (helle Knoten gehören dem Alignment-Constraint an)

$$\text{Anziehungskraft} = -\vec{n}_g * ((\vec{v} - \vec{g})\vec{n}_g)$$

Die Implementierung des Ansatzes funktioniert bereits, ein Ergebnis der Anwendung auf einen Baum ist in Abbildung 5.6 dargestellt.

Sequence

Die Knotenmengen, für die durch den Sequence-Constraint eine Ordnung bezüglich der X- bzw. Y-Achse vorgegeben wird, werden durch den Force-Directed-Algorithmus in mehreren Schritten verarbeitet. Dabei wird zwischen zwei Kraftarten unterschieden:

1. Mengenkraft: Die Mengenkraft wird errechnet bezüglich der Lage zu benachbarten Mengen. Sie sorgt dafür, dass die Mengen die korrekte Reihenfolge anstreben. Zur Berechnung der Kraft ist die geometrische Größe der Menge nötig.

2. Knotenkraft: Die Knotenkraft bezeichnet die Kraft des einzelnen Knoten innerhalb der Gruppe. Für jeden Knoten sollte eine Kraft zum Zentrum der Gruppe wirken, um einer räumlichen Ausweitung der Menge entgegenzuwirken.

Zusammenhangskomponenten

Um eine einwandfreie Umsetzung der Constraints zu erreichen, muss es möglich sein auch Constraint-Instanzen, die über mehrere Zusammenhangskomponenten (ZHK) reichen, zu erlauben. Das schon existierende Konzept des Force-Directed-Layouts betrachtet lediglich zusammenhängende Graphen und führt zu Fehlern bei nicht zusammenhängenden

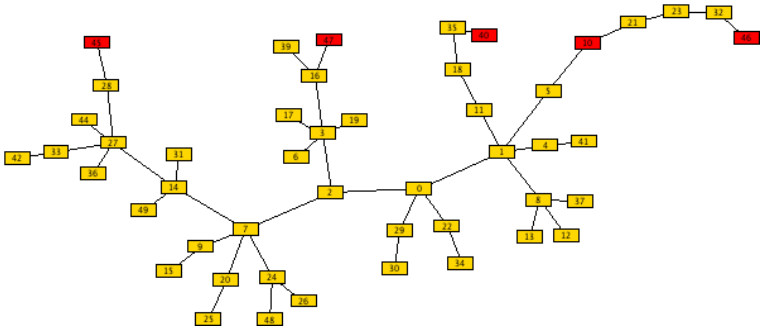

Abbildung 5.6: Ausgabe des aktuellen Algorithmus im GDE (dunkle Knoten gehören dem Alignment-Constraint an)

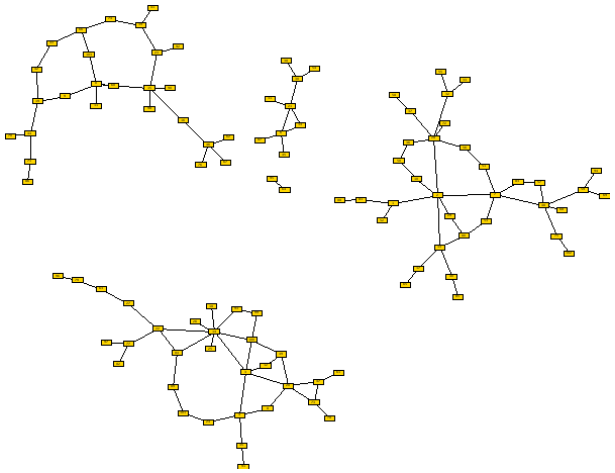

Abbildung 5.7: Ausgabe des GDE

Graphen. Die Ursache hierfür ist bei den globalen Abstoßungskräften zu finden. Da zwischen den einzelnen ZHKs keine Anziehungskraft wirkt, entfernen sich die ZHKs voneinander. In dem schon implementierten Algorithmus wird jede ZHK einzeln behandelt. Nach der Simulation werden die ZHKs einzeln angeordnet. Wie schon oben beschrieben ist für eine einwandfreie Umsetzung/Integration der Constraints eine globale Simulation nötig. Um dies zu erreichen, werden zwischen den ZHKs anziehende Kräfte hinzugefügt. Diese sind abhängig von der Größe der ZHK. In Abbildung 5.7 ist das Ergebnis des Ansatzes dargestellt.

Erweiterungen

Während der Entwicklung wurde der Algorithmus stets durch Versuche erweitert; in der aktuellen Version sind drei Erweiterungen übernommen worden.

Kraftübernahme aus vorhergehender Iteration Um eine Oszillation zu verhindern, schien es nach einigen Experimenten sinnvoll die resultierenden Kräfte mit jenen aus dem letzten Simulationsschritt zu mitteln. Diese Vorgehensweise führt zu dem Effekt der Massenträgheit. Ein Knoten muss, praktisch gesehen, beschleunigt bzw. gebremst werden. Damit wird ein Schwingen der Knoten verhindert.

$$f_i(v) = \frac{f_i(v) + f_{i-1}(v)}{2}$$

Kraftverteilung über Kanten Weiterhin werden Kräfte aus dem vorhergehenden Simulationsschritt auch über Kanten verteilt:

$$\text{Anziehungskraft} = \text{Anziehungskraft} + \sum_{(v,w)\in E} \frac{f_{i-1}(w)}{10}$$

Dreidimensionale Simulation Die Erweiterung der Simulation auf mehr als zwei Dimensionen macht auf den ersten Blick wenig Sinn. Knoten haben aber mehr Platz zu navigieren und es können „Barriere"-Probleme über- bzw. untersprungen werden. Allerdings muss am Ende der Simulation das Ergebnis zweidimensional sein, um im GDE dargestellt werden zu können. Die Lösung hierfür ist ein Zusammendrücken der dritten Dimension über den Lauf der Simulation (durch einen Faktor s), vergleichbar mit einer Gravitationskraft ausgehend von der XY-Ebene.

$$\text{Aktuelle Knotenposition } \vec{p} = \begin{pmatrix} x \\ y \\ z * s \end{pmatrix} s \in [0,1]$$

Die Berechnung in mehr als zwei Dimensionen verbessert das Bewegungsverhalten. Im zweidimensionalen Raum können zwei Knoten, die durch eine Kante verbunden sind, für einen dritten Knoten eine Barriere aufbauen (in Abbildung 5.8 zwischen den Bergen zu erkennen). Der dritte Knoten möchte aufgrund seiner Zielvorgaben zwischen den beiden hindurchwandern, scheitert allerdings an dieser Barriere. Diese Barriere kann bei einer Simulation im Raum durch Ausweichen passiert werden und führt zu besseren Bewegungsmöglichkeiten.

Einbau in das OGDF

Der Einbau in das OGDF wurde relativ einfach gehalten. Für den Algorithmus wurde eine neue Klasse erstellt, die von der Superklasse `LayoutModule` abgeleitet wurde. Um die Constraints eines Graphen dem `LayoutModule` zu übergeben, wird die `void call(GraphAttributes AG)` überladen und eine entsprechende Instanz vom Typ `GraphConstraints` übergeben: `void call(GraphAttributes AG, GraphConstraints *gc)`. Der Algorithmus ist selbst verantwortlich für die Aufbereitung der Constraint-Informationen.

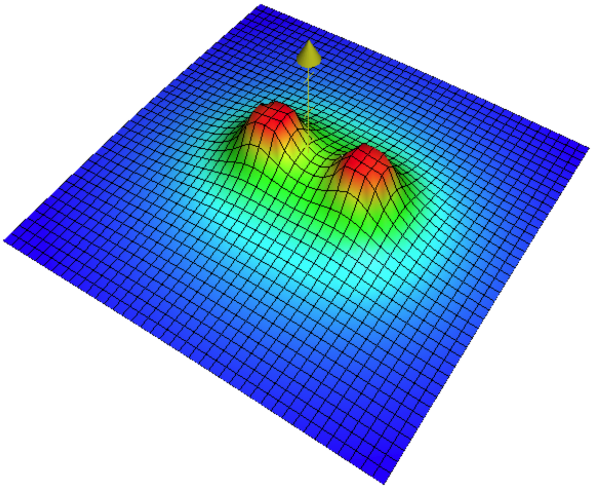

Abbildung 5.8: Die abstoßende Kraft zweier adjazenter Knoten in der Ebene. Die Höhe gibt den Betrag an. Zwischen den beiden Knoten (Bergen) ist die Barriere zu erkennen

5.1.6 Bewertung und Ausblick

Zum Ende des ersten Semesters war die Auswahl der Constraints in das OGDF erfolgt. Jedoch waren noch diverse kleinere Maßnahmen notwendig, um die Auswahl der implementierten Constraints durch den Force-Directed-Layoutalgorithmus behandeln zu können. Auch die Fertigstellung des `ConstraintManagers` verzögerte sich. Zur endgültigen Fertigstellung und auch späteren Erweiterung des GDE war also noch zusätzliche Zeit erforderlich. Wünschenswert für die Mächtigkeit des OGDF und dessen Funktionalität auf lange Sicht ist dessen Ausbau um weitere Constraints.

5.2 Erweiterung des GoVisual Diagram Editors

Nach der erfolgreichen Integration ausgewählter Constraints (siehe hierzu Kapitel 5.1) in das *OGDF* und der Entwicklung eines modifizierten Force-Directed Layoutalgorithmus für diese im ersten Semester der Projektgruppe, sollte nun in logischer Konsequenz die Erweiterung des *GoVisual Diagram Editors* (*GDE*) um Benutzerschnittstellen zur Erzeugung und Konfiguration dieser Constraints erfolgen.

Dieser Aufgabe widmete sich die Gruppe unter einer Neukonstellation der Mitglieder in verschiedenen Arbeitsphasen das ganze zweite Semester (SS06) über. Die zu Beginn des Semesters festgelegten Arbeitsphasen und deren Zeitplanung werden in Kapitel 5.2.1 vorgestellt. Daran anschließend folgt Kapitel 5.2.2, in dem allgemeine Anforderungen und Erfordernisse an die Bedienung der Benutzerschnittstelle sowie Lösungen vorgestellt werden. Kapitel 5.2.3 zeigt die Anwendung des Sequence-Constraints stellvertretend für die anderen Constraints im *GDE*. Kapitel 5.2.4 schließlich stellt den Entwurf und die Implementierungsansätze der wichtigen Komponenten Constraint-Explorer und Constraint-Property-Editor vor. Abschließend wird in Kapitel 5.2.5 das erreichte Resultat kritisch diskutiert und bewertet.

5.2.1 Arbeitsphasen- und Zeitplanung

Die Arbeit der Gruppe Constraints setzte sich im zweiten Semester aus zwei Phasen zusammen. Als erstes wurde durch Gruppenarbeit in Form von regelmässigen Treffen die Anforderungen und ein Konzept erarbeitet (siehe 5.2.2). Da sich die Gruppe sehr schnell auf ein Konzept geeinigt hatte, musste für diese Phase nur ca. 4 Wochen veranschlagt werden. In der zweiten Phase wurde bis zum Semesterende dieses Konzept umgesetzt und implementiert. Die Implementierung erfolgte möglichst unabhängig voneinander um Kollisionen und Wartepausen zu vermeiden. Auf den ersten Blick erscheint die Implementierungsphase sehr lang. Hierbei muss aber berücksichtigt werden, das zu der Hauptaufgabe noch andere Aufgaben/Arbeiten hinzukommen. Dazu zählen Arbeiten wie Quellcodepflege in den Ergebnissen des ersten Semesters und die Behebung von Fehlern.

5.2.2 Anforderungen und Lösungsansätze aus Anwendersicht

Die Hauptanforderung, eine grafische Benutzerschnittstelle in den *GDE* für die bereits genannten Constraints zu integrieren, splittete sich zu Beginn der Problemauseinandersetzung recht schnell in mehrere wichtige Einzelanforderungen.

Da aufgrund der Vielzahl von Constraints (siehe 5.1.1 für eine Aufzählung bekannter Constraints) mit ihren einstellbaren Parametern und ihrer Verschiedenheit untereinander entsprechend viele unterschiedliche Erzeugungs- und Konfigurationsszenarien denkbar sind, war die Erfordernis einer einheitlichen Benutzungsschnittstelle die wichtigste Anforderung. Damit, nicht widersprechend eng verknüpft, war die Forderung, nach Flexibilität und Erweiterbarkeit, gerade im Hinblick auf weitere in das *OGDF* und den *GDE* einzubindende Constraints. Darüber hinaus galt es, die bisherige Struktur von

GDE zu berücksichtigen, so dass Grenzen für ein Design existierten. Dennoch sollte die Schnittstelle möglichst intuitiv, fehlertolerant und komfortabel zu benutzen sein.

Insgesamt kristallisierten sich folgende Erfordernisse als wesentlich heraus, von denen einige zu direkten Lösungsansätzen führten:

- Gleichzeitige Anwendung mehrerer Constraints auf einen Graphen bzw. ein Graph-Dokument
- Sichtbarkeit aller einem Graphen zugeordneten Constraints
- Sichtbarkeit der Status von Constraints, wie z.B. aktiviert, benutzerdeaktiviert oder deaktiviert aufgrund von Konflikten
- Einheitliche, komfortable Parameterkonfiguration von Constraints
- Individuelle Namensvergabe für Constraints
- Selektion/Hervorhebung von beteiligten Graph-Elementen
- Einfache Generierung/Löschung von Constraints
- Einfache Zuweisung von Graph-Elementen an bereits erzeugte Constraints

Die gleichzeitige Anwendung mehrerer Constraints auf ein Graph-Dokument führte schließlich schnell zur konkreten Idee, den bestehenden Compound-Explorer, in dem Hierarchien mehrerer Graph-Dokumente mit ihren (Cluster-/Compound-)Knoten dargestellt werden, mit einem Constraint-Explorer auszubauen. In diesem werden Constraints übersichtlich als Liste inklusive ihres Status dargestellt. Gleichzeitig ermöglicht dieser eine individuelle Namensersetzung für die Standardnamen der Constraints (Anchor, Alignment, Sequence). Der Constraint-Explorer ist also die verwaltende Einheit für Constraints. Im Gegensatz zum Compound-Explorer stellt der Constraint-Explorer aus Übersichtlichkeitsgründen nur die Constraints des aktiven Dokuments dar. Die geschickte Einbettung von Compound- und Cluster-Explorer in einem sog. Toolbox-Widget ist platzeffizient, schnell bedienbar und für erfahrene *GDE*-Benutzer wenig gewöhnungsbedürftig. Ein erster, wenig konkreter Entwurf des Constraint-Explorers ist in Abbildung 5.9 zusehen.

Schwieriger gestaltete sich die Lösung, wie unterschiedliche Constraints und deren Parameter einheitlich und strukturiert dargestellt und konfiguriert werden können. Zunächst favorisierte man unter Missachtung einer einheitlichen Bedieunung jeweils individuelle Konfigurationsmasken für die verschiedenen Constraints unterhalb des Constraint-Explorers als zusätzliches Widget, da Constraints von sehr verschiedener Struktur sind. Eine abstraktere Sichtweise auf Constraints ließ jedoch eine elegante Lösung zu: Attribute, welche alle Constraints aufweisen, sind eine Menge von konfigurierbaren Parametern, sowie Mengen von Graph-Elementen (Knoten oder Kanten), für die das Constraint gilt. Property-Editoren, also Listen und/oder Baumansichten von ganz allgemeinen Parametern, boten sich hier geradezu selbst als eine adäquate Darstellungsform an. Diese

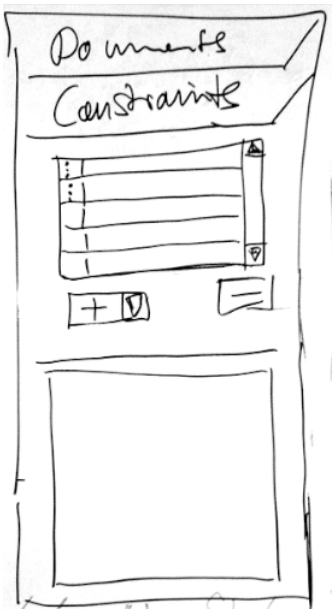

Abbildung 5.9: Erster Entwurf an der Tafel

erlauben eine aufzählende, strukturierte, einheitliche und dennoch individuelle Darstellung und Bedienung, ohne genaue Kenntnis der Parameter. Erweiterbarkeit ist mit ihnen durch einfaches Hinzufügen weiterer Parameter möglich. Vorteilhafte Konsequenzen ergaben sich auch für die Implementierung, weil die Property-Editoren eine generische Modellierung zuließen.

Für die Erzeugung von Constraints bzw. die Zuweisung von Graph-Elementen an Constraints existierten mehrere Lösungsansätze. Diese reichten von wizardgeführten Dialogen, über reine Menüsteuerungen bis hin zu einer komfortablen Drag and Drop Unterstützung. Die Drag and Drop Funktionalität setzte sich wegen ihrer intuitiven und effizienten Benutzung. Sie sollte sich auf die Graph-Zeichenfläche, den Constraint-Explorer und die jeweiligen Property-Editoren beschränken. Zusätzliche Interoperabilität zwischen Compound- und Constraint-Explorer wurde bewußt verworfen, um eine verwirrende und komplizierte Bedienung des *GDE* auszuschließen. Zudem hätte dies einen immensen Änderungsaufwand des Compound-Explorer nach sich gezogen.

Damit waren alle notwendigen Grundlagen aus Anwendersicht gelegt, um sich in der nächsten Arbeitsphase dem Entwurf und der Implementierung des Constraint-Explorers und der verschiedenen Property-Editoren zu widmen.

Zunächst stellt das folgende Kapitel die Erzeugung eines Sequence-Constraints und dessen Konfigurationsmöglichkeiten mittels Abbildungen dar. Dabei wird auch die Benutzung des Constraint-Explorers deutlich.

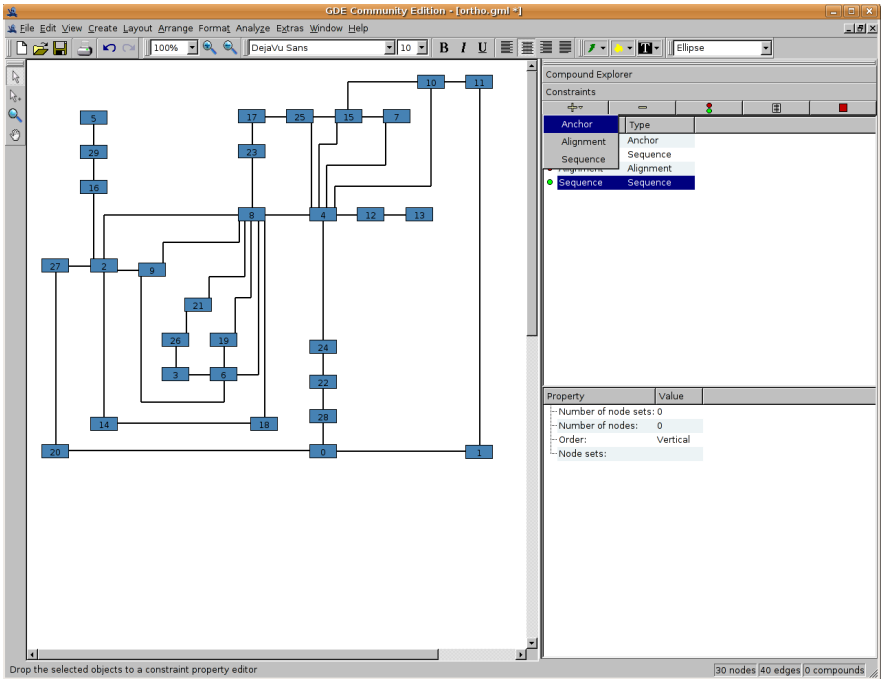

Abbildung 5.10: Hinzufügen eines Constraint zum Constraint-Explorer

5.2.3 Anwendungsszenarium Sequence-Constraint

Der Constraint-Explorer befindet sich auf der rechten Seite des *GDE* innerhalb eines sogenannten Toolbox Widgets zusammen mit dem Compound-Explorer. Die Ansicht kann durch Anklicken auf den entsprechenden Button in den Vordergrund geholt werden, der Compound-Explorer gibt dann seinen Platz frei. Siehe hierzu Abbildung 5.10.

Ist der Constraint-Explorer sichtbar, so kann der Benutzer leere Constraints über den „+“ -Button anlegen oder mittels eines Drags auf diesen ein Constraint erzeugen und gleichzeitig mit Graph-Elementen initialisieren. Die Löschung von Constraints erfolgt über den „-“ -Button, Statusänderungen können mittels des „Ampel“ -Buttons oder der Status-LED direkt neben dem Constraint vorgenommen werden. Ist ein Constraint erzeugt, so wird es in die Constraint-Liste mit seinem Namen aufgenommen. Eine Umbenennung ist innerhalb der Liste möglich wie Abbildung 5.11 zeigt.

Wird im Constraint-Explorer ein Constraint selektiert, so öffnet sich unterhalb desselben der Property-Editor (i.F. P-Editor) für dieses Constraint. In Abbildung 5.12 ist dies für den P-Editor des erzeugten Sequence-Constraint zu sehen. Dieser besteht aus den nicht editierbaren Attributen „Number of node sets“ (Anzahl der Knotenmengen des Constraints), „Number of nodes “ (Anzahl aller Knoten des Constraints), sowie den editierbaren Parametern „Order “ (Ordnung) und „Node sets “ (Knotenmengen). Knoten werden dem Constraint hinzugefügt, indem im Graphen beliebige Elemente selektiert werden und dann per Drag and Drop auf den P-Editor geführt werden. So wird eine neue Knotenmenge aus selektierten Knoten erstellt. Nach demselben Prinzip können

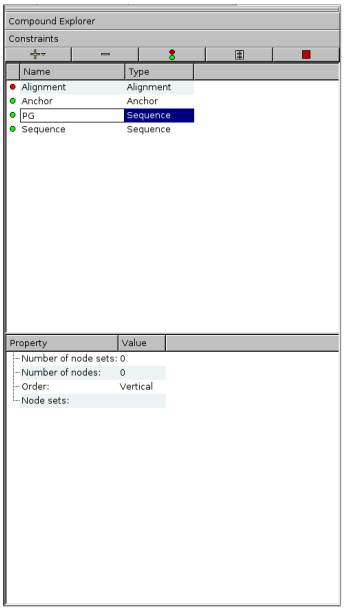

Abbildung 5.11: Umbenennung eines Constraint im Constraint-Explorer

weitere Knotenmengen hinzugefügt werden. Innerhalb des Editors können dann beliebige Selektionen von Knoten/-mengen in andere Knotenmengen wiederum per Drag and Drop verschoben werden, Knoten/-mengen gelöscht werden und die Positionen von Knotenmengen verändert werden. Andere Parameter, beispielsweise die Ordnung, können durch Anklicken auf ihren Wert geändert werden. Abhängig vom Typ des einzustellenden Wertes öffnet sich dann ein Editor-Widget z.B. in Form einer Auswahlbox oder eines Textfeldes. Abbildung 5.12 verschafft einen Überblick über das Kontextmenü einer Knotenmenge. Abbildung 5.13 zeigt die Editierung der Ordnung des Sequence-Constraint.

Die Position von Knotenmengen hat die Semantik, dass bei einer horizontalen (vertikalen) Ordnung die Knotenmengen im Force-Directed Layoutalgorithmus möglichst von links nach rechts (oben nach unten) ohne Überschneidung positioniert werden. Bei der horizontalen Ordnung hat also die erste Knotenmenge im P-Editor die linkeste Position und die letzte Knotenmenge die rechteste Position im Graphen. Bei vertikaler Ordung besitzt die erste Knotenmenge im P-Editor die oberste Position im Graphen usw. Die Abbildungen 5.14 und 5.15 zeigen den Graphen jeweils nach der Anwendung des Force-Directed Layoutalgorithmus mit horizontaler und vertikaler Konfiguration des Sequence-Constraints.

Des Weiteren können beliebige Selektionen im P-Editor in der Graph-Zeichnung selbst hervorgehoben werden. Das bedeutet, die Selektion wird abhängig von der Option „Highlight with zoom“ (Hervorhebung mit Zoom, siehe Abbildung 5.12) ohne oder mit Zoom mittig in der Zeichenfläche des Graphen angezeigt (siehe Abbildung 5.16). Damit ist stets eine Zuordnung zwischen dem Constraint und beteiligten Graph-Elementen möglich. Insbesondere bei komplexen Graphen erweist sich diese Funktionalität als hilfreich.

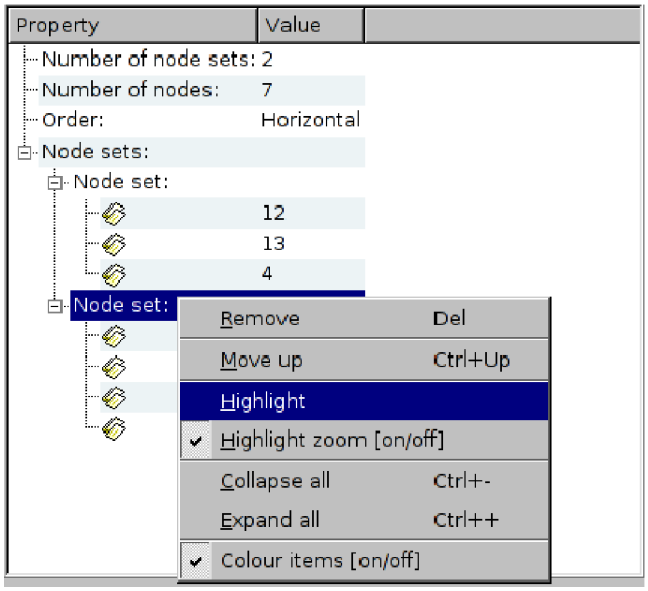

Abbildung 5.12: P-Editor eines Sequence-Constraints mit Kontextmenü

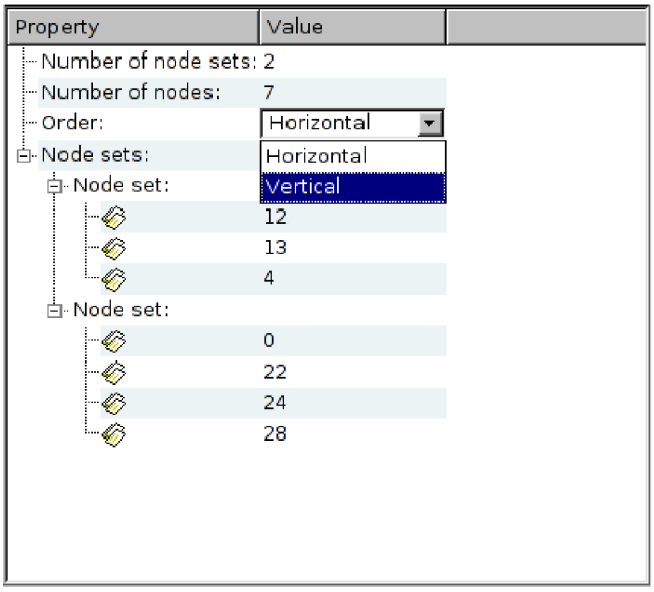

Abbildung 5.13: Auswahlbox zur Editierung der Ordnung des Sequence-Constraints

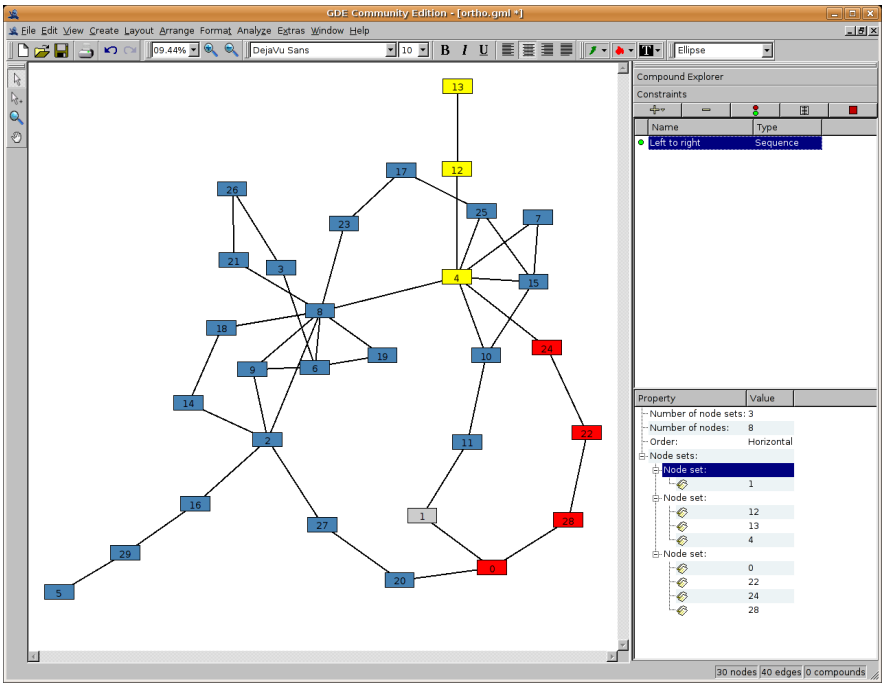

Abbildung 5.14: Graph nach Anwendung des Layoutalgorithmus mit horizontaler Ordnung

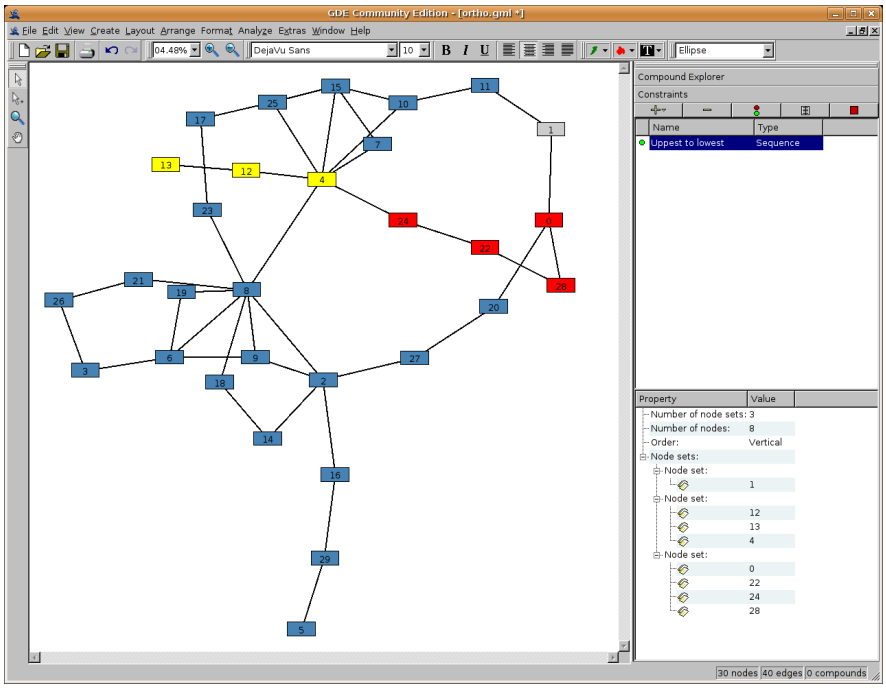

Abbildung 5.15: Graph nach Anwendung des Layoutalgorithmus mit vertikaler Ordnung

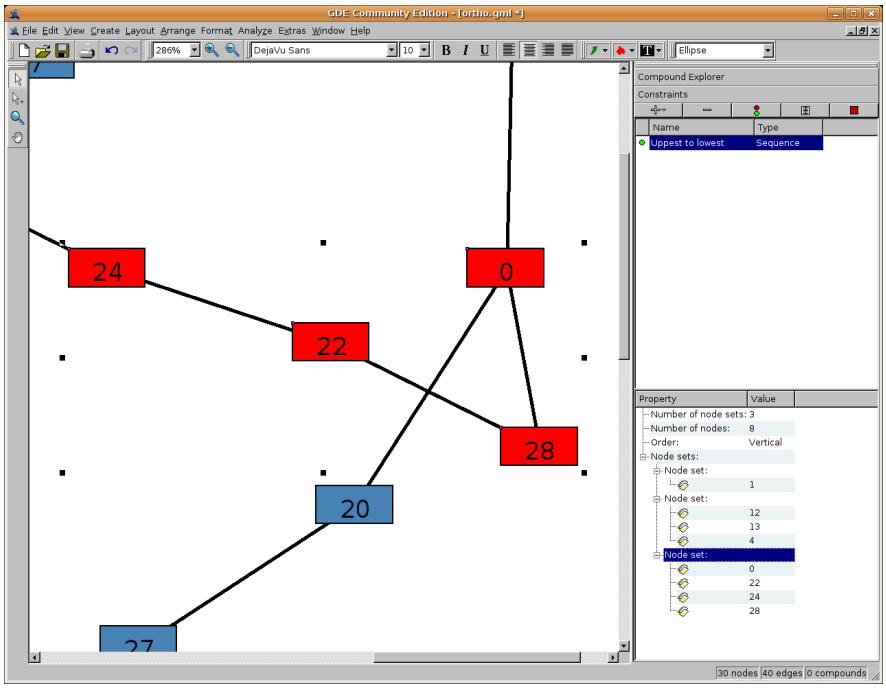

Abbildung 5.16: Hervorhebung einer Knotenmenge mit Zoom

Alle Editieroperationen des P-Editors können über die systemweite Undo-/Redo-Funktionalität des *GDE* rückgängig gemacht oder wiederholt werden. Operationen, z.B. Knotenumbenennungen oder -löschungen innerhalb der Graph-Zeichenfläche oder des Compound-Explorers, ziehen selbstverständlich entsprechende Aktualisierungen im P-Editor nach sich. Auch der Constraint-Explorer verfügt über das bequeme Undo-/Redo-Feature.

5.2.4 Entwurf und Implementierung

Constraint Explorer

Für die Umsetzung des Explorer-Konzeptes waren mehrere Schritte notwendig, die sich gruppieren lassen in:

- Modifikation, Ergänzung bestehender GDE Klassen
- Erstellung des Widgets und der eingebetten Steuerelemente
- Implementierung der Funktionalität
- Überarbeitung

Zuerst musste der bestehende Compound-Explorer in eine `QToolBox` umgebettet werden um das Hinzufügen des Constraint Explorer zu ermöglichen. Weiterhin musste die Klasse `GraphDoc` um ein `GraphConstraints` (Datenstruktur des OGDF die eine Liste der zu einem Graphen gehörigen Constraints verwaltet) ergänzt werden. Für dir Erstellung des Widgets musste eine Listensteuerelement entwickelt werden der die Constraints des zum aktuellen `GraphDoc` gehörenden `GraphConstraints` anzeigt. Hierfür wurde von der Klasse `QListView` eine neue Klasse `ConstraintsListView` (entsprechend auch `QListViewItem` und`ConstraintsListViewItem`) abgeleitet. Nach dem groben Entwurf des Widgets mussten die verschiedenen Funktionen implementiert werden:

- Anzeigen der aktuellen Constraints
- Anzeigen des aktuellen P-Editors für den ausgewählten Constraint.
- Hinzufügen eines neuen Constraint unter Angabe des Typs (Anchor, Alignment, Sequence)
- Löschen des ausgewählten Constraints
- Umbennen und de-/aktivieren des ausgewählten Constraints

Nach dem Fertigstellen der Funktionalität stellte sich schnell heraus das es ein Fehler war die Undo-/Redo-Funktionen am Ende zu implementieren. Dies führte beim Constraint-Explorer zu einem erheblichen Mehraufwand und Umstrukturierung des Quellcodes. Für die Implementierung der Undo-/Redo-Operationen wird für jede Operation (Erstellen, Löschen, Statusänderung) eine neue Klasse von `GraphEditOperation` abgeleitet und

eine `void applyRedo` bzw. `void applyUndo` Methode implementiert. Es ist leicht zu erkennen das es mehr Sinn macht die Funktionalität des Constraint-Explorers direkt dort zu implementieren und das Widget des Constraint-Explorers nur zur Anzeige und Interaktion zu benutzen.

Als letzter Schritt in der Implemtierung wurde das Interface komplett überarbeitet und um einige Features ergänzt um die Darstellung und Handhabung zu verbessern. Diese umfassen kleine Änderungen wie alternierende Farbgebung der Zeilen, eine zusätzliche Spalte zur Anzeige des Constrainttyps und das Verwenden einer Werkzeugleiste in der vorhandene Aktionen (Erstellen, Löschen, Statusänderung) eines Constraints und auch zusätzliche Funktionen (Anzeigen/Ausblenden des P-Editors, de-/aktivieren der gesamten Constraintunterstützung) angeordnet sind. Ein höherer Aufwand musste für die Implementierung der Drag'n'Drop Mechanismen betrieben werden. Der Benutzer ist nun in der Lage, ohne Aktivierung des Constraints, Knoten aus dem Dokument auf den Listeneintrag des gewünschten Constraints zu ziehen, um diese hinzuzufügen.

Constraint Property Editoren

Wie in den Abbildungen des Kapitels 5.2.3 zu sehen ist, sind die Property Editoren in ihrer Ansicht zweispaltige und mehrzeilige Listen, wobei ein Listeneintrag auch hierarchische Strukturen zulässt, z.B. bei Knotenmengen. Die Klasse `QListView` der eingesetzten Qt3-Bibiliothek bietet hierfür eine grundlegende Funktionalität. Mit ihr können solche Tabellen mit beliebiger Spalten- und Zeilenanzahl effizient verwirklicht werden. Die Klasse `QListViewItem` (i.F. sind Objekte dieser oder abgeleiteter Klassen einfach Items) repräsentiert einen Eintrag in einer `QListView`.

Für die gewünschte Funktionalität der Constraint Property Editoren reichten diese Basisklassen jedoch nicht aus, da `QListView` im Wesentlichen für die Anzeige strukturierter Daten geeignet ist, nicht aber für direkte Editierfunktionalität einzelner Items, z.B. beim Anklicken in der Liste. Es war also eine Erweiterung beider Klassen notwendig.

`EditableListView` ist eine aus `QListView` abgeleitete abstrakte Klasse, die für die Aufnahme editierbarer Items der Klasse `EditableListViewItem` und besonders für die Ereignisbehandlung wie z.B. Mausklicks auf Items zuständig ist. Bei Benutzerereignissen fängt `EditableListView` diese ab und ruft entsprechende Methoden von `EditableListViewItem` auf, um eine Editierung zu starten, zu stoppen oder die Größe eines angezeigten Editors zu aktualisieren. Ferner erweitert diese Klasse `QListView` um eine alternierende Färbung von Items für eine leserlichere Darstellung.

Die abstrakte Klasse `EditableListViewItem` ist abgeleitet aus `QListViewItem`. Die Erweiterung dieser Klasse sorgt für die Verwaltung eines Editors (also ein Widget wie `QLineEdit` für Benutzereingaben) und die Behandlung einer Editieranforderung von `EditableListView` an eine beliebige Spalte. Sollte die Spalte editierbar sein, so wird der zuvor zugewiesene Editor in passender Größe angezeigt. Der Benutzer ist nun in der Lage die Editierung an Ort und Stelle vorzunehmen. Ist die Editierung beendet, so wird das Ergebnis vom Editor-Widget abgefragt und als Wert des Items gesetzt.

Subklassen von `EditableListViewItem` stehen ihrem Namen nach für Items mit entsprechenden Editoren: `LineEditItem` mit einem Editor vom Typ `QLineEdit`, `SpinBox-`

`EditItem` mit einem Editor vom Typ `QSpinBox` und `ComboBoxEditItem` mit einem Editor vom Typ `QComboBox`. Weitere Items mit anderen speziellen Editoren sind selbstverständlich durch Ableitung aus `EditableListViewItem` konstruierbar. Weiters kann einem `EditableListViewItem`-Objekt jederzeit ein neuer Editor zugewiesen werden. Die lose Kopplung eines Items und eines beliebigen Editors vom richtigen Typ (bei `LineEditItem` entsprechend ein beliebiges `QLineEdit`-Objekt) ermöglicht eine Änderung des Editors zur Laufzeit sowie eine Konfiguration des Editor-Widgets ohne das ihn verwaltende/anzeigende Item.

`AnchorConstraintEditableListView`, `AlConstraintEditableListView` und `SeqConstraintEditableListView` sind Spezialisierungen von `EditableListView`. Diese Klassen verwalten ihrem Namen nach jeweils ein entsprechendes Constraint-Objekt und sorgen für die Mapping-Funktionalität zwischen Darstellung und Repräsentation des eigentlichen Constraint-Objekts, d.h. Benutzerereignisse/Editierungen in der „*`ListView`“ aktualisieren das verwaltete „*Constraint-Objekt“ über entsprechende Methoden. Zudem implementieren diese Klassen weitere wichtige Funktionalitäten:

- Kontextmenüerzeugung und Ereignisbehandlung
- Kommunikation mit einem Graph-Dokument über Drag and Drop
- Eventuelle interne Drag and Drop Funktionalität
- Undo-/Redo-Operationen

In Abbildung 5.17 ist das Klassendiagramm der Constraint Property Editoren ohne (Member-)Variablen und Methoden zu sehen. Dort können die eben beschriebenen Zusammenhänge nochmals anschaulich nachvollzogen werden.

5.2.5 Bewertung und Aussicht

Die Kleingruppe Constraints erreichte die am Anfang der PG festgelegten Ziele. Im ersten Semster wurden Mechanismen für drei verschiedene Constraints und einen Layoutalgorithmus ins OGDF eingebunden. Die Constraints Anchor, Alignment und Sequence können in Anschlussarbeiten um beliebige andere Constraints, z.B. durch einen aus Kapitel 5.1.1, hinzugefügt werden. Der von der Gruppe implementierte Layoutalgorithmus mit Force-Directed-Ansatz ist für diese drei Constraints der sinnvollste. Doch wenn es weitere Constraints gibt, sind der Erweiterung um Layoutalgorithmen, die Constraints berücksichtigen, keine Grenzen gesetzt.
Auch die Einbindung in den Editor verlief nach Plan. Ein Constraint-Explorer verwaltet die vom Benutzer angelegten Constraints, die durch constraintspezifische Property-Editoren an die Wünsche des Benutzers angepasst werden können. Trotz der genauen Planung gab es Probleme, die erst während der Implementierung sichtbar wurden, die jedoch im Verlauf der Implementierungsphase besprochen und behoben werden konnten. Das Ergebnis ist eine benutzerfreundliche Verwaltung von Constraints und deren Anwendung im Layoutalgorithmus.

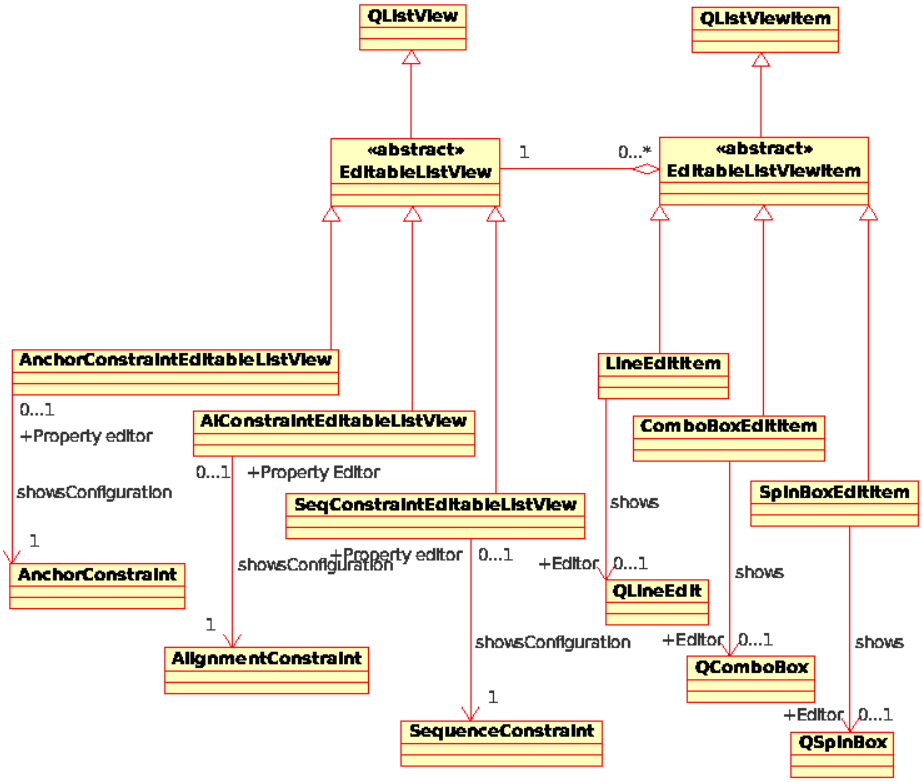

Abbildung 5.17: Klassendiagramm Constraint Property Editoren

6 Arbeitsgruppe Dateiformat

Die Arbeitsgruppe Dateiformat beschäftigte sich mit der Enwicklung einer Graphbeschreibungssprache, die insbesondere Compound-Graphen und Constraints unterstützen sollte. Hierbei konnte auch eine bereits existierende Sprache entsprechend der Zielvorstellung erweitert werden. Um das entwickelte Format im Rahmen des Open Graph Drawing Framework (OGDF) nutzbar zu machen, sollte das Projekt zudem um Funktionen erweitert werden, die das fehlerfreie Laden und Speichern von Dateien des zu entwickelnden Formats ermöglichen.

Das folgende Kapitel gibt einen Überblick über die Resultate der Arbeitsgruppe Dateiformat. Im ersten Abschnitt werden die existierenden Graphbeschreibungssprachen vor- und letztlich gegenübergestellt, während im darauf folgenden Abschnitt die neu entwickelte Graphbeschreibungssprache Open Graph Markup Language (OGML) beschrieben wird. Die Vorgehensweise bei der Implementierung der Serialisierungsmethoden wird im dritten Abschnitt dargestellt. Abschließend wird ein Ausblick zu der OGML und zum OGDF gegeben.

6.1 Graphbeschreibungssprachen

Die Arbeit der Gruppe begann damit, bereits existierende Graphbeschreibungssprachen zu sichten. Besonderes Augenmerk sollte dabei auf bestehende Features, Erweiterbarkeit in Hinblick auf fehlende Features und Lesbarkeit gelegt werden.

Die bestehenden Graphbeschreibungssprachen lassen sich in zwei Gruppen teilen, die der textbasierten und die der XML[1]-basierten Sprachen. Obwohl das *OGDF* momentan mit einem textbasierten Format arbeitet, wurden solche Formate im Rahmen der Sichtung vernachlässigt, da viele Vorteile für ein XML-basiertes Format sprechen:

- Gute Lesbarkeit/Transparenz
- Gute Erweiterbarkeit
- Gute Austauschbarkeit
- Hohe Akzeptanz/Verbreitung
- Verfügbarkeit freier, professioneller Parser
- Gute Transformierbarkeit mittels XSLT
- Einfache Syntax- und Semantikvalidierung mittels XML-Schema

[1] Extensible Markup Language (XML)

	GraphML	*GraphXML*	*GXL*	*XGMML*
(Un-)Gerichtete und gemischte Graphen	+	+	+	+
Hierarchische Graphen	+	+	+	+
Hypergraphen (und damit Hyperkanten)	+	-	+	-
Anwendungsspezifische Daten	+	+	+	+
Grafische Repräsentation	+	+	+	+
Constraints	+			
Referenzierung externer Daten	+	+	+	+

Tabelle 6.1: Features der Graphbeschreibungssprachen

6.1.1 Textbasierte Sprachen

Aus Gründen der Vollständigkeit seien hier zwei textbasierte Formate erwähnt. Die verbreitetste Sprache ist die *Graph Modeling Language (GML)*. Zur Zeit nutzt das *OGDF* eine Erweiterung dieses Formats zur Serialisierung von Graphen. Zudem gibt es die *Graph Description Language (GDL)*.

6.1.2 XML-basierte Sprachen

Die folgenden XML-Dialekte wurden einer genaueren Betrachtung unterzogen:

- *Graph Markup Language (GraphML)*
- *Graph Extensible Markup Language (GraphXML)*
- *Graph Exchange Language (GXL)*
- *Extensible Graph Markup and Modeling Language (XGMML)*

Tabelle 6.1 bietet einen Überblick über die Features der XML-basierten Sprachen.

Graph Markup Language (GraphML)

Die *Graph Markup Language*, kurz GraphML, ist eine im Jahre 2001 von der Graph Drawing Community [11] vorgestellte XML-basierte Graphbeschreibungssprache.

Prinzipiell stellt die GraphML Elemente zur strukturellen Beschreibung eines Graphen bereit und bietet darüber hinaus flexible Erweiterungsmöglichkeiten zur Speicherung beliebiger anwendungsspezifischer Daten wie z.B. Layout-Informationen. Zudem wird ein Mechanismus zur Referenzierung externer Daten geboten.

Die von der GraphML unterstützten Graphklassen sind ungerichtete, gerichtete und gemischte Graphen, Hypergraphen und hierarchische Graphen wie Cluster- und Compound-Graphen.

In puncto Erweiterung bietet die GraphML zwei Möglichkeiten. Die so genannten GraphML-Attribute erlauben das Binden beliebiger Daten an die Elemente zur strukturellen Beschreibung des Graphen, also Knoten, Kanten und den Graph selbst. Zudem

können die aufgezählten Strukturelemente um neue (XML-)Attribute erweitert werden. Zu diesem Zwecke muss das XML-Schema an dafür vorgesehenen Stellen redefiniert werden. Dies führt jedoch dazu, dass die Lesbarkeit der Instanzen durch andere GraphML-Parser nicht mehr gewährleistet ist.

Seit dem Erscheinen von GraphML wurden Erweiterungen veröffentlicht, die es erlauben anwendungsspezifische Daten zu typisieren und Informationen für optimierte Parser zu speichern. In Zukunft soll eine Erweiterung zur Speicherung abstrakter Layout-Informationen folgen.

Zusammenfassend bringt der universelle Ansatz der GraphML wie beschrieben viele Möglichkeiten mit sich. Durch die Erweiterungsmechanismen können beliebige Daten gspeichert werden. Nachteilig zu bewerten ist das Fehlen von Standards im Rahmen des Speicherns von Layout-Informationen und Constraints. Da diese als anwendungsspezifische Daten betrachtet werden, sind tendenziell nur die Strukturinformationen eines Graphen unter verschiedenen Anwendungen austauschbar. Durch den exzessiven Gebrauch von GraphML-Attributen leidet zudem die Lesbarkeit der Instanzen.

Graph Extensible Markup Language (GraphXML)

GraphXML ist ein XML-basiertes Dateiformat zur Speicherung von Graphen und wurde erstmalig auf dem Graph Drawing Symposium 2000 vorgestellt, um den Bemühungen der Graph Drawing Community [11] nach einer standardisierten Graphbeschreibungssprache Rechnung zu tragen, die einen einfachen Austausch von Graphdaten zwischen Anwendungen ermöglichen soll. Dabei sei aber angemerkt, dass es sich hierbei nicht um ein, von allen Teilnehmern erarbeitetes Format handelt, sondern dass es bereits spätestens 1999 von Ivan Herman und M. Scrott Marshall am CWI (National Research Institute for Mathematics and Computer Science, Amsterdam) entwickelt worden war.

Mit der Grundmenge von Sprachelementen, bestehend aus *graph*, *node* und *edge*, lassen sich ungerichtete und gerichtete Graphen auf einfache Weise beschreiben. Zusätzlich stehen noch die Elemente *label*, *data* und *dataref* zur Verfügung, um einen Graphen mit semantischen und teils benutzerspezifischen bzw. externen Informationen anzureichern, da diese Tags unter fast jedem der Strukturelemente als Kind einbindbar sind.

Der inhaltlich größte Vorteil von *GraphXML* besteht in der Fähigkeit hierarchische Graphen zu beschreiben, da *graph*-Blöcke auf derselben Ebene des Elementbaums Geschwister von sich haben und *node*s wiederum durch die Zuhilfename der Attribute *isMetanode* und *xlink:href* einen externen *graph*-Block referenzieren können. Die Einsatzmöglichkeiten, die sich daraus ergeben, sind aber limitiert, da das für Knoten standardmäßig anzugebende *name*-Attribut, durch das sich die Endpunkte von Kanten eindeutig identifizieren lassen sollen, nur innerhalb seines *graph*-Astes eindeutig sein muss.

Ein weiterer, der Visualisierung dienende Satz von Elementen steht zu Verfügung, um einen strukturell beschriebenen Graphen sowohl geometrisch als auch farblich auszuzeichnen. Möchte ein Benutzer einen gesamten Graphen oder Knoten geometrisch ausrichten, so bedient er sich des *size* elements, um die generelle Breite und Höhe des zu visualisierenden Objekts zu beschreiben. Desweiteren lassen sich Knoten und Kanten mittels des *position* Elements Koordinaten bzw. Haltepunkte zuweisen, da dieses Element dank

seiner Attribute für x, y und z Koordinaten auch für dreidimensionale Graphdarstellungen gerüstet ist. Eine farbliche Auszeichnung geschieht mittels eines *style*-Blocks, der Kind eines jeden Strukturelements sein darf und in dem mittels *line* und *fill* die farbliche Formatierung vorgenommen werden kann.

Eine Besonderheit, die *GraphXML* gegenüber verwandten Graphbeschreibungssprachen auszeichnet, ist die Möglichkeit, vergangene Bearbeitungsschritte dateiintern abzuspeichern. Diese Vorkehrung erlaubt es Benutzern, auf bequeme Art und Weise eine Bearbeitungsabfolge zurückzuverfolgen und weiterzugeben. Je nach Interpretation durch die Applikation können solche Bearbeitungsschritte auch zu Animationszwecken genutzt werden. Da diese Besonderheit aber nicht Kerngebiet einer Graphbeschreibungssprache ist, soll dies an dieser Stelle nicht weiter ausgeführt werden.

Vorteile

- Alle zu realisierende Graphklassen werden unterstützt
- Sehr gute Lesbarkeit
- Gute Grundlage für Geometrie- und Visualisierungsangaben
- Gute Grundlage für Animationsspeicherung
- Referenzierung von extern gespeicherten Subgraphen
- Sinnvolle Referenzierung von fremden Dateien mittles MIME Type
- Strikte XML Syntax
- Semantisch leicht erweiterbar mittels DTD

Nachteile

- Keine generelle Unterstützung von Constraints
- Kein flexibler Erweiterungsmechanismus
- Keine Trennung von Struktur- und Layout-Informationen
- Unsinnige Attribute, wie zur Planarität und Azyklik
- Baut auf dem veralteten System Doctype Definition auf, nicht auf XML Schemas
- Keine Hypergraphen

Graph Exchange Language (GXL)

Die *Graph Exchange Language*, kurz GXL, ist eine Graphbeschreibungssprache im Bereich der Softwareentwicklung, die insbesondere die Interoperabilität zwischen unterschiedlichen Software-Reengineering-Tools ermöglicht. Dennoch hat GXL den Anspruch, ein Standardformat für graphbasierte Anwendungen zu sein, und ist als eine XML-Untersprache konzipiert. GXL wurde seit 1998 an der Universität Koblenz-Landau entwickelt und ist in der Version 1.0 (Januar 2001) und der Version 1.1 alpha (Dezember 2002) verfügbar.

Die Besonderheit der GXL ist eine indirekte Beschreibung von Graphen und Graphklassen mittels sog. TGraphs. TGraphs sind getypte, attributierte, gerichtete und geordnete Graphen und lassen sich mit Hilfe der vordefinierten Sprachkonstrukte des GXL-Metaschemas in ein GXL-Dokument überführen. Durch die Modellierung von Graphen und Graphklassen über TGraphs wird die Austauschbarkeit dieser über denselben Mechanismus ermöglicht. Ein weiterer Vorzug der GXL ist ihre Unterstützung aller grundlegend gebrauchten Graphklassen wie ungerichtete, gerichtete und gemischte Graphen oder auch attributierte Graphen. Weiters werden hierarchische Graphen und damit auch Compound-Graphen unterstützt.

Nachteile der GXL bezüglich der Anforderungen an das Dateiformat des OGDF sind ein fehlender Mechanismus zur Formulierung von Constraints, eine umständliche Transformation von Graph-Schemata und Graph-Instanzen in TGraphs und daraus resultierend Overhead und eingeschränkte Lesbarkeit. Die GXL konnte wegen dieser Unzulänglichkeiten in den Kernanforderungen nur schwer als eine für das OGDF geeignete Sprache oder Orientierung einer eigenen Sprache dienen. Letztendlich kommt kein Mechanismus der GXL in der OGML (siehe Kapitel 6.2) zum Tragen.

Extensible Graph Markup and Modeling Language (XGMML)

Die XGMML (Extensible Graph Markup and Modeling Language) ist eine XML-basierte Graphbeschreibungssprache, welche die GML (Graph Modelling Language) als Vorbild nutzt und in ein XML-Format überträgt. Die Struktur der Sprache und die Schlüsselwörter wurden übernommen und somit ist eine Konvertierung eines GML-Dokumentes in ein XGMML-Dokument auf einfache Art und Weise durchführbar.

Die wesentlichen Elemente, um einen Graphen und dessen Layout in der XGMML zu beschreiben, sind *graph*, *node*, *edge*, *label* und *att*. Das *graph*-tag ist das Elter-Element für alle anderen Elemente und enthält die Definition eines Graphen mit Knoten, Kanten und Labels. Hierbei werden für jedes definierte Graphelement neben den strukturellen Informationen gleichzeitig die Layout-Daten angegeben.

Die XGMML kann implizit hierarchische Graphen wie Cluster- und Compound-Graphen beschreiben. Diese Funktion wird über das *att*-Element bereit gestellt. Um komplexere Strukturen darzustellen, wird dieses *Attachment*[2]-tag als Kind eines Graphelements eingesetzt. Um einen Untergraphen zu definieren, wird ein *graph*-Element in ein *att* geschachtelt, während dieses wiederum Kind eines Knotens wird. Das *att*-Element wird

[2]engl. attachment = Anhang

außerdem verwendet, um applikationspezifische Daten zu integrieren.

Die Graphbeschreibungssprache XGMML unterstützt darüber hinaus die Modellierung unterschiedlicher Graphtypen. Es ist möglich, gerichtete, ungerichtete und gemischte Graphen darzustellen. Diese Eigenschaft der Kanten wird durch das Attribut *directed* des *graph*-tags gesteuert. Allerdings wird die Modellierung von Hyperkanten durch die XGMML nicht zur Verfügung gestellt.

Ein weiterer fehlender Mechanismus dieser Sprache ist die Beschreibung von Constraints. Dies ist umso bedeutender, da es kein flexibles Verfahren für Erweiterungen gibt.

6.2 Open Graph Markup Language (OGML)

6.2.1 Anforderungen

Die im Vorfeld analysierten Graphbeschreibungssprachen lösten verschiedene Anforderungen auf zum Teil unterschiedlichen Wegen und besaßen somit verschiedene Stärken und Schwächen. Trotzdem erfüllte keiner der existierenden XML-Dialekte alle von uns gewünschten Grundeigenschaften oder löste beschreibungstechnische Probleme auf benutzerfreundliche Art und Weise. Somit wurde von der Arbeitsgruppe Dateiformat die *Open Graph Markup Language*, kurz OGML, entwickelt, bei der es sich ebenfalls um eine XML-basierte Graphbeschreibungssprache zur strukturierten Speicherung von Graphen und Diagrammen handelt. Die *OGML* ist somit ein Teilprojekt der Projektgruppe PG478. Die neue Sprache sollte folgenden Anforderungen genügen:

- Beschreibung von gerichteten, ungerichteten und gemischten Graphen
- Beschreibung von Multigraphen (Graphen mit Mehrfach- bzw. Multikanten)
- Beschreibung von hierarchischen Graphen (Cluster- und Compound-Graphen)
- Beschreibung von Hyperkanten und somit Hypergraphen
- Einbettung von Constraints
- Angabe von Layoutinformationen zu Visualisierungszwecken, inklusive geometrische Informationen und Farbformatierungen

6.2.2 Sprachfeatures

Mittels der OGML können Struktur- und Layout-Informationen verschiedenster Graphtypen gespeichert werden. Dabei unterscheidet sich die OGML von existierenden Sprachen bezüglich der Möglichkeit der Speicherung von Constraints und Compound-Graphen. Neben diesen Sprachkonstrukten lag das Hauptaugenmerk beim Design der OGML auf Konsistenz, Lesbarkeit und einer strikten Trennung von Struktur- und Layoutinformationen.

Folgende Aufzählung verschafft einen Überblick über die von der OGML gebotenen Features:

- Strikte Trennung von Struktur- und Layoutinformationen
- Unterstützung verschiedenster Graphklassen
  - Gerichtete, ungerichtete und gemischte Graphen
  - Multigraphen (Graphen mit Mehrfach- bzw. Multikanten)
  - Hierarchische Graphen (Cluster- und Compound-Graphen)
  - Hypergraphen (Graphen mit Hyperkanten)
- Möglichkeit der Einbettung anwendungsspezifischer Daten
- Toolkit zur anwendungsspezifischen Speicherung von Constraints
- Layoutinformationen zu Visualisierungszwecken
  - Formatierung von Strukturelementen mittels Styles und Style-Vorlagen
  - Vererbungsmechanismus im Rahmen von Style-Vorlagen
- Möglichkeit der Referenzierung externer Daten

6.2.3 Sprachaufbau

Eine OGML-Datei beherbergt aus Gründen der Lesbarkeit und einer klaren Trennung von Daten unterschiedlicher Graphen/Diagramme nur eine OGML-Instanz. Eine OGML-Instanz beginnt als eine XML-Instanz mit einem XML-Header. Daran schließt sich dann der allumfassende *ogml*-Container an, welcher seinerseits den *graph*-Container inkludiert. Der Inhalt des *graph*-Containers könnte auch direkt in den *ogml*-Container eingebettet werden. Allerdings erlaubt die Modellierung so, benutzerspezifische Daten (siehe hierzu auch 6.2.4) mittels eines *data*-Blocks einem Graphen bzw. Diagramms eindeutig zuzuordnen, indem der *graph*-Container und der *data*-Block dieselbe Rekursionstiefe besitzen.

Der *graph*-Container ist das Herzstück einer OGML-Instanz und lässt sich auf höchster Ebene in zwei große Blöcke unterteilen. Ein Block (i.F. Strukturblock) für Strukturinformationen und ein Block (i.F. Layoutblock) für Layoutinformationen (darstellungsbezogene Informationen) eines Graphen oder Diagramms. Man entschied sich bewußt zu dieser strikten Trennung, um höchste Lesbarkeit zu unterstützen und den Parsevorgang einer Instanz auch bei fehlerhaften oder fehlenden Layoutinformationen zu ermöglichen. Zudem lässt es die OGML so zu, für bestimmte Zwecke reine Strukturdaten zu speichern. Ein schematischer Überblick über den Aufbau einer OGML-Instanz wird durch das kommentierte Beispiel 6.1 deutlich.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!-- Wurzelelement -->
3 <ogml>
4   <!-- Evtl. anwendungspezifische Daten des Graphen/Diagramms -->
5   <data>
6     ...
7   </data>
```

```
8   <!-- Beschreibung des Graphen/Diagramms -->
9   <graph>
10    ...
11    <!-- Strukturinformationen -->
12    <structure>
13      <node />
14      ...
15      <edge />
16      ...
17    </structure>
18    <!-- Layoutinformationen -->
19    <layout>
20      <!-- Überschreibbare Style-Templates -->
21      <styleTemplates>
22        ...
23      </styleTemplates>
24      <!-- Individuelle Formatierung der Strukturelemente -->
25      <styles>
26        ...
27      </styles>
28      <!-- Constraints -->
29      <constraints>
30        ...
31      </constraints>
32    </layout>
33  </graph>
34 </ogml>
```

Listing 6.1: Aufbau einer OGML-Datei

Strukturblock

Der Strukturblock inkludiert beliebig viele *node*-, *edge*- und *label*-Elemente. Selbsterklärend dürfte die Bedeutung dieser Elemente aufgrund der Bezeichner sein. Knoten werden durch *node*-Elemente, Kanten und auch Hyperkanten durch *edge*-Elemente und textuelle Daten durch *label*-Elemente repräsentiert.

Elemente des Typs *node* können weitere *node*-Elemente einbetten. Dadurch wird die Speicherung hierarchischer Graphen, wie Cluster- und Compound-Graphen, realisiert. Label können Subelemente von Knoten, Kanten sowie des *graph*-Elements und der für (Hyper-) Kanten notwendigen *source*- und *target*-Elemente (genauere Beschreibung von Kanten siehe Kapitel 6.2.4) sein. Optional im Struktur- sowie Layoutblock ist die Verwendung von anwendungsspezifischen Daten durch *data*-Elemente in allen Elementen (siehe Kapitel 6.2.4). Beispiel 6.2 veranschaulicht eine mögliche Verwendung von *node*-Elementen mit *label*- und *data*-Elementen.

```
1 <!-- Einfacher Knoten -->
2 <node id="n01">
3   <!-- Knotenlabel mit anwendungsspezifischen Daten-->
4   <label>
5     <data name="Verweis">
6       <string name="Hyperlink">
```

```
7         "http://www.ogml.de"
8       </string>
9     </data>
10    ...
11    <content>
12      "Dies ist Knoten n01
13      und sein Label geht über mehrere Zeilen"
14    </content>
15  </label>
16  <data>
17    ...
18  </data>
19  ...
20 </node>
21 <!-- Compound Knoten -->
22 <node id="n02">
23   <label>
24     <content>
25       "Knoten n02 ist ein Elter(Compound)-Knoten"
26     </content>
27   </label>
28   <node id="n03"/>
29   <data>
30     ...
31   </data>
32   <node id="n04>
33     <node id="n05"/>
34   </node>
35 </node>
```

Listing 6.2: Knotenspeicherung in OGML

Layoutblock

Im Layout-Block existieren drei große Subblöcke, ein *styleTemplates*-Subblock, ein *styles*-Subblock und ein *constraints*-Subblock.

Der *styleTemplates*-Subblock kapselt Definitionen von Stil-Vorlagen für Knoten (*nodeStyleTemplate*-Element), Kanten (*edgeStyleTemplate*-Element) und Label (*labelStyleTemplate*-Element). Solche Stil-Vorlagen erlauben es dem Benutzer komfortabel, Informationen zur Mehrfachverwendung einmalig zu definieren, wie z.B. die Farbe von Kanten, die Gestalt von Knoten oder auch die Schriftart.

Im *styles*-Subblock werden diese Stil-Vorlagen dann als Standard an Strukturelemente gebunden oder individuell von Strukturelementen referenziert. Wird für ein Strukturelement eine Referenzierung vorgenommen ist es sogar möglich, Definitionen der Vorlage teilweise zu überschreiben.

Der *constraints*-Subblock wird zur Speicherung von Constraints verwendet. Dabei wurde dieser Block in den Layout-Block aufgenommen, da sich Constraints im engeren Sinn auf das Aussehen von Graphen oder Diagrammen beziehen. Constraints werden dabei mit einem Toolkit von vordefinierten Elementen formuliert. Für eine detailliertere Ausführung der Subblöcke wird auf die folgenden Kapitel verwiesen.

6.2.4 Spezielle Sprachfeatures

Kanten und Hyperkanten

Ein grundlegendes Element von Graphen sind Kanten und deren Obermenge, die Hyperkanten. Die Open Graph Markup Language bietet für beide Graphelemente eine einheitliche Syntax, um diese strukturell zu definieren und für das Layout anzupassen.

Die Definition einer Kante bzw. Hyperkante erfolgt im *structure*-Teil eines Dokuments. Visuelle Informationen sind im *layout*-Block zu finden.

Eingeleitet wird die Kanten-Definition durch das *edge-tag*. Wie jedes OGML-Element besitzt eine Kante eine eindeutige *id*. Ausgezeichnet wird eine Kante durch ihre Quellen und Ziele. Diese werden nach den Schlüsselwörtern *source* bzw. *target* über ihre Identifikatoren referenziert. Dabei ist neben Knoten auch die Angabe von Kanten-Ids möglich so dass Kanten modelliert werden können, die auf Kanten zeigen.

In der OGML existiert kein *tag*, welches eine Kante als Hyperkante auszeichnet. Die Differenzierung, welchen Typ die modellierte Kante hat, erfolgt implizit über die Anzahl der Quell- und Ziel-Knoten. Eine Hyperkante liegt vor, falls nur ein Quell- oder Ziel-Knoten referenziert wird, oder im Falle von mehr als zwei Quellen oder/und Zielen.

Darüber hinaus wird durch die Formulierung festgelegt, ob eine gerichtete, ungerichtete oder bidirektionale Kante vorliegt. Eine ungerichtete Kante wird durch die Referenzierung von zwei oder mehreren Quell-Knoten kodiert, zwei Ziel-Knoten implizieren eine bidirektionale Kante und eine gerichtete Kante wird durch die Angabe genau eines Quell- und eines Ziel-Knotens realisiert. Listing 6.3 veranschaulicht die Definition einer gerichteten Kante mit der id “e01“, welche zwischen Knoten “n01“ und “n02“ verläuft, während die Kante mit der id “e02“ eine ungerichtete Hyperkante zwischen den drei Knoten „n01“, „n02“ und „n03“ darstellt.

```
1 <edge id="e01">
2   <source idRef="n01" />
3   <target idRef="n02" />
4 </edge>
5 <edge id="e02">
6   <source idRef="n01" />´
7   <source idRef="n02" />
8   <source idRef="n03" />
9 </edge>
```

Listing 6.3: Definition einer gerichteten Kante und einer ungerichteten Hyperkante

Neben der strukturellen Definition einer Kante besteht im *layout*-Block die Möglichkeit jede Kante individuell zu formatieren. Das Schlüsselwort *edgeStyle* mit der zugehörigen *id*-Referenzierung umfasst die Beschreibung einer Kante.

Zu den Elementen bzw. Attributen, die gespeichert werden können, zählen die Informationen zur Gestaltung der Linie, der Kantenendendekoration und der Segmente, welche durch Pfadpunkte festgelegt werden. Diese Unterteilung der Kanten in Segmente offeriert eine uneingeschränkte Modellierung, insbesondere erhält das Layout von Hyperkanten dadurch einen simplen Beschreibungsmechanismus. In Listing 6.4 erhält man einen Einblick in ein Beispiel für eine Layout-Definition einer Kante. Es wird die einfache

Kante mit der id “e01“ beschrieben, welche den Linienstil *“dotted“* hat. Als Kantenendendekoration werden für Quellen der Typ *“diamond“* und für Ziele *“arrow“* gesetzt. Die Koordinaten der Kantenknicke werden in den Punkten “p01“ bis “p09“ gespeichert. Über die Segmente ist eine Überschreibung des Linienstils und der Dekoration der Kantenenden möglich. Dieser Mechanismus wird in diesem Beispiel jedoch nicht verwendet.

```
<edgeStyle idRef="e01">
  <line type="dotted" width="1" color="#FF0000" />
  <sourceStyle type="diamond" color="#FF0000" size="10" />
  <targetStyle type="arrow" color="#00FF00" size="10" />
  <point id="p01" x="450" y="20" />
  ...
  <point id="p09" x="520" y="30" />
  <segment>
    <endpoint idRef="s01" />
    <endpoint idRef="p01" />
  </segment>
  ...
  <segment>
    <endpoint idRef="p01" />
    <endpoint idRef="t01" />
  </segment>
</edgeStyle>
```

Listing 6.4: Layout-Definition einer Kante

Die Entwicklung der Kanten- und Hyperkanten-Modellierung in der OGML durchlief mehrere Phasen. Es stand nicht von Anfang an fest, wie eine Differenzierung zwischen den unterschiedlichen Graphtypen stattfinden soll. Probleme bereiteten dabei vor allem Hyperkanten und die Unterscheidung von ungerichteten und bidirektionalen Kanten. Ein separates *hyperedge-tag* wurde nach reiflicher Überlegung ausgeschlossen, da so die OGML überschaubarer gehalten wird. Es existiert nur ein Element für Kanten, über welches alle Kanten-Ausprägungen eindeutig modellierbar sind. Für die Kantenrichtung – gerichtet, ungerichtet oder bidirektional – wurde ebenfalls keine gesonderten *tags* eingeführt. Die Modellierung über die Kombinationen von *targets* und *sources* ermöglicht eine einfachere und übersichtliche Beschreibung.

Die Einteilung der Kanten in Segmente war vorerst nur für Hyperkanten vorgesehen. Normale Kanten sollten durch die Angabe der Koordinaten von Pfadpunkten ihren visuellen Verlauf erhalten. Durch die Zusammenlegung der beiden Kantentypen zur *edge* wurde diese Struktur auch für einfache Kanten übernommen. Diese Schreibweise ist allerdings ausführlicher und auf den ersten Blick unübersichtlicher. Auf der anderen Seite ermöglicht diese Semantik individuelle Formatierungen der einzelnen Segmente, der Segmentendendekorationen, und die Konsistenz der Sprache bleibt erhalten.

Constraints

Beim Design der OGML entpuppte sich die Anforderung eines flexiblen Mechanismus zur Speicherung von sowohl im Rahmen der Projektgruppe zu realisierenden als auch bislang unspezifizierten Constraints als eine Hauptschwierigkeit.

Mit dem Ziel einer klar strukturierten und höchst lesbaren OGML beabsichtigten wir zunächst, die im Rahmen der Projektgruppe zu realisierenden Constraints mit individuellen XML-Elementen zu formulieren und ihre Definition fest im XML-Schema der OGML (i.F. OGML-Schema) zu verankern. Die Arbeitsgruppe sah darin nicht nur einen hohen Grad an Strukturiertheit und Lesbarkeit der OGML als vorteilhaft, sondern auch die Möglichkeit der Validierung von Constraints innerhalb von OGML-Instanzen gegenüber dem OGML-Schema, auch für andere Anwendungen außerhalb von GDE. Folgen dieses Konzeptes wären allerdings eine große Abhängigkeit von Constraints gegenüber der OGML, häufige Redefinitionen des OGML-Schemas bei Veränderungen von unterstützten und neu hinzuzufügenden Constraints und dadurch eine geringe Flexibilität gewesen.

Deshalb entschieden sich die Entwickler für ein Toolkit von Elementen, ähnlich dem Typisierungskonzept höherer Programmiersprachen. Diese Idee ist in der Erkenntnis begründet, dass Constraints als Objekte aus einfachen und zusammengesetzten Daten aufgebaut sind. Deshalb besteht das Toolkit aus atomaren Elementen zur Speicherung von einfachen/atomaren Datentypen und einem komplexen Element zur Speicherung individueller zusammengesetzter Daten, welches aus atomaren und wieder komplexen Elementen zusammengesetzt sein kann:

1. Atomare Elemente:
   - *int*-Element für vorzeichenbehaftete Ganzzahlen
   - *num*-Element für vorzeichenbehaftete Fließkommazahlen
   - *bool*-Element für boolesche Werte
   - *string*-Element für Zeichen/-ketten
   - *nodeRef*-Element für Knotenreferenzierungen
   - *edgeRef*-Element für Kantenreferenzierungen
   - *labelRef*-Element für Labelreferenzierungen
2. Komplexes Element: *composed*-Element für zusammengesetzte Daten aus atomaren Elementen oder wiederum vom Typ *composed*

Durch das Toolkit und insbesondere über das *composed*-Element lassen sich sehr einfach Objekte und Objektmengen in OGML ausdrücken. Es ist sogar möglich, hierarchische Strukturen, wie z.B. Aggregationsbeziehungen zwischen Objekten, zu speichern. Beispiel 6.5 veranschaulicht die Speicherung eines Alignment-Constraints für eine Menge von Knoten.

```
1 <constraint id="c03" name="Alignment1" type="Alignment">
2   <!--Steigung der Geraden -->
3   <num name="angle" value="45" />
4   <!--Knoten auf dieser Geraden -->
5   <composed name="NodeSet">
6     <nodeRef idRef="n01"/>
7     ...
8     <nodeRef idRef="n23"/>
```

```
9   </composed>
10 </constraint>
```

Listing 6.5: Speicherung Alignment-Constraint

Das Konzept des Toolkits wird somit den Anforderungen Erweiterbarkeit und Flexibilität bezüglich Constraints gerecht. Die Formulierung von Constraints und deren Auswertung bleibt anwendungsspezifisch und somit unabhängig.

Hierarchische Graphen

Die Beschreibung von hierarchischen Graphen, wie z.B. *Compound-* oder *Cluster-Graphen*, ist durch die flexible Einsatzmöglichkeit der *node-* und *edge*-Elemente relativ einfach und intuitiv. Möchte man einen Compound darstellen, bedient man sich des Prinzips, dass Knoten strukturell Eltern weiterer Knoten sein können, so dass man unterhalb eines node-Elements weitere node-Elemente hinzufügen kann. Besitzt also ein Knoten strukturbeschreibende Knoten-Elemente als Kinder, so ist dieser entweder als Compound oder Cluster zu interpretieren.

```
1  ...
2  <node id="n01">
3    <label>
4      <content>
5        "Compound 1"
6      </content>
7    </label>
8    <node id="n01_01">
9      <label>
10       <content>
11         Knoten 1 Compound 1
12       </content>
13     </label>
14   </node>
15   <node id="n01_02">
16     <label>Knoten 2 Compound 1</label>
17   </node>
18 </node>
19 <node id="n02">
20   <label>Compound 2</label>
21   <node id="n02_01">
22     <label>Knoten 1 Compound 2</label>
23   </node>
24   <node id="n02_02">
25     <label>Knoten 2 Compound 2</label>
26   </node>
27 </node>
28 ...
```

Listing 6.6: Aufbau einer Knotenhierarchie

Da per Spezifikation alle *id*-Attribute dateiweit eindeutig sein müssen, ist auch die Zuweisung von Kantenenden zu Knoten oder ganzen Compounds stets eindeutig.

```
...
<node id="n01">
  <label>
    <content>
      Compound 1
    </content>
  </label>
  <node id="n01_01">
    <label>
      <content>
        Compound 1 Knoten 1
      </content>
    </label>
  </node>
  <node id="n01_02">
    <label>
      <content>
        Compound 1 Knoten 2
      </content>
    </label>
  </node>
</node>
<node id="n02">
  <label>
    <content>
      Compound 2
    </content>
  </label>
  <node id="n02_01">
    <label>
      <content>
        Compound 2 Knoten 1
      </content>
    </label>
  </node>
  <node id="n02_02">
    <label>
      <content>
        Compound 2 Knoten 2
      </content>
    </label>
  </node>
</node>
...
<edge id="e01">
  <source idRef="n01" />     <!-- Compound 1 -->
  <target idRef="n01_02" />  <!-- Compound 1 Knoten 2 -->
</edge>
<edge id="e02">
  <source idRef="n01_01" />  <!-- Compound 1 Knoten 1 -->
  <target idRef="n02_01" />  <!-- Compound 2 Knoten 1 -->
</edge>
```

Listing 6.7: Zuweisung der Knotenendpunkte

Styles und Style-Vorlagen

Zur strukturierten Speicherung darstellungsbezogener Informationen sind im Rahmen der OGML Style-Vorlagen, also Style-Informationen für Mengen von Strukturelementen gleichen Typs, und Style-Informationen für konkrete Strukturelemente zu unterscheiden.

Prinzipiell können Layout-Informationen für alle Strukturelemente, also Knoten, Kanten und Labels definiert werden.

Style-Vorlagen bieten wie angedeutet die Möglichkeit, bestimmte (selbst gebildete) Klassen von Elementen zentral zu formatieren. Die Formatierung könnte dabei z.B. entsprechend der Semantik der Elemente erfolgen.

Zu unterscheiden sind Style-Vorlagen für Knoten, Kanten und Labels. Mit Style-Vorlagen können also beliebig viele Elemente des entsprechenden Typs formatiert werden. Dazu können die Style-Vorlagen innerhalb der Style-Angaben zu den einzelnen Strukturelementen referenziert werden. Ein besonderes Merkmal stellt die Vererbung von Style-Vorlagen dar. Jede Style-Vorlage kann optional von einer anderen Style-Vorlage gleichen Typs alle Layout-Informationen erben. Die vererbten Informationen können in der erbenden Vorlage überschrieben werden. In Listing 6.8 ist beispielhaft eine Style-Vorlage für Knoten dargestellt. Das optionale *idRef*-Attribut enthält den Identifikator der Style-Vorlage für Knoten, von der geerbt werden soll.

```
1 <nodeStyleTemplate id="nSpecial" idRef="nDefault">
2   <shape type="rect" width="100" height="100"/>
3   <fill color="#ff22aa" pattern="solid" patternColor="#222222"/>
4   <line type="dashed" width="2" color="#000022"/>
5 </nodeStyleTemplate>
```

Listing 6.8: Definition einer Style-Vorlage für Knoten

Darüber hinaus kann jedes Strukturelement individuell formatiert werden. Auch die Kombination aus Referenzierung einer Style-Vorlage und (zusätzlicher) individueller Formatierung ist erlaubt. In Listing 6.9 ist dies anhand der Formatierung eines Labels veranschaulicht. Über das optionale *template*-Element können dabei die Layout-Information einer beliebigen Style-Vorlage entsprechenden Typs geerbt werden. Das *location*-Element dient der Positionierung des Labels.

```
1 <labelStyle idRef="l01">
2   <template idRef="lSpecial"/>
3   <location x="340" y="170"/>
4   <text alignment="center" decoration="underline" transform="lowercase"
        rotation="90"/>
5   <font family="serif" style="italic" variant="small-caps" weight="bolder"
        stretch="wider" size="10" color="#AA2233"/>
6 </labelStyle>
```

Listing 6.9: Individuelle Formatierung eines Labels

Style-Vorlagen für Knoten erlauben es die Gestalt der Knoten zu definieren. Es können Angaben zu Form, Abmessungen, Füllung und Rahmenlinie gemacht werden. Im Rahmen der individuellen Formatierung von Knoten können neben der Position des Knotens auch Verbindungspunkte für ein- und ausgehende Kanten definiert werden.

Style-Vorlagen für Kanten ermöglichen die Formatierung des Linientyps, der Linienbreite und Linienfarbe sowie der Pfeilspitzen an den Start- und Zielknoten. Im Rahmen der individuellen Formatierung von Kanten können darüber hinaus Wegpunkte definiert werden. Mit Hilfe dieser Wegpunkte können in Folge die einzelnen Kantensegmente bzw. -abschnitte definiert werden. Kantensegmente sind Verbindungen zwischen jeweils zwei Wegpunkten, zwischen zwei Endpunkten oder zwischen einem Weg- und einem Endpunkt. Dabei können Linienformat und Kantenendendekorationen der definierten Segmente individuell angepasst werden. Dieses Konzept ermöglicht es, Kanten beliebigen Aussehens zu definieren. So könnten z.B. Hyperkanten in einer Venn-Diagramm ähnlichen Darstellung als Kreise um die entsprechende Knotenmenge gezeichnet werden. In Listing 6.4 ist die Formatierung einer Kante dargestellt.

Style-Vorlagen für Labels bieten die Möglichkeit, Angaben zu Ausrichtung, Dekoration und Rotation des Label-Texts zu treffen. Zudem kann die Schrift des Labels formatiert werden. Hierbei können Angaben zu Schriftart, -größe, -farbe, -gewicht etc. gemacht werden. Bei der individuellen Formatierung eines Labels ist es außerdem möglich, das Label absolut zu positionieren.

Anwendungsspezifische Daten

Zur Speicherung anwendungs- und benutzerspezifischer Daten in OGML dienen *data*-Elemente. Diese kapseln analog zu den *constraint*-Elementen die atomaren Elemente vom Typ *int*, *num*, *bool* und *string* (siehe hierzu auch 6.2.4) und können wie *composed*-Elemente wiederum *data*-Elemente enthalten. Dadurch können anwendungsspezifische Daten von hierarchischer Struktur formuliert werden. Das *string*-Element nimmt hierbei eine gewichtige Rolle ein. Alle Daten, welche sich nicht durch die genannten atomaren Elemente beschreiben lassen, können immer durch dieses Element abgebildet werden.

Das Konzept der *data*-Elemente erlaubt es Applikationen, ganze Graphen/ Diagramme, Graphelemente (*node*, *edge*, *label*) oder Elemente für Layoutinformationen mit ganz individuellen Informationen strukturiert anzureichern, für die die OGML keine festen Sprachkonstrukte zur Verfügung stellt. Solche Informationen können beispielsweise Strukturinformationen sein, die von einem bestimmten Layoutalgorithmus verarbeitet werden.

Beispiel 6.10 veranschaulicht die Verwendung des *data*-Elements.

```
1 <data>
2   <string name="description"/>
3     "Color Patterns"
4   </string>
5   <data name="RGB1">
6     <int name="R1" value="#255"/>
7     <int name="G1" value="#00"/>
8     <int name="B1" value="#00"/>
9   </data>
10
11  <data name="RGB2">
12    <int name="R1" value="#00"/>
13    <int name="G1" value="#00"/>
14    <int name="B1" value="#255"/>
```

```
15   </data>
16   <bool name="doGradient" value="true"/>
17   ...
18   <num name="evenDistance" value="0.5"/>
19   <data>
20     ....
21   </data>
22
23 </data>
```

Listing 6.10: Verwendung des data-Elements

Die ursprüngliche Modellierung der OGML sah es vor, das *key-data*-Konstrukt der GraphML in einer ähnlichen Variante zur Verfügung zu stellen. Das *key*-Element hatte dabei die Aufgabe, Typen von anwendungsspezifischen Daten zu deklarieren und Defaultwerte für diese zu definieren. Die *data*-Elemente referenzierten diese Schlüssel und überschrieben bei Bedarf den Defaultwert. Dieser Ansatz wurde allerdings verworfen, da sich abzeichnete, dass ein Parser zur Validierung dieses Konstrukts Kontextinformation benötigt und ein Parsevorgang nur unter erhöhtem Aufwand erfolgreich wäre. Im Übrigen liegen die einzigen Vorteile des *key-data*-Konstrukts in der einmaligen Deklaration von Datentypen und der Definition von Defaultwerten, dem editierenden Benutzer hingegen wird dadurch keine Schreibarbeit erspart.

6.3 Implementierung

Nach dem Abschluss der Spezifikation des neu entwickelten Dateiformats für Graphbeschreibungssprachen war das Ziel der Arbeitsgruppe Dateiformat die Implementierung der notwendigen Klassen, um OGML-Dokumente in Zukunft mit Hilfe des OGDF Speichern und Laden zu können. Außerdem sollten Anpassungen am graphischen Visualisierungstool, dem GoVisual Diagram Editor, getätigt werden, so dass die Schnittstellen für die OGML-Routinen hinzugefügt werden können.

Um die Ziele effizient zu realisieren teilte sich die Arbeitsgruppe in zwei Untergruppen auf, wobei der Schwerpunkt auf der einen Seite im Implementieren einer Speichern-Routine und bei der anderen Teilgruppe bei der inversen Operation – dem Laden eines OGML-Dokumentes – lag.

6.3.1 Speichern

Die Speicherung von Graphdaten wird hauptsächlich in den von `GraphAttributes` erbenden Klassen `ClusterGraphAttributes` und `CompoundGraphAttributes` realisiert. Diese Klassen wurden aus entwicklungstechnisch historischen Gründen gewählt, weil sie schon die *GML*-Serialisierungsmethoden enthielten. Dennoch war eine klassenhierarchisch zentrale Implementation in `GraphAttributes` aufgrund der anfänglich unterschiedlichen Behandlung von *Clusters* und *Compounds* nicht möglich.

Im Laufe des zweiten Semesters wurden von der Arbeitsgruppe *Constraints* zusätzlich diverse *Constraint*-Klassen implementiert, die es ebenfalls zu serialisieren galt. Durch

den unterschiedlichen Aufbau der Serialisierungsdaten, die sich von Constraint zu Constraint unterscheiden, entschlossen sich daraufhin beide Arbeitsgruppen, die Speicherung der *Constraint*-spezifischen Daten den jeweiligen Instanzen einer von `Constraint` erbenden Klasse zu überlassen, so dass sich letztenendes ein spezialisiertes Modul, welches lediglich für die Speicherung von Constraints-Daten zuständig ist, an den Serialisierungsmechanismus angliedert. Ein grosser Vorteil der Eigenserialisierung von *Constraints* ist zudem die Tatsache, dass dadurch die zukünftige Implementierung weiterer Constraints vereinfacht wird.

Während der Ausführung des Algorithmus werden grundsätzlich *drei Ausgabeströme* für Zeichenketten verwendet, um die Anzahl an Iterationen durch alle Graphelemente zu minimieren. Der erste Ausgabestrom dient der Speicherung von *Strukturdaten* und leitet direkt in die entsprechende Datei. Der zweite fungiert als *Puffer*, um XML-Daten, die sich auf *Layout-Daten* beziehen, zwischenzuspeichern. Der dritte Strom wird dagegen als *Puffer* für *Constraints*-Daten verwendet und lediglich initialisiert, um ihn bei der Behandlung von Constraints-Daten den entsprechenden Klasseninstanzen mitzugeben. Somit belasten nur der zweite und dritte Strom den Arbeitsspeicher.

Gegeben durch die Weise, wie die *OGML* Graphelemente verwaltet, werden Knoten und Kanten unterschiedlich behandelt. Während der Algorithmus die Knoten aufgrund ihrer Baumstruktur rekursiv per *Depth-First-Search* durchsuchen muss, können Kanten *iterativ* durchlaufen werden. Dennoch gleicht sich die Behandlung dieser Graphelemente grundsätzlich, da ihre Strukturdaten direkt in die Datei ausgeschrieben werden können, während ihre Layout-Daten zwischengespeichert werden.

6.3.2 Laden

Das Laden eines OGML-Dokumentes wird zeitlich und strukturell in drei aufeinander folgende Phasen eingeteilt:

- Parsen der OGML-Datei und Erstellen des Parsebaums
- Validieren der OGML-Datei anhand des Parsebaums
- Erzeugung des strukturellen Aufbaus des Graphen und Abbildung der Layoutinformationen

Die wesentlichen Implementierungen wurden in den Dateien `OgmlParser.cpp`, `OgmlParser.h` und `Ogml.h` vorgenommen. In dem Headerfile `Ogml.h` ist eine Auflistung aller OGML-spezifischen *Tags*, Attribute und festen Attribut-Werte zu finden. Die Definitionen und Implementierungen der Klassen `OgmlTag` und `OgmlAttribute`, welche im Validier-Vorgang benötigt werden, stehen in der `OgmlParser.h`. Hier wird außerdem die Klasse `OgmlParser` deklariert.

Die Entwicklung eines Parsers für OGML-Dokumente, welcher gleichzeitig einen Parsebaum aufbaut, gehörte nicht zu den Aufgaben der Arbeitsgruppe, da im OGDF bereits ein XML-Parser[3] zur Verfügung stand. Dieser wurde lediglich im Detail erweitert, um

[3] *siehe* `DinoXmlParser.h`

einen besseren Fehlerreport bei unzulässigen Dokumenten zu ermöglichen. Der modifizierte Parser vergleicht nun öffnende und schließende *Tag*-Inhalte und liefert bei einer inkorrekten Datei die Zeile, in welcher die XML-Regeln missachtet werden, beispielsweise wenn zu einem öffnenden *Tag* der korrespondierende Abschluß fehlt. Außerdem wird die Tiefe im erzeugten Parsebaum für jeden Knoten gespeichert. Weitere Änderungen bzw. Erweiterungen am Parser waren nicht nötig, um die nachfolgenden Methoden zu ermöglichen.

Im Anschluss an das Parsen folgt der Vorgang des Validierens, welcher den kompletten Parsebaum traversiert und auf Korrektheit überprüft bzgl. Syntax und Semantik. Dies bedeutet, dass im nachhinein folgende Zusicherungen gemacht werden können:

- *Tag*-Namen sind OGML-spezifisch
- Die Struktur entspricht der OGML-Syntax
- Jedes *Tag* enthält die korrekten *Tag*-Elemente und Attribute
- Attribut-Werte entsprechen dem richtigen Typ
- Ids sind eindeutig und Referenzierungen anderer Elemente sind regelgerecht

Aufgrund des ausführlichen Validierens, welches weit über das Überprüfen der Syntax hinaus geht, kann im weiteren Verlauf größtenteils auf Fehlerüberprüfungen verzichtet werden.

Die abschließende Generierung des Graphen kann wiederum in zwei Phasen unterteilt werden, welche sich außerdem im Aufbau von OGML-Dateien wiederspiegeln. Zuerst werden Knoten und Kanten, welche im *structure*-Block definiert werden, mit ihrer *id* erzeugt. Dies geschieht mit Hilfe einer Traversierung des Parsebaums über die Kindelemente des *structure-Tags*. Bei hierarchischen Knoten, also ineinander geschachtelten *node-Tags*, erfolgen rekursive Aufrufe um die *Compound*-Struktur erfassen zu können. Knoten werden hierbei aufgrund ihrer hierarchischen Abhängigkeit im OGML-Dokument im erzeugten *Compound*-Graphen als Kinder ihrer Eltern-Knoten generiert. Kanten können, nachdem die Knoten erstellt wurden, nun einfach erzeugt werden. Nach der vollständigen Traversierung des *structure*-Teilbaums ist der gespeicherte Graph strukturell rekonstruiert. Um nun die Layoutinformationen zu verarbeiten, folgt eine Traversierung des *layout*-Teilbaums. Hierbei werden die Unterbäume *templates*, *styles* und *constraints* sequentiell durchlaufen, da keine hierarchischen Strukturen in diesem Dokumentteil zu finden sind. Knoten und Kanten können durch ihre Id eindeutig dereferenziert werden und die visuellen Eigenschaften werden gesetzt. Dabei werden eine Reihe von Hilfsmethoden benutzt, um die OGML-spezifischen Werte auf OGDF-Werte abzubilden. Das Laden der Constraints geschieht anders als bei den bisherigen Graphelementen nicht durch die Laden-Routine, sondern ist bei jedem Constraint gesondert implementiert. Es wird lediglich der Constraint-Manager aufgerufen, der zu dem jeweiligen Constraint-Typ, welcher in dem *Tag*-Attribut *type* gespeichert wird, eine Instanz liefert. Dieser wird der Teilbaum des Constraints übergeben, so dass jeder Constraint sich selber laden kann. Dieses Verfahren wurde deshalb so gewählt, damit der Laden-Code vom Hinzufügen neuer Constraints nicht beeinflusst wird und dieses generell eine Vereinfachung ist.

Implementierungs-Details der Laden-Routine

Die Methode des Validierens baut auf den Klassen `OgmlTag` und `OgmlAttribute` auf. In diesen werden für jedes OGML-*Tag* die Anzahl der Kind-*Tags* und die Attribute gespeichert, welche das *Tag* enthalten muss und optional enthalten kann. Außerdem werden für jedes Attribut die möglichen Attributwerte gesetzt. Diese OGML-Strukturen werden vor dem eigentlichen Validier-Vorgang in einer Hashtabelle aufgebaut und während der Traversierung wird der Parsebaum mit den korrespondierenden Elementen aus dieser Tabelle verglichen und somit auf Korrektheit überprüft.

Im Zuge der Entwicklung eines Algorithmus für das Validieren schien es der Arbeitsgruppe Dateiformat sinnvoll, eine Methode zu implementieren, welche in dem zugrunde Liegenden Parsebaum den Graphtypen erkennen kann. Dieses Verfahren läuft rekursiv durch den Baum und prüft im ersten Schritt ob hierarchische Knoten existieren. Ist dies nicht der Fall wird die Ausgabe "normaler Graph" geliefert. Im anderen Fall muss überprüft werden, ob ein Cluster- oder ein Compound-Graph vorliegt. Ein „Compound-Graph" wird ausgegeben, falls die inzidenten Knoten einer Kante in unterschiedlichen Hierarchieebenen liegen. Ansonsten wird „Cluster-Graph" ausgegeben.

Da die Ids im OGML-Dokument und im OGDF nicht die selbe Syntax benutzen – *integer*-Werte im OGDF und *string*-Werte im OGML – musste eine Verfahren entwickelt werden, um in den verschiedenen Phasen eine eindeutige und gleichbleibende Zuordnung zu garantieren. Diese Zuordnung geht über die Id-Verweise hinaus, da im *layout*-Block Graph-Knoten mit Hilfe der OGML-Id dereferenziert werden müssen. Deshalb wird in der ersten Phase – die strukturelle Erzeugung des Graphen durch den *structure*-Block – eine Hashtabelle erzeugt, in welcher den OGML-Ids die zugehörigen Knoten und Kanten des Graphen zugeordnet werden. Durch diese Konstruktion ist es bei der Verarbeitung der Layoutinformationen möglich, effizient aus Ids des OGML-Dokuments die Graphelemente zu erhalten. Für das Laden der Constraints ist diese Hashtabelle insbesondere elementar, da diese sonst keine Möglichkeit haben, aus den OGML-Ids die zugehörigen Graphelemente heraus zu finden.

Ein implementierungstechnisches Problem war die Speicherung der Labels von Knoten. Diese werden im OGDF im HTML-Format gesichert. Ein simples Hinausschreiben dieses Strings in das OGML-Dokument würde beim Laden zu Schwierigkeiten führen, da der XML-Parser diesen HTML-Code interpretiert und daraus jeweils Teilbäume generiert. Dadurch müsste beim Laden dieser Teilbaum wieder in einen String umgewandelt und außerdem dieser Teilbaum vailidiert werden, was bei der komplexen HTML-Syntax der Gruppe als zu schwierig und zeitaufwendig erschien. Deshalb entschied man sich für eine einfachere Lösung, welche den String vor dem Herausschreiben manipuliert und diese Änderungen nach dem Laden wieder rückgängig macht. Dabei wird jede öffnende Klammer durch den Teilstring „>" und jede schliessende Klammer durch „<" ersetzt. Dadurch wird die Interpretation des HTML-Codes durch den XML-Parser umgangen und die Generierung des Teilbaums unterdückt.

Style-Templates bieten in der OGML einen Mechanismus, um mehrfach auftretende, identische Layoutinformationen für Knoten oder Kanten zu speichern. Diese Vorlagen, welche sich in einem Dokument zu Beginn des *layout*-Blocks befinden müssen

während der gesamten Verarbeitung und Abbildung der Stilwerte zur Verfügung stehen. Aus diesem Grund wurden zwei Klassen implementiert, `OgmlNodeTemplate` und `OgmlEdgeTemplate`, welche instanziiert Templates repräsentieren. Diese werden durch ihre Id in einer Hashtabelle referenziert und speichern in den Klassen-Variablen die gewünschten Layoutinformationen.

6.4 Ausblick

Der Zeitaufwand für die Spezifikation der Open Graph Markup Language war höher als anfänglich erwartet. Die Arbeitsgruppe Dateiformat hielt dennoch weiterhin die Maxime bei, alle Ziele des Konzepts umzusetzen, um einen Grundstein für zukünftige Arbeiten am Dateiformat zu setzen.

So entstand bis zum Ende des Sommersemesters eine solide Implementierung des Serialisierungsmechanismus, der den zuvor definierten Anforderungen genügte und auch neue Features, wie z.B. Constraints, berücksichtigte. Aus algorithmischer Sicht werden zukünftigen Änderungen am OGML mit relativ geringem Aufwand durchzusetzen sein, auch wenn als Problemstellung noch die fehlende Unicode-Unterstützung offen steht, da sich die Speichermethoden lediglich den Funktionsumfang der Standard Bibliotheken beschränken. Inwiefern die Problemlösung ohne Zuhilfenahme von fremden Unicodefähigen Bibliotheken möglich ist, bleibt fragwürdig. Architektonisch gesehen wäre als nächster Schritt eine Zentralisierung der Routinen und eine Säuberung des Codes wünschenswert, da durch die Vereinheitlichung von Compounds und Clusters gegen Ende der Implementierungsphase eine unterschiedliche Behandlung während der Speicherung obsolet geworden ist und somit auch aus vererbungstechnischer Sicht eine Trennung nicht mehr von Nöten ist.

7 Arbeitsgruppe GDE-Features

Diese Kleingruppe wurde im zweiten Semester mit der Aufgabe betraut, den GDE benutzerfeundlicher zu machen und auszubauen. Denn die Arbeit mit dem Editor während des ersten Semesters hatte gezeigt, dass der GDE teilweise nicht ganz intuitiv bedienbar war bzw. dass man sich an einigen Stellen mehr Wahlmöglichkeiten und Funktionen wünschte. Die Aufgabe unserer Kleingruppe war es also, dem GDE einige für die praktische Benutzung wichtige Dinge beizubringen, wie zum Beispiel:

- eine Möglichkeit, die Knotenform bereits existierender Knoten zu verändern (in Ellipsen, Hexagone etc.)
- eine Undo-/Redo-Funktion für das Ändern von Knotenformen hinzufügen
- die Möglichkeit, Bilder einzubinden
- Kantenattribute hinzufügen
- das Propertyfenster erweitern
- Multioperationen zulassen, so dass Operationen auf mehrere Objekte gleichzeitig angewendet werden können
- Highlighting implementieren, also bestimmte Objekte in der Darstellung hervorheben
- Vereinheitlichung des Verhaltens (zum Beipsiel die Properties der Objekte bei Neuerstellung aus den Toolbar-Einstellungen übernehmen).

Da es unserer Kleingruppe in der kurzen Zeit nicht möglich gewesen wäre, alle diese Aufgaben zu bearbeiten, wurde es uns überlassen, welche dieser Aufgaben und in welcher Reihenfolge wir sie lösen wollen.

7.1 Knotenformen

Wir haben uns dafür entschieden, zuerst verschiedene Knotenformen zu implementieren. In dem bisherigen Grapheditor gab es schon die Möglichkeit, andere Knotenformen als Rechtecke (insbesondere Ellipsen, Hexagone und UML-Klassen) auszuwählen. Hierfür existierte in der Toolbar eine ComboBox, die so genannte nodeTemplateComboBox, in der man aber nur einstellen konnte, welche Form neu erzeugte Knoten bekommen sollten. Es gab keine Möglichkeit, die Form eines schon existierenden Knotens zu verändern. Wir haben uns dazu entschlossen, dieses Problem zuerst zu lösen.

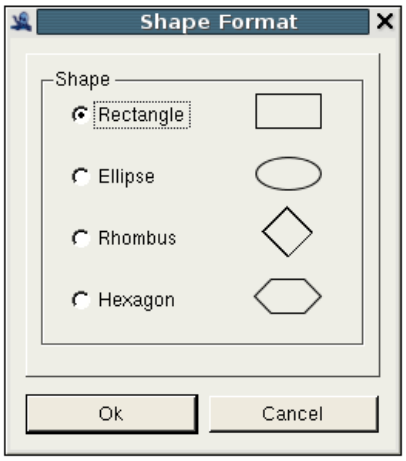

Abbildung 7.1: Das Dialogfenster „Shape Format“

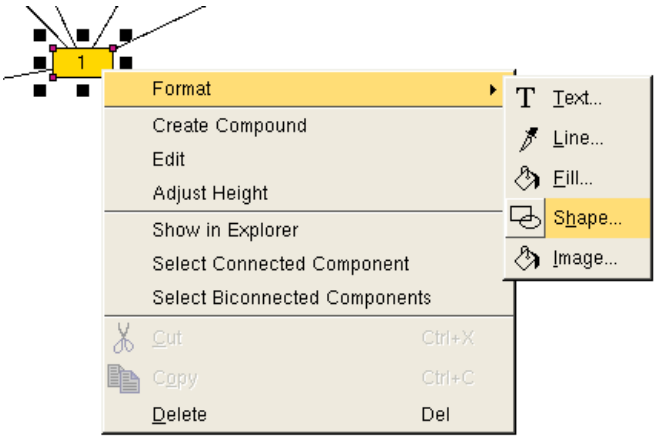

Abbildung 7.2: Das Popup-Menü zum Öffnen des „Shape Format“-Dialogs

Dazu haben wir das Dialogfenster „Shape Format“ eingefügt, das eine Möglichkeit bietet, die Form eines Knotens auszuwählen (siehe Abbildung 7.1). Wenn der Benutzer mit der rechten Maustaste auf einen Knoten klickt, erscheint ein Popup-Menü. Wenn er nun mit der Maus über den Menüpunkt „Format“ fährt, erscheint ein weiteres Popup-Menü, das bisher nur die Punkte „Fill“, „Text“ und „Line“ enthielt. Wir haben dieses Menü um einen weiteren Unterpunkt „Shape“ erweitert, wie man in Abbildung 7.2 sehen kann. Wenn der Benutzer auf diesen Unterpunkt klickt, erscheint das von uns erstellte Dialogfenster, in dem er eine Knotenform auswählen kann.

Auch in der Menüleiste gab es schon einen Menüpunkt „Format“, der die Punkte „Fill“, „Text“ und „Line“ enthielt. Dieses Menü haben wir ebenfalls um den Punkt „Shape“ ergänzt, so dass man auch von hier aus das Dialogfenster „Shape Format“ öffnen kann. Dieser Menüpunkt ist allerdings nur dann aktiviert (also anwählbar), wenn min-

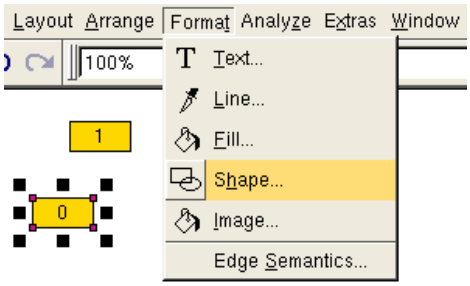

Abbildung 7.3: Der Menüpunkt „Format → Shape“ ist nur anwählbar, wenn der Benutzer mindestens einen Knoten ausgewählt hat

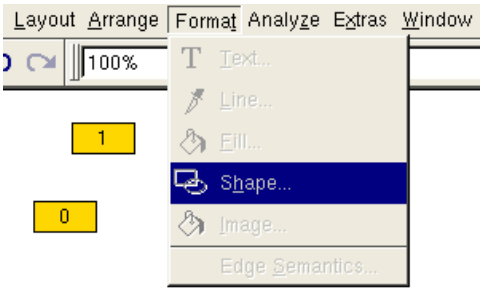

Abbildung 7.4: Wenn keine Knoten ausgewählt sind, ist der Menüpunkt „Format → Shape“ deaktiviert

destens ein Knoten ausgewählt ist, analog zu den restlichen Menüpunkten. Das Aussehen dieses Menüpunkts in beiden Zuständen ist in den Abbildungen 7.3 und 7.4 dargestellt.

7.1.1 Multioperationen

Es ist auch möglich, die Knotenform von mehreren Knoten gleichzeitig zu verändern. Dazu kann der Benutzer einfach mehrere Knoten auswählen, und dann wie oben beschrieben das Dialogfenster „Shape Format“ öffnen. Durch einen Klick auf den Button „OK“ wird die gewählte Knotenform jetzt für alle Knoten übernommen.

7.1.2 Undo / Redo

Für den Fall, dass der Benutzer Änderungen, die er am Graphen vorgenommen hat, rückgängig machen will, gibt es in der Menüleiste den Punkt „Edit → Undo“. Um das Rückgängigmachen dieser Änderungen wieder rückgängig zu machen, gibt es hier auch einen Punkt „Edit → Redo“. Wir haben die Undo- und Redo-Funktion auch für an Knotenformen gemachten Änderungen implementiert.

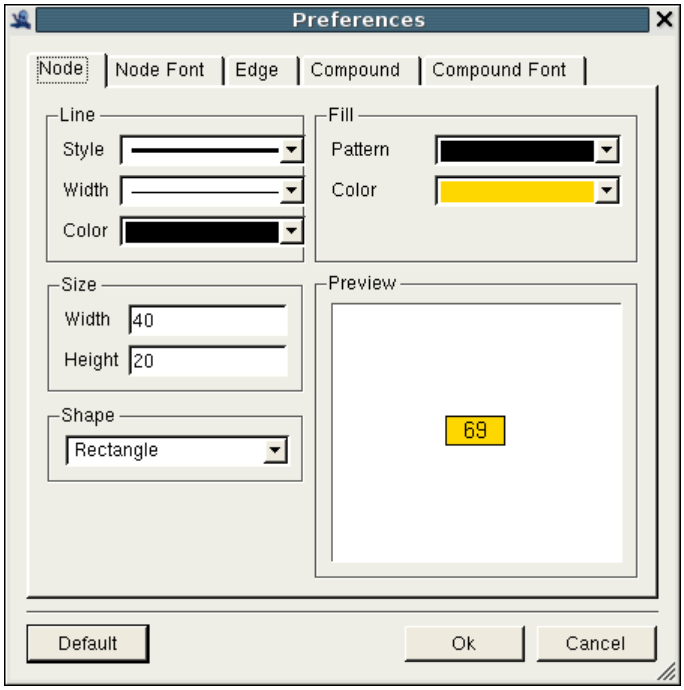

Abbildung 7.5: Das erweiterte Dialogfenster „Preferences“

7.1.3 Das Preferences-Dialogfenster

Eine weitere Aufgabe, die wir uns vorgenommen haben, war, das Dialogfenster „Preferences“ um die Möglichkeit zu erweitern, auch hier Knotenformen auszuwählen. Der Preferences-Dialog kann über den Punkt „Extras“ in der Menüleiste aufgerufen werden und ist dazu da, alle möglichen gewünschten Einstellungen für zukünftig erzeugte Knoten vorzunehmen. Er hat fünf verschiedene Registerkarten, eine davon beinhaltet alle Einstellungen an Knoten, wie zum Beispiel Einstellungen an der Knotengröße, ihrer Füllfarbe etc. Außerdem enthält er eine Preview, die anzeigt, wie ein neu erzeugter Knoten mit den aktuellen Nutzereinstellungen aussehen würde. Diese Registerkarte haben wir also um eine ComboBox erweitert, in der man jetzt die Knotenform für neu erzeugte Knoten bestimmen kann. Danach mussten wir noch die Preview anpassen, so dass sie auch die in der ComboBox gewählte Knotenform berücksichtigt. Das Preferences-Fenster mit der neuen Auswahlbox kann man in Abbildung 7.5 sehen.

Allerdings ergab sich jetzt dadurch ein Problem, dass an zwei Stellen im GDE die Einstellungen für die Knotenformen neu erzeugter Knoten vorgenommen werden konnten, nämlich in dem Preferences-Fenster und in der Format-Toolbar. Wir haben dieses Problem so gelöst, dass die gewünschte Knotenform immer aus der ComboBox in der Toolbar ausgelesen wird. Allerdings bewirkt eine Änderung der Knotenform im Preferences-Fenster durch den Benutzer und ein anschließender Klick auf „OK“, dass die in der Toolbar eingestellte Knotenform automatisch auf die im Preferences-Fenster gewählte

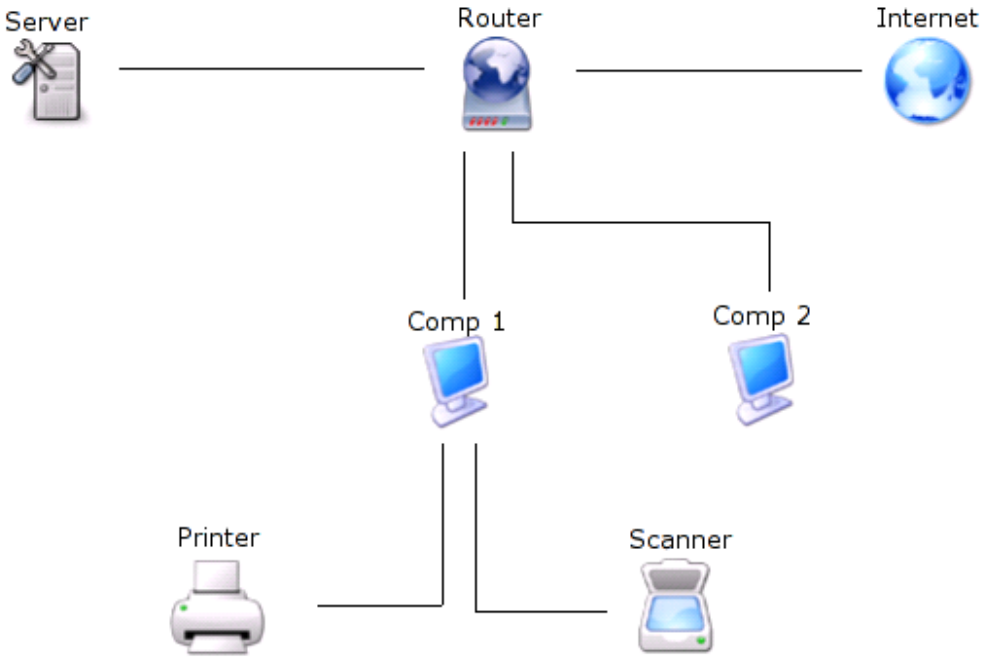

Abbildung 7.6: Ein Beispielgraph bei dem Bilder verwendet wurden

Knotenform umspringt.

7.2 Bilder für Knoten

Ein weiterer visueller Aspekt, welcher bei der Benutzung des GDE auffällt ist die Eintönigkeit der Knotendarstellung. Um visuell ansprechende Diagramme zu erzeugen und benutzerfreundlich darzustellen fehlte die Möglichkeit zur Einbindung von Bildern für Knoten. Diese Funktionalität wurde deshalb im weiteren von der Arbeitsgruppe implementiert. Ein Beispiel eines Graphen, in welchem Bilder verwendet werden ist in Abbildung 7.6 zu sehen.

Um dieses Feature zu implementieren mussten primär Anpassungen am GDE vorgenommen werden. Allerdings musste auch das OGDF erweitert werden, um Bilder speichern und laden zu können.

Die Änderungen im GDE bezogen sich hauptsächlich auf die Zeichnen-Methode der Knotenformen. Beim Zeichnen wird nun üeberprüft, ob das entsprechene Knotenobjekt ein Bild referenziert und im positiven Fall wird dieses auf die Zeichenfläche gemalt und ansonsten wird die gewohnte Methode benutzt. Der Dialog zum Setzen oder modifizieren eines Bildes und dessen Optionen öffnet sich wahlweise über das Popupmenü, welches durch die Rechte Maustaste über einem Knoten aufgerufen wird, oder über den

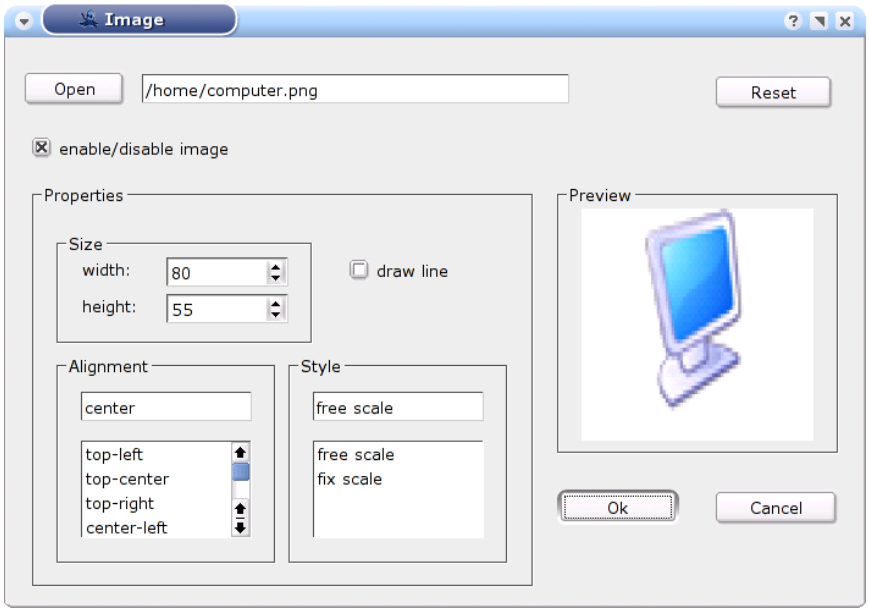

Abbildung 7.7: Der Dialog zum Editieren und Setzen eines Bildes

Menüpunkt im „Format“-Bereich. Dieses Vorgehen ist analog zu den Aufrufmöglichkeiten bei der Änderung der Shape.

Um Bilder für Knoten speichern zu können wurde eine Klasse `GdeImage`[1] implementiert. Die Attribute welche für ein Bild spezifiziert werden können sind:

- *scale* – das Bild wird an die Knotengrösse angepasst oder die Grösse wird vom Benutzer festgelegt
- *alignment* – falls das Bild eine exakte Grösse haben soll, kann hiermit die Ausrichtung des Bildes gesetzt werden
- *drawLine* – ein Wahrheitswert, welcher angibt ob die Umrandungslinie gezeichnet werden soll
- *size* – die exakte Grösse, welche vom Anwender spezifiziert werden kann

Der Dialog, in welchem die Einstellungen vorgenommen werden, ist in Abbildung 7.7 zu sehen. Die Undo-Redo-Funktionalität und Multioperationen wurden ebenfalls für Knotenbilder implementiert, analog zu den Shapes. Die Anbindung an das OGDF erforderte Änderungen in der Klasse `GraphAttributes` und den Laden- und Speichern-Routinen.

[1] *siehe* `graphics.h`

7.3 Ausblick

Bei der Arbeit mit dem GDE mussten wir feststellen, dass es nicht sinnvoll ist, in der oben erwähnten nodeTemplateComboBox und in der von uns hinzugefügten ComboBox im Preferences-Fenster sowohl Knotenformen als auch einen Knotentyp (die UML-Klasse) auswählen zu können. Sinnvoller wäre es, für das Auswählen von Knotentypen und Knotenformen verschiedene Auswahlboxen zu verwenden, denn dadurch wird es zum Beispiel möglich, das Ändern der Knotenform einer UML-Klasse zu verbieten, während man bei „normalen“ Knoten beliebig zwischen verschiedenen Formen wählen kann. Eine Aufgabe für die zukünftigen Entwickler des GDE könnte also sein, die Funktion der schon existierenden Auswahlbox so abzuändern, dass der Benutzer darin nur noch zwischen verschiedenen Knotentypen wählen kann, und zusätzlich eine weitere Auswahlbox einzufügen, mit der man die Knotenform aller ausgewählten Knoten verändern kann.

Insgesamt ist die Funktion der Toolbar, in der sich unter anderem die nodeTemplateComboBox befindet, recht inkonsistent: Auf der einen Seite werden die Änderungen, die der Benutzer in der Toolbar an den Knotenformen vornimmt, für neu erstellte Knoten (und nicht für bereits existierende, aktuell ausgewählte Knoten) übernommen. Auf der anderen Seite werden die Änderungen, die der Nutzer zum Beispiel an der Text- oder der Linienfarbe in der Toolbar vornimmt, nicht für neu erstellte Knoten übernommen, dafür kann man aber das Aussehen der ausgewählten Knoten mit dieser Toolbar verändern.

Um die Funktion der Toolbar zu vereinheitlichen, könnte man in Zukunft zum Beispiel festlegen, in der Toolbar nur Änderungen an den ausgewählten, also an bereits existierenden Knoten zuzulassen. Dazu sollte man also in der Toolbar die ComboBox zur Auswahl verschiedener Knotenformen darauf beschränken, die Formen von ausgewählten Knoten zu verändern. Um das Aussehen von neu erstellten Knoten festzulegen, gibt es wie oben schon erwähnt in der Menüleiste unter „Extras“ bereits den Punkt „Preferences“. Somit wäre auch das in 7.1.3 erwähnte Problem gelöst, dass in zwei verschiedenen Auswahlboxen die Form für zukünftig erzeugte Knoten eingestellt werden kann.

Schließlich sind auch noch einige der Aufgaben zu erledigen, die unserer Kleingruppe zur Auswahl gestellt wurden, die wir aber zurückgestellt haben. Von den anfangs erwähnten Aufgaben ist noch die Erweiterung des Property-Fensters und die Implemetierung des Highlightings übrig geblieben sowie die Aufgabe, neue Kantenattribute hinzuzufügen.

8 Fazit

In diesem Kapitel soll in einem zusammenfassenden Rückblick festgestellt werden, in welchem Ausmaß die Minimalziele der PG 478 erreicht worden sind.

Im OGDF wurde ein Konzept zur Unterstützung von Compounds umgesetzt. Zusätzlich wurde ein passender Layoutalgorithmus implementiert. Die Compoundstrukturen wurden im zweiten Semester in den Editor eingebunden.
Ein weiteres Ziel war die Unterstützung von Constraint-Mechanismen inklusive geeigneter Layoutalgorithmen. Drei verschiedene Constraints wurden zunächst im OGDF entwickelt um dann später durch einen Constraint-Explorer mit dem Editor verbunden zu werden. Auch der Layoutalgorithmus wurde der Auswahl hinzugefügt und berücksichtigt die vom Benutzer ausgewählten Constraints.
Ein geeignetes Dateiformat, OGML, wurde entwickelt und unterstützt Compounds, Constraints und weitere Semantik wie z.B. die Beschreibung von gerichteten, ungerichteten und gemischten Graphen.
Der Editor wurde um mehrere Punkte erweitert. Die Endversion ist ein lauffähiger Editor, der nun Compounds und Constraintmechanismen unterstützt. Außerdem gibt es die Möglichkeit, sich die Übergänge zwischen zwei Layouts als Animation anzusehen. Eine Gruppe kümmerte sich um die Benutzerfreundlichkeit des GDE, der nun z.B. diverse Knotenformen und die Einbindung von Bildern anbietet.
Die Minimalanforderungen, einen Zwischenbericht und einen Abschlussbericht zu verfassen, sind hiermit erfüllt. Die fachbereichsweite Ergebins-Präsentation der Projektgruppe ist geplant und wird zu Beginn des folgenden Wintersemesters 2006/2007 stattfinden.

Zusammendfassend lässt sich sagen, dass die PG 478 alle Minimalziele erreicht und darüber hinaus weitere Aufgaben erledigt hat. Wichtig für die PG waren jedoch nicht nur die inhaltlichen Themen, sondern auch die praktische Programmiererfahrung und das Arbeiten im Team.

Literaturverzeichnis

[1] G. Di Battista. Planarization of clustered graphs. *Revised Papers from the 9th International Symposium on Graph Drawing, pp.60-74*, 2001.

[2] F. Bertault and M. Miller. An algorithm for drawing compound graphs. *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, 1999.

[3] P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE Transactions on computer, Vol.49, No.8*, 2000.

[4] K. Boehringer and F. Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. *Proceedings of the ACM Conference on Computer Human interaction (CHI), pp.43-51*, April 1990.

[5] U. Brandes, M. Eiglsperger, and M. Kaufmann. Sketch-driven orthogonal graph drawing. *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, 2002.

[6] M. Chimani, C. Gutwenger, and K. Klein. `http://ls11-www.cs.uni-dortmund.de/people/chimani/PG478/`. 2005.

[7] C. Friedrich and P. Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications (JGAA), pp.353-370*, 2002.

[8] C. Friedrich and M.E. Houle. Graph drawing in motion ii. *Graph Drawing, pp.220-231*, 2002.

[9] T. M. J. Fruchtermann and E.M. Reingold. Graph drawing by force-directed placement. *Software Practice and Experience, Vol.21, pp.1129-1164*, November 1991.

[10] E. R. Gansner. A technique for drawing directed graphs. *Software Engineering, Vol.19, No.3*, 1993.

[11] Graph Drawing Community. http://graphdrawing.org/.

[12] C. Gutwenger, P. Mutzel, and M. Jünger. Crossings and planarization. *CRC Press, chapter 4*, 2006. to appear.

[13] W. He and K. Marriott. Constrained graph layout. *Constraints: An International Journal, Vol.3*, 1998.

[14] I. Herman, G. Melancon, and M.S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics, Vol.6, No.1, pp.24-43*, 2000.

[15] K. Ryall, J. Marks, and S. Shieber. An interactive constraint-based system for drawing graphs. *Proceedings of the 10th Annual Symposium on User Interface Software and Technology, pp.47-104*, Oktober 1997.

[16] G. Sander. Layout of compound directed graphs. *Technical Report Vol.A03*, Juni 1996.

[17] K. Shoemake and T. Duff. Matrix animation and polar decomposition. In *Proceedings of the conference on Graphics interface '92*, pages 258–264, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[18] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Vol.21 No.4*, 1991.