Brandenburgische Technische Universität Cottbus

Lehrstuhl Graphische Systeme

Diplomarbeit

Layout- und Filterverfahren zur Graphdarstellung in GroIMP

Birka Fonkeng

09. Juni 2007

Diplomarbeit am
Lehrstuhl Grafische Systeme
der Brandenburgischen Technischen Universität Cottbus
Gutachter:
Prof. Dr. Winfried Kurth
Prof. Dr. Claus Lewerentz
Betreuer:
Dipl.-Phys. Ole Kniemeyer
Prof. Dr. Winfried Kurth

Danksagung

An dieser Stelle möchte ich den Menschen danken, die mich bei der Erstellung der Diplomarbeit unterstützt haben.

Ganz besonders möchte ich mich bei Prof. Dr. Winfried Kurth bedanken, der mir durch seine Ratschläge bezüglich des Inhalts und der Ausarbeitung der Diplomarbeit sehr weitergeholfen hat.

Großen Dank ebenfalls an Herrn Ole Kniemeyer, der mir durch sein umfassendes Wissen über GroIMP in Fragen der Implementierung immer mit Rat und Tat beiseite stand.

Cottbus, im Juni 2007 Birka Fonkeng

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

Ein Graph, wie er für diese Diplomarbeit definiert ist, besteht aus je einer endlichen Menge von Knoten und Kanten, wobei die Knotenmenge nicht leer sein darf und jede Kante zwei unterschiedliche Knoten miteinander verbindet. Es gibt mindestens einen Wurzelknoten. Das sind Knoten, die nur ausgehende, aber keine eingehenden Kanten zu anderen Knoten besitzen.

Ein Graph kann gerichtet oder ungerichtet sein. Im ersten Fall besitzen die Kanten je einen eindeutig identifizierbaren Quell- und Zielknoten und sind durch einen Pfeil in Richtung Zielknoten gekennzeichnet. Theoretisch können auch mehrere Kanten zwischen einem Knotenpaar auftreten, was aber im Rahmen dieser Diplomarbeit nicht gesondert berücksichtigt wurde, da dieses Merkmal für den konkreten Anwendungsfall vernachlässigbar war.

Meist werden Graphen zur graphischen Visualisierung von Problemen genutzt. Die Knoten stehen dabei symbolisch für bestimmte Objekte, während die Kanten deren Beziehungen untereinander darstellen. So können sie biochemische Netzwerke darstellen (vgl. [ZH06]), Software-Strukturen, wie z. B. ER- oder Klassendiagramme darstellen. Sie finden sich aber auch in Bereichen des alltäglichen Lebens wieder, so z. B. bei der Zeichnung von U-Bahn-Linienplänen (vgl. [NW06]). Während Rechner mit einer zweidimensionalen Matrix als Graphdefinition bestens auskommen, hat der menschliche Benutzer, neben der Richtigkeit, zusätzlich auch noch andere Anforderungen an ein gutes Graphlayout. Diese werden in Kapitel 2.1 beschrieben. Anschließend werden in 2.2 und 2.3 einige Arten und Beispiele von Layoutalgorithmen vorgestellt.

Eine konkrete Implementierung der zuvor vorgestellten Layoutverfahren in die Software GroIMP wird in Kapitel 2.4 dokumentiert. GroIMP steht für *Growth Grammar Related Interactive Modelling Platform* und ist eine 3D-Modellierungsplattform auf der Grundlage von relationalen Wachstumsgrammatiken. [KBK07] Sie wurde am Lehrstuhl für Grafische Systeme an der BTU Cottbus entworfen und befindet sich in ständiger Weiterentwicklung.

Das Anwendungsgebiet dieser Software liegt im naturwissenschaftlichen Bereich. Es können z. B. Pflanzen in ihrem Aussehen oder Verhalten (z. B. Reaktionen auf Umwelteinflüsse) sowohl auf biologischer als auch auf chemischer Ebene modelliert werden.

Die Regeln, die zur Beschreibung des Wachstums, des Verhaltens und der 3D-Visualisierung

notwendig sind, werden durch relationale Wachstumsgrammatiken ausgedrückt. Diese stellen eine Erweiterung der L-Systeme dar, welche biologische Strukturen und Regeln mit Hilfe von Zeichenketten und einem festgelegten Ersetzungssystem definieren können. Die Erweiterung besteht nun darin, dass die Informationen nicht mehr rein in Zeichenketten codiert werden müssen, sondern eine Graph-Struktur aufgebaut wird, welche aus Knoten und gerichteten Kanten besteht [KN06]. Relationale Wachstumsgrammatiken sind also eine spezielle Klasse paralleler Graph-Grammatiken.

Die durch Regelanwendung erhaltenen Graphen können in GroIMP in einer zweidimensionalen Ansicht betrachtet werden. Je nach Komplexität des 3D-Modells kann der Graph viele oder wenige Knoten besitzen. Die Knoten haben meist ein einheitliches Aussehen. Einige fest vordefinierte Kantentypen bewirken eine unterschiedliche Strichelung der gezeichneten Kanten.

Ziel dieser Diplomarbeit ist es, diese zweidimensionale Ansicht für den Betrachter übersichtlicher darzustellen, um die Arbeit mit den Graphen angenehmer gestalten zu können, damit notwendige Informationen leichter gefunden werden können. Hierfür wurden einige Layoutalgorithmen implementiert, welche zwar unterschiedliche Ansätze verfolgen, aber miteinander gemeinsam haben, die Struktur des Graphen möglichst harmonisch abzubilden. Ein Teil der Layoutalgorithmen beruht auf der Diplomarbeit von Dexu Zhao (vlg. [ZH06]). Die kantenbasierten Verfahren wurden komplett neu entwickelt.

Desweiteren wurden Filter entwickelt, die es dem Betrachter erlauben, bestimmte Knoten an Hand von frei wählbaren Kriterien aus der aktuellen Sicht zu entfernen (ohne sie löschen zu müssen) bzw. sie in einem einzigen Knoten zusammenzufassen, oder sie mit geeigneten graphischen Formatierungen im Graphenbild zu betonen. Diese werden in Kapitel 3 näher erläutert.

2 Layout-Algorithmen

2.1 Vorbemerkungen

Ein Graphlayout ist eine visuelle Abbildung von Knoten und Kanten. Dabei werden den Knoten Positionen innerhalb eines zweidimensionalen Koordinatensystems zugewiesen. Die Knoten haben neben der Positionsangabe weitere graphische Eigenschaften wie Form, Höhe und Breite. Ein Layoutalgorithmus berechnet unter Zuhilfenahme von bestimmten Parametern die Positionswerte der Knoten und damit auch das Graphlayout.

Es existieren viele verschiedene Arten von Layoutalgorithmen. Eine mögliche Unterteilung wäre die Unterscheidung von knoten- und kantenbasierten Layoutverfahren. Während die einen das Graphlayout durch optimale Positionierung der Knoten erreichen, legen die anderen ihren Schwerpunkt auf die Kanten des Graphen und ordnen diese nach einem bestimmten Algorithmus neu an. Es gibt einfache Verfahren, die die Knoten eines Graphen wie ein geometrisches Muster anordnen. Andere wiederum betonen die Beziehungen der Knoten miteinander, indem durch Kanten verbundene Knoten dicht aneinander positioniert, und Knotenpaare, die keine gemeinsame Kanten besitzen, weiter voneinander weg platziert werden. Darüber hinaus gibt es Algorithmen, die den Schwerpunkt auf die Darstellung möglicher Hierarchiebeziehungen setzen: Ausgehend von den Wurzeln (Knoten ohne eingehende Kanten) werden die Knoten baumartig angeordnet.

Layoutalgorithmen erhöhen die Übersichtlichkeit von Graphen. Um dieses Ziel zu erreichen, gibt es verschiedene Kriterien, die von den Verfahren manchmal mehr, manchmal weniger und manchmal auch gar nicht priorisiert werden (aus [SU02]):

closeness Verbundene Knoten werden möglichst nah beieinander platziert

smallest separation Knoten sollten nicht zu dicht aneinander gezeichnet werden

fixed edge length Kantenlängen haben eine einheitliche Größe

symmetry Spiegelsymmetrie der Knoten im gezeichneten Graphenbild

uniform distribution Innerhalb eines Zeichenfensters, das eine feste Höhe und Breite hat, sollen die Knoten möglichst gleichmäßig verteilt werden

adaption to the frame Das Format der Zeichnung ist an das Zeichenfenster angepasst

edge crossing minimization Minimierung der Anzahl der Kantenüberkreuzungen

edge directions Berücksichtigung der Richtung von gerichteten Kanten

Auf den folgenden Seiten werden einzelne Verfahren in der Theorie vorgestellt. Im Kapitel Implementierungen wird desweiteren gezeigt, wie diese Algorithmen in GroIMP umgesetzt wurden.

2.2 Vorstellung verschiedener knotenbasierter Layoutverfahren

Knotenbasierte Layoutalgorithmen ordnen die Knoten eines Graphen anhand von festgelegten Regeln neu an. Mögliche Arten dieser werden in den folgenden Unterkapiteln beschrieben.

2.2.1 Einfache Layout-Algorithmen

Bei den folgenden Layout-Algorithmen spielen die Kanten des Graphen keine Rolle. Die Knoten werden per Zufall oder durch geometrische Formeln angeordnet.

Random

Der Algorithmus sieht vor, die Positionen aller Knoten des Graphen per Zufallsfunktion neu zu bestimmen. Dabei bekommt jeder Knoten neue, zufällig bestimmte x- und y-Koordinaten. Konstanten helfen, dass die Knoten innerhalb eines festgelegten Zeichenfensters bleiben, damit der Graph nicht zu sehr auseinander gezogen wird.

Da dieses Layout ausschließlich auf dem Zufall basiert, kann es keine Kriterien zur verbesserten Darstellung von Graphen erfüllen. Die Knoten können nach Anwendung des Verfahrens übereinander liegen, die Kanten sind beliebig lang und überkreuzen sich möglicherweise öfters. Eine Symmetrie ist im Layout nicht erkennbar.

Bei einer Variante dieses Verfahrens wird ein bereits gezeichneter Graph lediglich so modifiziert, dass alle Knoten um einen Zufallsvektor (mit beschränkter Länge) verschoben werden.

Square

Ausgehend von einem Knoten des Graphen, der keine eingehende Kanten besitzt, zeichnet der Square-Layout-Algorithmus spaltenweise die Knoten nacheinander in einem Raster. Dabei

kann das Raster von links nach rechts aufgebaut werden, oder aber auch von oben nach unten.

Zuerst wird ermittelt, wieviele Knoten in einer Spalte Platz haben, um ein gleichmäßig gezeichnetes Rasterbild zu bekommen. Die maximale Anzahl $numberOfNodesPerLayer$ von Knoten pro Spalte beträgt demnach:

$$numberOfNodesPerLayer := Knotenanzahl/Spaltenanzahl \tag{2.1}$$

Ergibt sich aus der obigen Rechnung ein Restwert, so muss zum Ergebnis noch eine Eins hinzuaddiert werden. Ein Zähler zählt die Position des Knotens innerhalb der Spalte. 0 ist die 1. Stelle, die maximale Anzahl Knoten pro Spalte minus 1 die letzte. Nach dem Zeichnen eines Knotens an der letzten Stelle einer Spalte rutschen die nächsten noch ungezeichneten Knoten eine Spalte weiter und der Zähler beginnt wieder bei Null.

Dieser Algorithmus erfüllt nur näherungsweise das Symmetrie-Kriterium bezüglich der Anordnung der Knoten. Die Knoten werden zwar regelmäßig in das Raster eingeordnet, die letzte Spalte muss aber nicht komplett gefüllt sein. Den Kanten wird bei Berechnung des Algorithmus keine besondere Beachtung geschenkt.

Circle

Dies ist ein Layout-Algorithmus, der die Knoten des Graphen auf einem oder mehreren Kreisen anordnet. Sind auf Grund einer hohen Anzahl von Knoten mehrere Kreise (= Schichten) gewünscht, werden diese der Übersichtlichkeit halber ineinander gelegt.

Den Mittelpunkt der Zeichenfläche bilden die Koordinaten (0,0). In Abhängigkeit von der Knotenzahl und der vom Benutzer gewünschten Anzahl von Kreisen wird zuerst der Abstand vom Mittelpunkt zu allen anderen Knoten berechnet. Dieser Radius ist für alle Knoten, die auf demselben Kreis liegen, gleich. Um sie gleichmäßig auf die Kreise zu verteilen, kann die Funktion 2.1 benutzt werden. Der Radius r des ersten Kreises berechnet sich dann aus der Anzahl der Knoten im Kreis geteilt durch π. Alle weiteren Kreise liegen mit geeignetem Abstand außerhalb von diesem.

Um eine gleichmäßige Anordnung der Knoten auf den Kreisen zu erreichen, muss anschließend eine konstante Winkelgröße ermittelt werden, die wiederum abhängig von der Anzahl der Knoten ist. Diese Winkelgröße ergibt sich aus der Division von dem 360°-Zeichenbereich mit der Anzahl der Knoten auf dem Kreis. Zusammen mit dem Radius können so die neuen Positionen der Knoten auf dem Kreis berechnet werden. Dabei gelten folgende geometrische Kreisformeln, um die Knoten auf einem Kreis im Uhrzeigersinn anzuordnen [BE98]:

Gegeben: r..Radius, ϕ..Winkel, Mittelpunkt des Kreises = (0,0)

$$x\text{-}Koordinate := r * \sin\phi \tag{2.2}$$

$$y\text{-}Koordinate := r * \cos\phi \tag{2.3}$$

Je Knoten wird ein Vielfaches von ϕ angenommen.

Durch die gleichmäßige Anordnung der Knoten auf Kreise erhält das Layout eine näherungsweise symmetrische Darstellung bezüglich der Knoten. Andere Ästhetikkriterien werden nicht berücksichtigt.

2.2.2 Hierarchische Layout-Algorithmen

Layoutverfahren dieser Gruppe eignen sich vor allem für gerichtete Graphen. So kann mit diesen Layouts der Weg durch den Graphen beginnend von dem Knoten ohne eingehende Kanten (Wurzel) bis zu denen ohne ausgehende Kanten (Blätter) verfolgt werden. Das neue Graphlayout kann von oben nach unten bzw. von links nach rechts aufgebaut werden. Die auszuführenden Schritte aller hierarchischen Verfahren lauten [DB99]:

1. Temporäres Entfernen von Zyklen im Graph (falls vorhanden)
2. Zuweisung der Knoten zu horizontalen (vertikalen) Schichten und Berechnung der y-(x-)Koordinate (abhängig von der Richtung des Graphaufbaus)
3. Reduktion von Kantenkreuzungen, indem Knoten innerhalb der Schichten geordnet werden
4. Berechnung der x-(y-)Koordinate eines jeden Knoten

Tree

Ein Baum ist ein aus Knoten und Kanten bestehender Graph, der keine Zyklen enthält. Man kann mit diesem Modell Hierarchien darstellen. Der Layout-Algorithmus beginnt bei der Wurzel. Dies ist ein Knoten, der keine eingehenden Kanten besitzt. Er wird als besucht markiert. Ausgehend von ihm wird nun ein unmittelbar mit ihm verbundener Knoten betrachtet. Dieser wird eine Hierarchieschicht tiefer gezeichnet und ebenfalls markiert. Für alle seine Geschwisterknoten wird die Tiefensuche rekursiv aufgerufen, sofern der aktuell betrachtete Knoten und seine Kindsknoten sowie deren Geschwister rekursiv durchsucht und markiert wurden. Auch wenn laut Definition keine Zyklen im Graph erlaubt sind, wird durch das Markieren besuchter Knoten vermieden, dass Knoten mehrmals durchlaufen werden.

Man kann das Tree-Layoutverfahren in zwei Phasen einteilen:

1. Zuweisung der Knoten zu ihren Hierarchieebenen
2. Platzierung der Knoten in angemessenen Abständen innerhalb der Schichten

Wird der Baum von unten nach oben gezeichnet, dann können im ersten Schritt die y-Koordinaten, im zweiten die x-Koordinaten berechnet werden. Beim horizontalem Anordnen der Knoten dagegen werden zuerst alle x-Positionswerte ermittelt und im zweiten die y-Werte aller Knoten.

Sugiyama

Da das Sugiyama-Layout Hierarchie-Beziehungen der Knoten innerhalb des Graphen darstellen soll, ist das wichtigste zu berücksichtigende Zeichenkriterium für diesen Algorithmus das Minimieren von Kanten, die in entgegengesetzter Richtung der Hierarchie verlaufen [SU02]. Eine möglichst gleichmäßige Verteilung der Knoten gilt es desweiteren zu erreichen (*uniform distribution*) und ebenso ist zu vermeiden, dass die Kanten über zu viele Hierarchiestufen verlaufen, was der Übersichtlichkeit des Graphenlayouts schaden würde (*minimization of edge crossings*).

Der Ablauf des Sugiyama-Algorithmus orientiert sich an dem allgemeinen Modell der hierarchischen Layoutverfahren. Zu Beginn werden die Knoten den Schichten zugewiesen. Wurzelknoten liegen auf der ersten, ihre direkten Kinder auf der zweiten, deren Kinder auf der dritten Schicht, usw. Im Gegensatz zum Tree-Layout wird die Reihenfolge der Knoten auf jeder Schicht nicht willkürklich durch die Tiefensuche festgelegt, sondern ein Kantenkreuzungsminimierungsalgorithmus versucht durch Verschieben der Knoten möglichst wenige Kantenüberkreuzungen zwischen jeweils zwei Schichten zu erreichen. Eine solches Verfahren ist die Barycenter-Methode. Die Position eines Knotens in einer Schicht wird durch den Durchschnitt der Positionen der mit ihm verbundenen Knoten der vorhergehenden Schicht ermittelt (nach [LA03]):

Gegeben: m .. aktuell betrachteter Knoten, $N(n)$ .. Menge der Knoten der vorher betrachteten Schicht, die mit dem aktuell betrachteten Knoten verbunden sind, $index(n)$ .. Index in der Schicht

$$avg(m) := 1/|N(m)| * \sum_{e \in N(n)} index(n) \tag{2.4}$$

Nach Berechnung dieses Wertes für jeden Knoten innerhalb der Schicht werden die Knoten, beginnend mit dem, der den niedrigsten *avg*-Wert besitzt, und endend mit dem, für den der höchste *avg*-Wert errechnet wurde, parallel auf der Zeichenfläche angeordnet.

2.2.3 Kräftebasierte Layout-Algorithmen

Zeichenverfahren, die dieser Kategorie angehören, berechnen für die Knoten des Graphen, auf den der Algorithmus angewendet werden soll, die Anziehungs- bzw. Abstoßungskräfte zu anderen Knoten. Dabei gilt: Knoten, die eine Kante gemeinsam haben, ziehen sich an, Knoten, die nicht direkt miteinander verbunden sind, stoßen sich ab. Die Layout-Algorithmen berechnen für anziehende Kräfte zwischen allen verbundenen Knotenpaaren eines Graphen positive Werte, für abstoßende Kräfte zwischen allen möglichen Knotenpaaren negative, die als Vektoren gespeichert werden. Daraus ergibt sich, dass im Idealfall die Summe der Kräfte aller Knoten eines Graphen gleich 0 ist. Dies ist aber in der Realität bei relativ großen Graphen auf Grund der sehr vielen Iterationsschritte nur schwer zu erreichen und dann auch nur mit hohem Rechenaufwand möglich, da allein ein Durchlauf des Kraftalgorithmus' eine Komplexität von $O(n^2)$ hat. Daher gibt es eine Variable *accuracy*, die die noch akzeptable Abweichung der vom Algorithmus berechneten Gesamtkraft zur tatsächlichen (= 0) beschreibt. Ist sie erreicht, wird von weiteren Iterationen zur Berechnung noch besserer Knotenpositionen abgesehen.

Spring

Das Spring-Layout ist ein sehr einfaches kräftebasiertes Verfahren, das nach den oben geschilderten Prinzipien funktioniert: (Unverbundene) Knotenpaare stoßen sich ab, Kanten agieren als Federn und ziehen dagegen die durch sie verbundenen Knoten aneinander an. Je nach Größe der Abstoßungskonstante stoßen sich unverbundene Knoten mehr oder wenig stark ab. Beispielfunktionen zur Berechnung der Kräfte sind im Kapitel Implementierungen angegeben.

Dieses Layout erfüllt das Kriterium, dass verbundene Knoten möglicht nah beieinander und unverbundene nicht zu dicht aneinander gezeichnet werden sollen. Kantenkreuzungen werden nicht minimiert.

Eades

Das Eades-Layout baut auf das Springverfahren auf. Verbundene Knoten ziehen sich sich an oder stoßen sich ab, gemäß ihrer Entfernung zueinander und der Wirkung ihrer vorhan-

denen oder nicht vorhandenen Kante. Der Algorithmus legt besonders Wert auf einheitliche Kantenlängen (fixed edge length) und eine ausgeglichene Verteilung der Knoten und ihrer Kanten im Zeichenfenster.

Für jede Kante des Graphen wird die Anziehungskraft der beiden Knoten, die durch diese Kante verbunden sind, berechnet (nach [DH96]):

Gegeben: e .. Kante des Graphen, $dist(e)$ .. Länge von e, m, n ... Knoten verbunden durch e, $springConst$ = const., $springLength$ = const.

$$Anziehungskraft(m, n) := springConst * \log(dist(e)/springLength) \tag{2.5}$$

Die Konstante $springLength$ ist abhängig vom minimalen Abstand der Knoten zueinander.

Für jedes Knotenpaar wird die Abstoßungskraft zueinander wie folgt berechnet (nach [DH96]):

Gegeben: m, n ... Knoten eines Graphen, $dist(m, n)$ .. Entfernung zweier Knoten voneinander, $repellingConst$ = const.

$$Abstoßungskraft(m, n) = repellingConst/dist(m, n)^2 \tag{2.6}$$

FruchtermanReingold

Der Fruchterman-Reingold-Layout-Algorithmus ähnelt dem Eades-Layout. Er stellt eine Verbesserung dessen dar. Er legt besonders auf zwei Dinge wert. Das sind zum Einen, dass Knoten, die durch Kanten miteinander verbunden sind, möglichst dicht zueinander gezeichnet werden sollen, und zum Anderen, dass allgemein alle Knoten nicht zu dicht aneinander platziert werden sollen [FR91]. Die Qualität des sich aus diesem Algorithmus ergebenden Layouts hängt von der Knotenanzahl und der Größe der zur Verfügung stehenden Zeichenfläche ab. Die ideale Distanz id zwischen den Knoten ergibt sich folgendermaßen [SU02]:

Gegeben: $frameX$, $frameY$: Breite und Länge der Zeichenfläche; $scaleConst$ = const.

$$id := scaleConst * \sqrt{(frameX * frameY)/Knotenanzahl} \tag{2.7}$$

Wie bei allen bisher beschriebenen kräftebasierten Algorithmen berechnet auch der Fruchterman-Reingold-Algorithmus die Anziehungskräfte nur für adjazente Knoten, die abstoßenden allerdings für alle Knotenpaare. Dies geschieht jeweils einmal pro Iteration inklusive der anschließenden Verschiebung der Knoten gemäß der erhaltenen Kräfte. Anders sind allerdings die Funktionen zur Ermittlung dieser Kräfte. Die anziehende Kraft zweier Knoten m

und n berechnet sich wie folgt:

$$Anziehungskraft(m,n) := (dist(m,n))^2/id \tag{2.8}$$

Die Abstoßungskraft ergibt sich aus:

$$Abstoßungskraft(m,n) := -(id)^2/dist(m,n) \tag{2.9}$$

Im besten Fall, wenn an einem Knoten die Summe der Anziehungskräfte und der Abstoßungskräfte gleich 0 sind, ist die Entfernung des Knoten zu den anderen gleich der idealen Distanz.

DavidsonHarel

Es handelt sich um einen Layout-Algorithmus, der durch die Technik des Simulated Annealing das Layout eines Graphen verbessert.

Simulated Annealing bedeutet, dass bei jeder Iteration dieses Layout-Algorithmus für jeden Knoten des Graphen neue Koordinaten getestet werden. Wird die Gesamtkraft, die wie bei jedem kräftebasierten Layoutverfahrens gegen 0 tendiert, bei dem Schritt verringert, dann werden diese Positionen auch tatsächlich den Knoten zugewiesen. Verschlechtert sich dagegen die Gesamtkraft durch die möglichen neuen Koordinaten, dann wird mit einer Zufallsfunktion entschieden, ob diese Koordinaten angenommen oder verworfen werden. Die Wahrscheinlichkeit der Akzeptanz wa berechnet sich folgendermaßen [LA03]:

Gegeben: $\Delta E := E - E_{neu}$, Energieänderung, T .. Temperatur

$$wa := e^{\Delta E/T} \tag{2.10}$$

Der Algorithmus des Simulated Annealing sieht wie folgt aus (vgl. [DH96]):

1. Festlegen einer initialen Temperatur und Konfiguration
2. Wiederholung:
   a) Ermitteln neuer Knotenpositionen für jeden Knoten
   b) Prüfen aller neuen Positionen auf Energiereduzierung oder Erfüllen der Zufallsfunktion, bei Erfolg Zuweisung zu den Knoten
3. Erniedrigung der Temperatur

4. Ende bei Erfüllung der Abbruchbedingung (Zeit, Kontrollparameter, ...), ansonsten weiter mit Schritt 2

Eine Kostenfunktion beschreibt die für das Ziellayout wichtigen Kriterien und deren Gewichtungen. Um ein harmonisches Layout zu erhalten, gilt es, diese Kostenfunktion, die sich aus der Summe aller betrachteten Kriterien ergibt, zu minimieren.

$$Kostenfunktion := a + b + c + d + e + ... \tag{2.11}$$

a Abstoßungskräfte aller Knotenpaare

b Verteilung der Knoten innerhalb der Zeichenfläche

c Anziehungskräfte der durch Kanten verbundenen Knoten

d Anzahl der Kantenüberkreuzungen

e Optimierung der Knoten-Kanten-Entfernungen

... weitere mögliche Kriterien

Davidson und Harel legen bei ihrem Layoutverfahren auf folgende Kriterien besonders wert [DH96]:

uniform distribution: Zum Einen soll sichergestellt werden, dass die Knoten nicht zu dicht beieinander gezeichnet werden sollen. Um so mehr Knoten ein Graph besitzt, desto kleiner muss die Entfernung aller Knotenpaare zueinander sein, um sie in einem fest vorgebenen Rahmen zeichnen zu können. Dies sichert die Formel zur Berechnung der Abstoßungskraft zwischen den Knotenpaaren. Sie ist identisch mit der von Eades (vgl. Formel 2.5).

Außerdem sollen die Knoten auch innerhalb des vorgegebenen Zeichenfensters gleichmäßig verteilt werden.

Gegeben: m .. Knoten, $positionConst$ .. Normalisierungsfaktor, r(ight), l(eft), t(op), b(ottom) ... Entfernungen von einem Knoten zu den Grenzen des Zeichenfensters

$$Positionierung(m) := positionConst * (1/r_m^2 + 1/l_m^2 + 1/t_m^2 + 1/b_m^2) \tag{2.12}$$

fixed edge length: Gegeben: e .. Kante, $attractionConst$ .. Anziehungskonstante, $dist(e)$ .. Kantenlänge

$$Anziehungskraft(e) := attractionConst * dist(e)^2 \tag{2.13}$$

Desweiteren sind laut Davidson und Harel auch Kriterien zur Minimierung von Kantenüberkreuzungen (edge crossing minimization) und die Optimierung von Knoten-Kanten-Entfernungen für ein gutes Layout ebenso wünschenswert. Weil diese aber oftmals im Konflikt zu anderen Darstellungskriterien stehen, können sie zwar im Layoutalgorithmus berücksichtigt werden, würden allerdings eine nicht so hohe Priorität wie die anderen Kriterien genießen. Davidson und Harel geben keinen expliziten Kantenüberkreuzungsminimierungsalgorithmus an. Sie führen eine Konstante ein, die als Bestrafung für eine Kantenüberkreuzung steht und die auch in die Kostenfunktion des Verfahrens miteingeht. Ebenso können auch Knoten und Kanten, die sich zwar nicht überkreuzen, aber dennoch ungewünscht sehr dicht aneinander liegen, in die Kostenfunktion miteinfließen.

Gegeben: m .. Knoten, e .. Kante, $distConst$ .. Entfernungskonstante, $dist(m, e)$ .. Entfernung von m zum dichtesten Punkt auf e

$$Knoten\text{-}Kanten\text{-}Entfernung := distConst/(dist(m, e))^2 \tag{2.14}$$

2.2.4 Weitere knotenbasierte Layout-Algorithmen

EnergyModel

Ein Energiemodell besteht aus einer Funktion, die die Gesamtenergie eines Layouts berechnet und aus einem Algorithmus, der versucht, die berechnete Gesamtenergie zu minimieren. Als Beispiel wurde das r-PolyLog-Energiemodell (nach [NO03]) implementiert. Die Energie eines Graphenlayouts ist folgendermaßen definiert:

Gegeben: E .. Kantenmenge, V .. Knotenmenge, $dist(e)$ .. Kantenlänge, $dist(m, n)$ ... Entfernung der Knoten zueinander

$$rPolyLogEnergie := \sum_{(e)\in E} dist(e)^r - \sum_{(m,n)\in V} ln(dist(m, n)) \tag{2.15}$$

Der erste Teil der Funktion beschreibt die anziehenden Kräfte zwischen den durch Kanten verbundenen Knoten des Graphen. Je nach Wahl des Parameter r ergibt sich eine hohe Bildung von Subgraphen (Clusterung), einheitliche Kantenlängen oder etwas zwischen diesen

beiden Extremen. Der zweite Teil der Funktion steht für die Berechnung der abstoßenden Kräfte zwischen Knotenpaaren im Graphen.

Um die Energie dieses Modells zu verringern, eignet sich der Barnes-Hut-Algorithmus. Er unterteilt den Graphen rekursiv in Subgraphen, auch Quadtrees (vier Teilbäume) genannt. Für jeden Quadtree berechnet der Algorithmus das Massezentrum und die Gesamtmasse von allen Knoten innerhalb des Subgraphen. Durch ein anschließendes Durchlaufen dieser Knoten werden die einwirkenden Kräfte auf diese ermittelt. Die Subgraphen werden wiederum in Subgraphen unterteilt, und deren Kräfte ermittelt. Das geschieht soweit, bis die Subgraphen nicht mehr geteilt werden können.

Touch

Das Touch-Layout wurde für das JGraph-Projekt (vgl. [JG07]) entwickelt. Zusammen mit den Erkenntnissen über das Touch-Layout von Dexu Zhao (vgl. [ZH06]) und eigenen Anpassungen entstand daraus das Touch-Layout für GroIMP. Es ähnelt den kräftebasierten Layoutalgorithmen insofern, weil es Funktionen zur Berechnung der neuen Knotenpositionen für verbundene Knotenpaare und andere für alle Knotenpaare eines Graphen bereitstellt. Allerdings berechnet dieses Layoutverfahren keine zu minimierende Gesamtenergie des Graphen, sondern lediglich neue Koordinatendaten der Knoten. Funktionen, die für alle miteinander verbundenen Knotenpaare neue Koordinaten bestimmen, sind im Folgenden dargestellt (frei nach [ZH06]):

Gegeben: m, n .. Knoten, die durch eine Kante e miteinander verbunden sind, c .. const.

$$m.x := m.x - ((m.x - n.x) * rigidity * dist(e)) \tag{2.16}$$

$$m.y := m.y - ((m.y - n.y) * rigidity * dist(e)) \tag{2.17}$$

$$n.x := n.x + ((m.x - n.x) * rigidity * dist(e)) \tag{2.18}$$

$$n.y := n.y + ((m.y - n.y) * rigidity * dist(e)) \tag{2.19}$$

Aufgrund der unterstellten Anziehung zwischen Knoten, die eine Kante gemeinsam haben, rückt diese Funktion die Knotenpaare näher zueinander. Dabei muss der *rigidity*-Faktor aber auch sehr klein (z. B. 0.00005) gewählt werden.

Eine abstoßende Wirkung hat dagegen die Funktion zur Berechnung der neuen Knotenpositionen zwischen allen (= verbundenen und unverbundenen) Knotenpaaren. Sie wird ausgeführt, wenn die Entfernung des aktuell betrachteten Knotenpaares unter einem bestimmten

Grenzwert (z. B. bei $c = 5$, wenn $dist(m, n) < 5$) gefallen ist:

Gegeben: m, n .. Knoten, c .. const.

$$m.x := m.x - (m.x - n.x)/dist(m, n) * (MAX(m.width, n.width) * c)^2; \tag{2.20}$$

$$m.y := m.y - (m.y - n.y)/dist(m, n) * (MAX(m.height, n.height) * c)^2 \tag{2.21}$$

$$n.x := n.x + (m.x - n.x)/dist(m, n) * (MAX(m.width, n.width) * c)^2 \tag{2.22}$$

$$n.y := n.y + (m.y - n.y)/dist(m, n) * (MAX(m.height, n.height) * c)^2 \tag{2.23}$$

Die Werte *height* und *width* eines Knoten ergeben sich aus der Größe ihrer Darstellung im Layout. Sind die Knoten beispielsweise als Rechtecke dargestellt, dann ergibt sich *height* aus der y-Ausdehnung, *width* aus der x-Ausdehnung des gezeichneten Rechtecks. Eine Zufallsfunktion lässt die oben beschriebene Funktion auf 97% aller Knotenpaare anwenden. Bei den übrigen Fällen wird die Abstoßungskraft um den Faktor 3 erhöht.

Der Algorithmus berechnet als erstes die neuen Knotenpositionen der verbundenen Knoten mittels der Funktionen 2.16 - 2.19. Danach werden die neuen Koordinaten aller Knoten mit Hilfe der Funktionen 2.20 und 2.23 bestimmt. Umso öfter dieser Algorithmus wiederholt wird, desto mehr nähern sich verbundene Knoten an und desto weiter entfernen sich unverbundene Knoten voneinander.

Am Ende einer jeder Iteration muss geprüft werden, ob die Knoten des Graphen in einer angemessenen Geschwindigkeit bewegt wurden. War sie im Gegensatz zu der bei der vorhergehenden Iteration zu schnell, können Störungen im Layout auftreten. Um diese zu mildern, gibt es einen Dämpfungsfaktor, der für jede Iteration neu berechnet wird. So wird gewährleistet, dass das Graphlayout weder zu viel noch zu wenig auf einmal verändert wird. Die Größe dieser Dämpfungsvariable sowie ihre Werteintervalle, in denen sie mehr oder weniger stark wirkt, können nicht allgemeingültig angegeben werden. Beispielwerte, die sich für die getesten Graphen in GroIMP als sinnvoll erwiesen haben, finden sich im Implementationsteil.

2.3 Vorstellung kantenbasierter Layoutalgorithmen

2.3.1 EdgeBased

Die vorhergehend beschriebenen Layoutalgorithmen haben miteinander gemein, dass sie knotenbasiert sind, d. h. um ein übersichtliches Layout zu erhalten, berechnen die Layoutalgorithmen für die Knoten des Graphen neue Positionen, ohne die anliegenden Kanten und ihre Typen näher einzubeziehen. Kantenbasierte Layoutverfahren dagegen berechnen das neue Graphlayout anhand der Eigenschaften oder aktuellen Positionen der Kanten. Eine mögliche Art von kantenbasierten Layouts beschreiben Ganter und Wille [GW96]. Dabei gehen sie von einer partiell geordneten Menge aus und einer Projektion, die jedem Element einen reellen Vektor mit positiver y-Koordinate zuweist. Das Ergebnis nennen sie ein additives Liniendiagramm. Die Idee, die dahintersteckt, ist nicht ohne weiteres auf einfache Graphen mit Kanten, die jeweils genau einen Kantentyp haben, übertragbar. Ganter und Wille gehen von Begriffsverbänden aus. Begriffe können mehrere Merkmale haben. Der zugewiesene Richtungsvektor zu einem Begriff ist die Summe aller Vektoren der zugehörigen Merkmale. Abb. 2.1 zeigt ein additives Liniendiagramm (aus [GW96]).

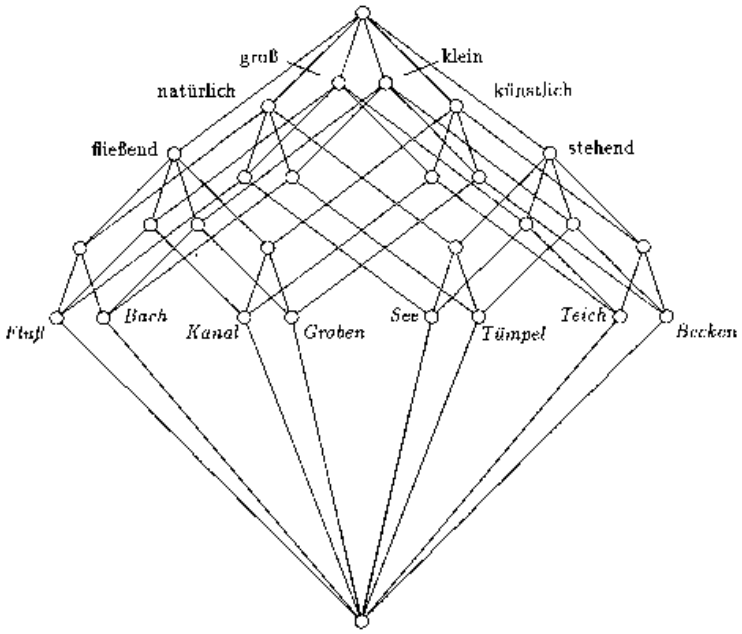

Abbildung 2.1: Additives Liniendiagramm zum Wortfeld Gewässer

Eine Abwandlung dieser Idee ist das im Folgenden beschriebene Edge-Based-Layout.

Simple Edge-Based

Das Edge-Based-Layout ordnet jedem Kantentyp eines Graphen eine Richtung, in der die Kanten des betreffenden Typs gezeichnet werden sollen, zu. Die Möglichkeiten von Richtungen an einem bestimmten Punkt im Koordinatensystem beschränken sich auf einen 180°-Bereich, wenn keine Kanten auf andere mit anderen Kantentypen gezeichnet werden sollen.

Gegeben: z .. Anzahl der Kantentypen

$$directionDistance = 180°/z \tag{2.24}$$

Es wird festgelegt, dass die 0°-Richtung direkt auf der x-Achse in positiver Richtung des Koordinatensystems des Zeichenfensters liegt, mit y-Werten gleich 0. Die Richtungen der folgenden Kantentypen sind abhängig von der Anzahl dieser innerhalb des Graphen. Bei zwei Kantentypen liegen die Kanten des ersten Typs parallel zur x-Achse, die anderen des zweiten Typs zur y-Achse (90°). Bei beispielsweise drei Typen wird der 180°-Zeichenbereich durch 3 geteilt. Kanten des zweiten Kantentyps werden in 60°, die Kanten des dritten Typs in 120°-Richtung gezeichnet.

Gegeben: $edgeNumber$ .. laufende Nummer des aktuell betrachteten Kantentyps (z. B. 0 für ersten Kantentyp, 1 für zweiten, usw.)

$$direction = edgeNumber * directionDistance \tag{2.25}$$

Normalerweise können die Positionen der Kanten eines Graphen nicht direkt verändert werden. Um sie dennoch zu verändern, müssen die Koordinaten der Quell- und Zielknoten der Kanten neu berechnet und gesetzt werden.

Um die Kanten entsprechend der ermittelten Richtungen zu zeichnen, durchläuft der Algorithmus alle Knoten beginnend von einem Knoten ohne eingehende Kanten in Tiefensuche bis hin zu den Blättern. Die Position des aktuell betrachteten Knotens ist fest (bzw. wurde eine Iteration früher gesetzt, sollte es sich nicht um die Wurzel des Graphen handeln). Die Positionen der Knoten, die mit diesem Knoten verbunden und die gleichzeitig die Zielknoten der Kante sind, ergeben sich aus einer einfachen Geradengleichung, die im folgenden dargestellt ist:

Gegeben: m .. Startknoten einer Kante e, n .. Zielknoten der Kante e, $direction$ .. Richtungsvektor des Kantentyps, dem e angehört, $edgeLength$ .. Kantenlänge, einheitlicher Wert für alle Kanten

$$(n.x, n.y) := (m.x, m.y) + edgeLength * (\cos(direction), \sin(direction)) \tag{2.26}$$

Wenn sichergestellt werden könnte, dass an jedem Knoten nur maximal eine Kante pro Kantentyp anläge, dann liefert dieser Algorithmus ein symmetrisches Graphenlayout. Bei mehr als einer Kante des gleichen Kantentyps an einem Knoten würden sich diese aber überlagern, und da der Algorithmus uniforme Kantenlängen vorschreibt, lägen zudem auch noch die Knoten übereinander. Deshalb muss der Algorithmus zunächst so abgeändert werden, dass auch das Zeichnen der Kanten mit invertierter Richtung ihres Kantentyps erlaubt wird. So werden Kanten des ersten Kantentyps an einem bestimmten Knoten ausgehend von diesem parallel zur x-Achse abwechselnd mit steigenden bzw. fallenden x-Werten gezeichnet. Sollten nun auch noch mehr als zwei Kanten desselben Kantentyps an einem Knoten vorkommen, dann werden die folgenden Kanten mit bestimmter Abweichung von der vorhergehenden Kante gezeichnet. In Funktionen ausgedrückt bedeutet dies, dass die unter 2.25 angegebene Rechnung folgendermaßen erweitert werden muss:

Gegeben: $edgeNumber$ .. laufende Nummer des aktuell betrachteten Kantentyps, $typeCounter$ .. Zähler, Nummer der aktuell betrachteten Kante des aktuell betrachteten Kantentyps am aktuell betrachteten Knoten beginnend bei 0, DIV .. ganzzahlige Division, $degreeDeviation$ .. Gradabweichung

Wenn $typeCounter$ MOD $2 == 0$, dann:

$$direction = edgeNumber * directionDistance + typeCounterDIV2 * degreeDeviation \tag{2.27}$$

andernfalls:

$$direction = 180° + (edgeNumber * directionDistance + typeCounterDIV2 * degreeDeviation) \tag{2.28}$$

Sollte auf der neu berechneten Positon bereits ein Knoten liegen, muss zur Richtung die Grad-Abweichung hinzuaddiert werden. Dieses Prüfen und Neuberechnen geschieht solange, bis ein unbesetzter Platz im Zeichenfenster gefunden wurde. Dies kann allerdings zu einem großen Problem werden, wenn der Graph über sehr viele Knoten verfügt (oder die vorhandenen Knoten recht groß gezeichnet wurden), die neuen Kantenlängen aber recht klein gewählt worden sind. Es liegt an der konkreten Implementierung, dieses Problem zu lösen.

Edge-Based2

Das Edge-Based-Layout2 kann man als Erweiterung des Simple Edge-Based-Layouts verstehen. Es untscheidet häufige und weniger häufige Kantentypen innerhalb eines Graphen und ordnet unter diesem Gesichtspunkt die Zeichenrichtungen und Kantenlängen zu. Zuerst

werden zwei Kantentypen ermittelt, die am häufigsten vorkommen. Diese beiden bekommen die waagerechte (0°, y-Koordinate = const.) bzw. senkrechte (90°, x-Koordinate = const.) Richtung als Richtungsvektoren zugeordnet. Die übrigen Kantentypen bekommen jeweils einen Richtungsvektor zwischen 0° und 90°, sowie zwischen 90° und 180° zugewiesen. Die Berechnung der Richtungen ab dem dritten Kantentyp lautet folgendermaßen:

Gegeben: $|edgeTypes|$ .. Anzahl der Kantentypen im Graph

$$directionDistance := 180°/(|edgeTypes| + 1) \tag{2.29}$$

Gegeben: DIV .. ganzzahlige Division, *counter* .. laufende Nummerierung der Kantentypen beginnend mit 1

$$direction = ((counter + 1)DIV2) * directionDistance \tag{2.30}$$

Genau wie beim Simple EdgeBased-Layout auch gibt es hier Mechanismen, um die Überlappungen von Kanten gleichen Typs an einem Knoten zu vermeiden. Analog zum bereits gezeigten Schema des abwechselnden Zeichnen der Kanten in Vektorrichtung bzw. entgegengesetzt davon und der Grad-Abweichung läuft es beim EdgeBased-Layout2 gleichermaßen ab.

Desweiteren gibt es bei diesem Layout zwei unterschiedliche Kantenlängen. Die Kanten, die den ersten beiden Typen angehören, bekommen die längere, die anderen bekommen die kürzere. So werden mögliche Knotenüberlappungen weitestgehend vermieden.

2.4 Implementierungen in GroIMP

Lädt GroIMP einen Graphen, dann wurden bisher seine Knoten zufällig auf einer räumlich begrenzten Fläche angeordnet, die unabhängig von der tatsächlichen Größe der Zeichenfläche ist. Die x- und y-Koordinaten der Knoten sind Fließkomma-Zahlen. Der Punkt (0,0) stellt die Mitte der Fläche dar. Es wurde nicht darauf geachtet, ob sich Knoten oder Kanten überlagern. Umso höher die Anzahl an Objekten, desto unübersichtlicher wurde auch die gezeichnete Graphdarstellung. Um diese Situation zu verbessern und dem Benutzer eine angenehmere Darstellung zu präsentieren, sind Layoutalgorithmen unabdingbar.

Die Grundlage vieler der im folgenden beschriebenen knotenbasierten Layoutalgorithmen boten die Graph-Darstellungsverfahren, welche in der Diplomarbeit von Dexu Zhao [ZH06] zusammengetragen und erläutert wurden. Sie wurden an die aktuellen GroIMP-Schnittstellen angepasst und verbessert und können nun in der Software auf 2D-Graphen angewendet werden. Abbildung 2.2 zeigt eine Layoutverbesserung eines Graphen in GroIMP.

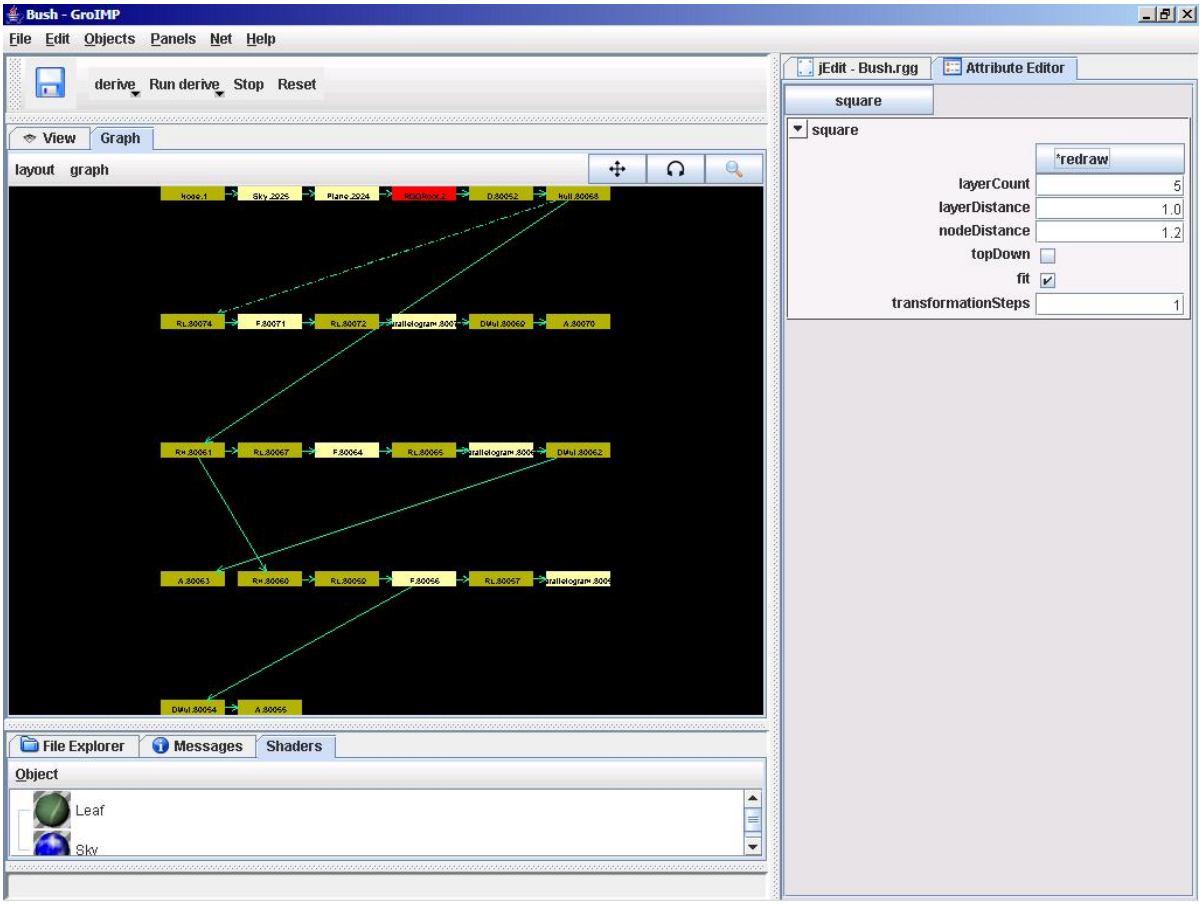

Abbildung 2.2: Auswahl und Anwendung des Square-Layouts auf einen Graphen in GroIMP

Da die Layoutalgorithmen zu den 2D-Ansichten gehören, sind sie im Package *de.grogra.imp2d.layout* zu finden.

Alle Layout-Algorithmen erben von der abstrakten Klasse *Layout.java*, die im GroIMP-Paket schon vorhanden war und nur erweitert werden musste und somit als Basis für alle zu implementierenden Graphen-Layouts verwendet wurde. Sie beinhaltet eine abstrakte Klasse namens *Algorithm*, in der in ihrer von den erbenden Klassen noch zu implementierenden Methode *layout(Node nodes)* der entsprechende anzuwendende Layout-Algorithmus ausgeführt wird. Der Benutzer kann durch eine fit-Funktion entscheiden, ob das berechnete Layout in der ursprünglichen Dimension angezeigt wird, oder ob es an die aktuelle Fenstergröße angepasst wird. Im letzteren Fall führt *Layout* eine Transformation durch. Es wird getestet, ob das Layout nur durch gleichmäßige Verschiebung aller Knoten im gesamten Zeichenfenster sichtbar gemacht werden kann. Reicht das nicht aus, wird das Graphenlayout zusätzlich verkleinert und anschließend zentriert. Sollte dagegen das Layout sehr klein sein und nicht die gesamte Fenstergröße ausfüllen, kann die fit-Funktion das Graphenlayout auch vergrößern. Die Abstände der Knoten zu anderen, sowie die Kantenlängen bleiben im gleichen Verhältnis erhalten. Desweiteren beinhaltet die Klasse *Algorithm* weitere Methoden, die das Arbeiten mit dem Graphen ermöglichen sollen.

Im Folgenden werden die implementierten Layout-Algorithmen dargestellt. Es werden ihre Eingabeparameter gezeigt, die Idee und die für das Verständnis wichtigsten Algorithmenschritte.

2.4.1 Random, Square, Circle

Alle Layout-Algorithmen in diesem Kapitel erben auf Grund ihrer geringen Komplexität direkt von der Klasse *Layout.java*.

Random

In einer benutzerdefinierten Anzahl von Iterationsschritten werden für alle Knoten des Graphen jeweils zwei Fließkomma-Zufallszahlen zwischen 0 und 1 bestimmt, die den neuen Positionskoordinaten des Knoten entsprechen bzw. mit denen festgelegt wird, um wieviel der Knoten in x- bzw. y-Richtung vom aktuellen Standpunkt aus verschoben wird.

Als Eingabeparameter fordert der Algorithmus folgende Variablen, die in Tab. 2.1 dargestellt sind.

Das Verschieben der Knotenpositionen um zufällig ermittelte Werte ist in Alg. 1 dargestellt. Einzig der Wurzelknoten wird der besseren Übersichtlichkeit halber auf einen festen

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
count	$\mathbb{N}$	1	legt fest, wie oft der Algorithmus wiederholt auf alle Knoten des Graphen angewendet werden soll
magnitude	$\mathbb{Z}$	5	Gewichtung einer Zufallszahl
startWithRandom	boolean	false	True: Knoten bekommen eine neue per Zufallsfunktion berechneten Position; False: Algorithmus verschiebt Knotenpositionen um die Zufallswerte
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewandt werden soll

Tabelle 2.1: Eingabeparameter des Random-Layouts

Platz in der oberen linken Ecke des Zeichenfensters gesetzt.

Algorithmus 1 RandomLayout

Require: *count*, *magnitude*, *nodes*, $startWithRandom$ = false
1: **for** 1 to *count* **do**
2: **for all** node n of *nodes* **do**
3: $r1$:= random(0,1)
4: $r2$:= random(0,1)
5: $n.x := n.x + (magnitude * (r1 - 0.5))$
6: $n.y := n.y + (magnitude * (r2 - 0.5))$
7: **end for**
8: **end for**

Square

Es kann zwischen einem horizontalen oder vertikalen Zeichnen ausgewählt werden. Bei der ersten Auswahlmöglichkeit wird der erste Knoten auf dem Punkt (0,0) des Zeichenfensters platziert, die nächsten ordnet der Algorithmus jedes Mal ein Stückchen weiter rechts an. Ist die maximale Anzahl der Knoten pro Zeile (beim vertikalen Zeichnen würde man eher von einer Spalte als von einer Zeile reden) erreicht, wird der folgende Knoten etwas tiefer unter dem ersten gesetzt, die folgenden dann wieder rechts von ihm, usw.

Das vertikale Zeichnen beginnt ebenfalls mit dem ersten Knoten auf (0,0), dann werden allerdings die nächsten Knoten jeweils ein bisschen tiefer angeordnet, bis diese Spalte die maximale Knotenanzahl des Gitters erreicht hat und der Algorithmus in die nächste Spalte direkt rechts von dem ersten Knoten wechselt.

Bei beiden Zeichenvarianten wird der Wurzelknoten auf der oberen linken Ecke des Zeichenfensters positioniert.

Die benötigten Eingabeparameter des Square-Layouts sind in Tab. 2.2 aufgelistet.

Alg. 2 zeigt das implementierte Square-Layoutverfahren.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
layerCount	$\mathbb{Z}$	3	Anzahl der Spalten (topDown == true) bzw. Zeilen (topDown == false), in denen die Knoten angeordnet werden sollen
layerDistance	$\mathbb{R}$	1.0	Abstand der Spalten bzw. Zeilen voneinander
nodeDistance	$\mathbb{R}$	0.6	Abstand der Knoten innerhalb einer Spalte/Zeile
topDown	boolean	false	False: Graph wird ausgehend von seiner Wurzel von links nach rechts gezeichnet true: Graph wird von oben nach unten gezeichnet
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.2: Eingabeparameter des Square-Layouts

Algorithmus 2 SquareLayout

Require: *layerCount*, *layerDistance*, *nodeDistance*, *topDown*
1: *numberOfNodesPerLayer* := number of nodes / *layerCount*
2: **if** number of nodes MOD *layerCount* != 0 **then**
3: *numberOfNodesPerLayer* := *numberOfNodesPerLayer* + 1
4: **end if**
5: *counter* := 0
6: *r* := 0
7: **for all** node *n* of *nodes* **do**
8: **if** *topDown* == *true* **then**
9: *n.x* := *r*
10: *n.y* := -*counter* * *nodeDistance*
11: **else**
12: *n.x* := -*counter* * *nodeDistance*
13: *n.y* := -*r*
14: **end if**
15: **if** *counter* + 1 == *numberOfNodesPerLayer* **then**
16: *r* := *r* + *layerDistance*
17: *counter* := −1
18: **end if**
19: *counter* := *counter* +1
20: **end for**

Circle

Das Layout ordnet die Knoten eines Graphen auf Kreise an. Der Radius r gibt den Abstand vom Mittelpunkt des Kreises zu den Knoten an. Sollen mehrere Kreise ineinander gezeichnet werden, dann wird die maximal mögliche Anzahl von Knoten pro Kreis berechnet, um bei Erreichen dieses Maximums in den nächsten zu wechseln. Dann vergrößert sich der Radius natürlich jedes Mal dabei.

Um dem Benutzer das Wiederfinden des ersten Knoten zu erleichern, wird dieser immer auf den innersten Kreis an der obersten Position gesetzt.

Tab. 2.3 zeigt die benötigten Parameter zur Berechnung des Circle-Layouts.

Wie dieses Layoutverfahren im Detail abläuft, zeigt Alg. 3. Zur Variable *maximumHeight* ist anzumerken, dass Knoten eines Graphen eigentlich keine Höhe besitzen. Auf der Zeichenfläche werden sie allerdings durch geometrische Figuren, wie z. B. Kreise, Rechtecke,

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
layerCount	$\mathbb{N}$	2	Anzahl der Kreise im Layout
layerDistance	$\mathbb{R}$	1.0	Abstand der Kreise voneinander
nodeDistance	$\mathbb{R}$	1.0	Abstand der Knoten voneinander
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.3: Eingabeparameter des Circle-Layouts

dargestellt und diese besitzen durchaus eine Höhe. In dem Fall von *maximumHeight* ist die größte y-Ausdehnung der gezeichneten Symbole, die die Knoten repräsentieren, gemeint.

Die Funktion zum Berechnen der Knotenanzahl pro Kreis ($numberOfNodesPerLayer$) ist identisch mit der Berechnung der Knoten pro Zeile/Spalte im Square-Layout (siehe Funktion 2.3).

Algorithmus 3 CircleLayout

Require: *layerCount*, *layerDistance*, *nodeDistance*, *nodes*
1: *maximumHeight* := biggest value of height of a drawed node
2: $numberOfNodesPerLayer$:= number of nodes / *layerCount*
3: **if** number of nodes MOD *layerCount* != 0 **then**
4: $numberOfNodesPerLayer = numberOfNodesPerLayer + 1$
5: **end if**
6: radius r := $numberOfNodesPerLayer$ * *maximumHeight* / π
7: angle *phi* := 2 * π / $numberOfNodesPerLayer$
8: $currentNodeNumber := 0$
9: **for all** node n of *nodes* **do**
10: $currentNodeNumber := currentNodeNumber + 1$
11: **if** *layerCount* > 1 AND $currentNodeNumber$ MOD ($numberOfNodesPerLayer$ + 1) == 0 **then**
12: $r := r + layerDistance$
13: $currentNodeNumber := 0$
14: **end if**
15: $n.x := r * \cos((currentNodeNumber - 1) * phi)$
16: $n.y := r * \sin((currentNodeNumber - 1) * phi)$
17: **end for**

2.4.2 Tree, Sugiyama

Es gibt eine Klasse *GraphUtilities* mit Hilfsfunktionen speziell für hierarchische Graphenlayouts, die u. a. die Wurzeln der Graphen zurückgeben, alle verbundenen oder unverbundenen Knoten zurückliefern oder Knoten bzw. Kanten als besucht markieren kann.

Sie wird auch für die kantenbasierten Layoutverfahren benutzt, da diese die Knoten, ähnlich wie das Tree-Layout, in Tiefensuche durchlaufen.

In diesem Kapitel werden Graphen synonym als Bäume bezeichnet. Bäume sind zusammenhängende Graphen, in denen keine Zyklen vorkommen dürfen. Jeder Baum mit n Knoten hat genau n-1 Kanten [BR99]. Bäume können sich sehr gut als hierarchische Strukturen modellieren lassen.

Tree

Der Aufbau eines Baumes wurde mit der Tiefensuche implementiert. Tab. 2.4 zeigt die benötigten Eingabeparameter.

Name	**Datentyp**	**Vorgabewert**	**Beschreibung**
		Benutzerspezifisch	
minDistanceX	$\mathbb{R}$	1.0	minimaler horizontaler Abstand der Knoten zueinander
minDistanceY	$\mathbb{R}$	1.0	minimaler vertikaler Abstand der Knoten zueinander
topDown	boolean	true	True: Graph wird von oben nach unten gezeichnet; False: Graph wird von links nach rechts gezeichnet
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.4: Eingabeparameter des Tree-Layouts

Wie man anhand der Eingabeparameter sieht, lässt sich der Baum entweder von links nach rechts oder von unten nach oben zeichnen. Im Folgenden wird die Berechnung des horizontalen Zeichnens näher erläutert. Das vertikale Zeichnen ist fast analog, es reicht meistens, die errechneten x- und y-Positionswerte der Knoten zu vertauschen bzw. das durch das horizontale Zeichnen erhaltene Layout um 90° zu drehen.

Ausgehend von den Wurzelknoten wird jeder verbundene Knoten besucht. Dabei wird in die Tiefe gegangen; erst wenn der aktuell betrachtete Knoten ein Blatt (= keine ausgehenden Kanten) ist, wandert der Algorithmus eine Stufe höher und besucht seinen Geschwisterknoten. Nach Abarbeitung aller Geschwisterknoten und ihrer Kinder steigt der Algorithmus wieder eine Stufe höher usw. Damit das Verfahren einmal besuchte Knoten nicht wiederholt durchläuft, werden abgearbeitete Knoten markiert.

Beim horizontalen Zeichnen werden zuerst die neuen x-Koordinaten aller Knoten berechnet, d. h. es wird ermittelt, welcher Knoten auf welcher Hierarchieschicht liegt (calcXCoord). Erst danach werden den Knoten die neuen y-Werte zugewiesen, die für den Platz der Knoten innerhalb einer Schicht stehen (calcYCoord). Siehe Alg. 4.

Algorithmus 4 TreeLayout

```
Require: minDistanceX, minDistanceY
 1: x := minDistanceX
 2: y := minDistanceY
 3: for all Rootnodes r do
 4:     calcXCoord(r, x)
 5:     tempY := calcYCoord(r, y)
 6:     if tempY >= 0 then
 7:         y := tempY
 8:     end if
 9: end for
```

Das Durchlaufen der Knoten mit der Tiefensuche, sowie die Berechnung der x-Koordinaten (vorausgesetzt ein Knoten wurde noch nicht besucht und hat demnach noch keine neue x-Koordinate erhalten) wird durch rekursives Aufrufen der Methode calcXCoord für jeden Knoten n und x-Koordinate x erreicht, welche in Alg. 5 dargestellt ist.

Algorithmus 5 TreeLayout: Methode void calcXCoord

```
Require: minDistanceX, minDistanceY, Node n, double x
 1: n.x := x
 2: n.y := n.layoutVar.y
 3: x := x + width of drawed n + minDistanceX
 4: n.layoutVar.x := n.x
 5: n.layoutVar.y := n.y
 6: for all edge e of n do
 7:     if target node of e != n then
 8:         calcXCoord( target node of e, x)
 9:     else
10:         calcXCoord(source node of e, x)
11:     end if
12: end for
```

In den Variablen $layoutVar.x$ bzw. $layoutVar.y$, die jeder Knoten besitzt, können Zwischenwerte der berechneten Positionen gespeichert werden. Vor Beginn des erstes Aufrufs der Funktion *calcXCoord* werden diese Variablen auf 0 gesetzt.

Anschließend müssen noch für jeden Knoten n die neuen y-Koordinaten y berechnet werden. Alg. 6 zeigt den Ablauf.

Algorithmus 6 TreeLayout: Funktion double calcYCoord

```
Require: minDistanceX, minDistanceY, Node n, double y
 1: newY := y
 2: node node1 := node2 := null
 3: for all edge e of n do
 4:     tmpY := 0
 5:     node tmpNode := null
 6:     if target node of e != n then
 7:         tmpY := calcYCoord(target node of e, newY)
 8:         tmpNode := target node of e
 9:     else
10:         tmpY := calcYCoord(source node of e, newY)
11:         tmpNode := source node of e
12:     end if
13:     if tmpY >= 0 then
14:         newY := tmpY
15:         if node1 == null then
16:             node1 := tmpNode
17:         else
18:             node2 := tmpNode
19:         end if
20:     end if
21: end for
22: if node1 == null then
23:     n.y := newY
24:     n.layoutVar.x := n.x
25:     n.layoutVar.y := newY
26:     newY := newY + height of drawed node n + minDistanceY
27: else
28:     if node2 == null then
29:         tmpY := 0
30:         n.y := node1.y
31:         tmpY := n.y + height of drawed node n + minDistanceY
32:         n.layoutVar.x := n.x
33:         n.layoutVar.y := n.y
34:         if tmpY > newY then
35:             newY := tmpY
36:         end if
37:     else
38:         tmpY := 0
39:         n.y := (node1.y + node2.y)/2
40:         tmpY := ((node1.y + node2.y)/2) + height of drawed node n + minDistanceY
41:         n.layoutVar.x := n.x
42:         n.layoutVar.y := n.y
43:         if tmpY > newY then
44:             newY := tmpY
45:         end if
46:     end if
47: end if
48: return newY
```

Sugiyama

Eine Übersicht der Eingabeparameter des Sugiyama-Layouts ist in Abb. 2.5 dargestellt.

Der in Alg. 7 dargestellte Algorithmus zeigt die Berechnung des Sugiyama-Layouts. Zuerst ordnet der Sugiyama-Algorithmus von den Wurzeln des Graphen ausgehende Knoten auf Schichten an (*fillLevels*, siehe Alg. 8). Anschließend werden die Kantenüberkreuzungen minimiert und eine Ordnung der Knoten innerhalb der Schichten festgelegt (*moveToBarycenter*, siehe Alg. 9). Dies geschieht nach dem im Abschnitt 2.2.3 dargestellten Prinzip. Am

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
topDown	boolean	false	True: Zeichnen des Graphen von oben nach unten; False: Zeichnen von links nach rechts
minDistanceX	$\mathbb{R}$	1.0	minimaler horizontaler Abstand der Knoten zueinander
minDistanceY	$\mathbb{R}$	2.0	minmaler vertikaler Abstand der Knoten zueinander
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.5: Eingabeparameter des Sugiyama-Layouts

Schluss werden zu diesen Ordnungszahlen horizontale und vertikale Abstände zu den anderen Knoten hinzuaddiert, welche dann die neuen Koordinaten bilden.

Algorithmus 7 SugiyamaLayout

```
Require: topDown, minDistanceX, minDistanceY
levels := ()
maxBounds := (biggest height value of a drawed node, biggest width value of a drawed node)
minBounds := (smallest height value of a drawed node, smallest width value of a drawed node)
for all Rootnodes r do
    fillLevels(levels, 0, r)
end for
movementsCurrentLoop := -1
while movementsCurrentLoop != 0 do
    movementsCurrentLoop := 0
    for all levels fLevel do
        movementsCurrentLoop += moveToBarycenter(levels, level)
    end for
end while
for rowCount = 0 to size of levels − 1 do
    for all node n in levels(rowCount) do
        n.x := (min.x + minDistance.x ∗ ((topDown)?n.gridPosition : rowCount))
        n.y := (min.y + minDistance.y * ((topDown) ? rowCount : n.gridPosition))
    end for
end for
```

Die Funktion *fillLevels* wird für jede Wurzel eines Graphen aufgerufen. Durch Selbstaufrufe durchläuft sie in Tiefensuche jeden Knoten des Graphen und ordnet diese auf Schichten (= *levels*) an. Jeder durchlaufene Knoten wird markiert, um sicherzugehen, dass er kein zweites Mal betrachtet wird. Mit *isAccessed* kann man jeden Knoten püfen, ob er besucht wurde (*true*) oder nicht (*false*).

Die Wurzeln gehören zur Schicht 0, ihre direkten Kinder zur Schicht 1; die Kinder dieser Kinder werden auf der Schicht 2 hintereinander angeordnet, usw. Die Reihenfolge der Knoten innerhalb einer Schicht ist zur Zeit noch unwichtig, da erst im Anschluss der Minimierungsalgorithmus der Kantenüberkreuzungen ausgeführt wird, welcher dann eine Ordnung vorgibt. Für diesen werden die Knoten jetzt schon in $NodeWrapper$ gepackt. Jeder $NodeWrapper$ enthält einen Knoten sowie Informationen über seine Schichtzugehörigkeit und Reihenfolge in dieser. Alg. 8 zeigt die wesentlichen Ablaufschritte der Methode $fillLevels$.

Algorithmus 8 SugiyamaLayout: Methode void fillLevels

Vector *levels*, int *level*, Node *n*

```
1: if size of levels == level then
2:     levels += (level, new ArrayList())
3: end if
4: if nodeTemp.isAccessed = false then
5:     return
6: end if
7: nodeTemp.isAccessed := false
8: currentLevel := levels.get(level)
9: numberForTheEntry := size of currentLevel
10: nodeWrapper := (level, numberForTheEntry, n)
11: currentLevel += nodeWrapper
12: for all edge e of n do
13:     if e.isAccessed == false then
14:         continue
15:     end if
16:     e.isAccessed := false
17:     m := node connected with n via e
18:     fillLevels(levels, (level + 1), m)
19: end for
```

Das Minimieren der Kantenüberkreuzungen erfolgt mit der Barycenter-Methode (siehe Fkt. 2.4). Der implementierte Algorithmus ist in Alg. 9 dargestellt. Die zu jedem *NodeWrapper* zugehörige Variable *gridPosition* gibt die aktuelle und nach der Berechnung auch zukünftige Position des Knoten innerhalb einer Schicht wieder.

Algorithmus 9 Sugiyama: Funktion int moveToBarycenter

Vector *levels*, int *level*, Node *n*)

```
1: movements := 0
2: currentLevel := levels.get(level)
3: for currentIndexInTheLevel := 0 to currentIndexInTheLevel < size of currentLevel do
4:     nodeWrapper := currentLevel.get(currentIndexInTheLevel)
5:     gridPositionsSum := 0
6:     countNodes := 0
7:     for all edge e of node n in nodeWrapper do
8:         gridPositionsSum += neighborWrapper.gridPosition
9:         countNodes := countNodes +1
10:    end for
11:    if countNodes > 0 then
12:        nodeWrapper.gridPosition := gridPositionsSum / countNodes
13:    end if
14: end for
15: return movements
```

2.4.3 Spring, Eades, FruchtermanReingold, DavidsonHarel

Die im Folgenden beschriebenen kräftebasierten Layout-Algorithmen erben von der Klasse *ForceBasedLayout.java*, die die Kräftewerte der einzelnen Knoten berechnet. Die eigentlichen Algorithmen werden in den erbenden Klassen definiert. Diese müssen folgende Methoden implementieren, auf die *ForceBasedLayout.java* zugreift:

Methode	**Beschreibung**
void computeForce (Node s, Node t, Vector2f $force$)	Berechnung der abstoßenden Kraft zwischen zwei Knoten s und t, das Ergebnis wird in $force$ gespeichert
void computeForce (Edge e, Vector2f $force$)	Berechnung der anziehenden Kraft zwischen den Knoten, die durch die Kante e miteinander verbunden sind, das Ergebnis wird in $force$ gespeichert
void setRandomPosition(Node n, Random rnd)	Berechnung einer Zufallsposition des übergebenen Knoten n, auf die er vor Ausführung des Layout-Algorithmus gesetzt wird, falls $startWithRandom ==$ true

Tabelle 2.6: Methoden der kräftebasierten Algorithmen

Für jeden kräftebasierten Layout-Algorithmus können die Werte, die in Tabelle 2.7 dargestellt sind, verändert werden.

Name	**Datentyp**	**Vorgabewert**	**Beschreibung**
		Benutzerspezifisch	
relaxation	$\mathbb{R}$	0.0378	Entspannungskoeffizient
accuracy	$\mathbb{R}$	0.01	Die anzustrebende Genauigkeit, die vom Algorithmus erreicht werden soll
count	$\mathbb{Z}$	100	Maximale Anzahl der auszuführenden Iterationen des Layout-Algorithmus
startWithRandom	boolean	true	True: Knoten bekommen vor Ausführung des Layout-Algorithmus eine per Zufallsfunktion berechnete Position; False: Layout-Algorithmus nimmt bei Beginn der Ausführung aktuelle Knotenposition als Grundlage

Tabelle 2.7: Eingabeparameter aller kräftebasierten Layout-Algorithmen

Alg. 10 zeigt das Grundgerüst dieser Klasse. Anzumerken ist hierbei, dass jedem Knoten eines Graphen neben seinen Positionskoordinaten auch andere Werte zugeordnet werden können. Besonders für diesen Algorithmus wichtig ist $layoutVar$. Diese Vektor-Variable speichert neu berechnete Koordinatendaten, weil es Algorithmen gibt, die bis zum vollständigen Durchlauf mit den alten Positionswerten rechnen, bei denen aber die neuen nebenbei schrittweise verändert werden müssen. Erst am Ende des Algorithmus werden diese Werte mit dem Entspannungskoeffizienten multipliziert und den echten x- und y-Werten zugewiesen.

Der Algorithmus bricht ab, wenn entweder der gerade noch akzeptable Differenzwert *accuracy* unterschritten oder wenn der Algorithmus *count*-mal durchlaufen wurde.

Algorithmus 10 Kräftebasierter Algorithmus

```
Require: relaxation, accuracy, count, startWithRandom
 1: for 0 to count do
 2:     sumForce := (0, 0)
 3:     tmpForce := (0, 0)
 4:     maxForce := 0
 5:     Set layoutVar of all nodes := (0, 0)
 6:     for all node n of nodes do
 7:         for all node m of nodes do
 8:             if n != m then
 9:                 computeForce (n, m, sumForce)
10:                 if n has edge e to m then
11:                     computeForce (e, tmpForce)
12:                     sumForce := sumForce + tmpForce
13:                 end if
14:                 if m has edge e to n then
15:                     computeForce (e, tmpForce)
16:                     sumForce := sumForce − tmpForce
17:                 end if
18:                 n.layoutVar := n.layoutVar − sumForce
19:                 m.layoutVar := m.layoutVar + sumForce
20:             end if
21:         end for
22:     end for
23:     maxForce := biggest value of (|layoutVar.x of a node|, |layoutVar.y of a node|, maxForce)
24:     n.x := n.x + relaxation ∗ n.layoutVar.x
25:     n.y := n.y + relaxation ∗ n.layoutVar.y
26:     if maxForce < accuracy then
27:         break
28:     end if
29: end for
```

Nachdem für alle Knoten neue Positionswerte berechnet wurden, wird der Graph im Ganzen auf der Zeichenfläche so verschoben, dass der Wurzelknoten, der in GroIMP *Node.1* heißt, in der Mitte auf Punkt (0,0) liegt. So kann dieser Knoten im Layout schnell wiedergefunden werden, das neu berechnete Layout wird dadurch nicht berührt. So ergeben alle kräftebasierten Layoutalgorithmen harmonische Graphdarstellungen.

Spring

Knotenpaare, die durch eine Kante miteinander verbunden sind, ziehen sich an, unverbundene Paare stoßen sich ab. In der GroIMP-Implementierung dieses Layoutverfahrens wurden dafür zwei einfache Funktionen definiert. Die Eingabeparameter werden in Tabelle 2.8 gezeigt.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
attraction	$\mathbb{R}$	1.0	Anziehungskonstante
repulsion	$\mathbb{R}$	1.0	Abstoßungskonstante
		Weitere	
nodes	List		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.8: Eingabeparameter des Spring-Layouts

Alg. 11 zeigt die Implementierung zur Berechnung der abstoßenden Kraft je Knotenpaar. In Abhängigkeit von der vom Benutzer gewählten Abstoßungskonstante sowie der aktuellen Entfernung der Knoten voneinander wird die Abstoßungskraft ermittelt.

Algorithmus 11 SpringLayout: Berechnung abstoßende Kraft zwischen zwei Knoten s und t

Require: *repulsion*
1: $force := t - s$
2: $d = |force|$
3: $force := repulsion * force/d^3$

Alg. 12 stellt die Berechnung der anziehenden Kraft zwischen je zwei verbundenen Knoten dar. Hierfür werden die einstellbare Anziehungskonstante und die Gewichtung der Kante gebraucht, die die aktuell betrachteten Knoten miteinander verbindet. Defaultmäßig besitzt jede Kante dieselbe Gewichtung 1. Die Ergebnisse werden als Vektoren gespeichert.

Algorithmus 12 SpringLayout: Berechnung der anziehenden Kraft zwischen zwei Knoten s und t mit Kante e

Require: *attraction*
1: $force := t - s$
2: $force := force * (attraction/e.weight - |force|)/|force|$

Eades

Die beiden Funktionen zur Berechnung der Anziehungs- bzw. Abstoßungskräfte sind im Gegensatz zum Spring-Layout ein wenig komplexer, die Eingabeparameter dagegen weitestgehend gleich. Zur besseren Übersichtlichkeit kann hier der Benutzer die minimalen Abstände zu anderen Knoten festlegen. Konstanten zur Abstoßungs- und Anziehungskraft wurden ebenfalls definiert (Tab. 2.9).

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
repellingConst	$\mathbb{R}$	1.0	Abstoßungskonstante
springConst	$\mathbb{R}$	4.0	Anziehungskonstante
minDistanceX	$\mathbb{R}$	0.4	minimaler x-Abstand der Knoten zueinander
minDistanceY	$\mathbb{R}$	0.4	minimaler y-Abstand der Knoten zueinander
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.9: Eingabeparameter des Eades-Layouts

Die Funktionen zur Berechnung der Kräfte sind in Alg. 13 (Berechnung der Abstoßungskraft) und Alg. 14 (Berechnung der Anziehungskraft) dargestellt.

Algorithmus 13 EadesLayout: Berechnung abstoßende Kraft zwischen zwei Knoten s und t

Require: *repellingConst*
1: *distance* := distance from t to s
2: **if** *distance* == 0 **then**
3: *distance* := 0.001
4: **end if**
5: $factor := repellingConst \ / \ distance$
6: $force := t - s$
7: $force := (1/distance) * force$
8: $force := factor * force$

Algorithmus 14 EadesLayout: Berechnung der anziehenden Kraft zwischen zwei Knoten s und t mit Kante e

Require: *minDistanceX*, *minDistanceY*, *springConst*
1: *distance* := distance from s to t
2: **if** *distance* == 0 **then**
3: *distance* := 0.001
4: **end if**
5: $springLength := \sqrt{(minDistanceX * minDistanceX) + (minDistanceY * minDistanceY)}$
6: $factor := \log(distance/springLength) * springConst$
7: $force := s - t$
8: $force := 1/distance * force$
9: $force := factor * force$

Unter der Differenz zweier Knoten versteht man das Subtrahieren des x-Positionswertes des zweiten Knoten vom dem des ersten, sowie die Subtraktion der beiden y-Koordinaten.

FruchtermanReingold

Dieses Layoutverfahren erweitert das Eades-Layout, indem hier die Größe des Zeichenfensters berücksichtigt wird und von einer idealen Entfernung *id* die Rede ist, die sowohl für die Berechnung der Anziehungs- als auch für die der Abstoßungskräfte von Bedeutung ist:

Gegeben: $frameX$, $frameY$ .. Breite und Höhe des Zeichenfensters, $scaleFactor$ .. Skalierungsfaktor.

$$id = \sqrt{|(frameX * frameY)/\text{number of nodes}| * scaleFactor} \tag{2.31}$$

Daher verändert sich die Menge der Eingabeparameter im Gegensatz zum Eades-Layout um einige Variablen, wie in Tab. 2.10 dargestellt.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
$scaleFactor$	$\mathbb{R}$	1.0	Skalierungsfaktor
$frameX$	$\mathbb{R}$	10.0	x-Wert des Fensters, in dem der Graph gezeichnet werden soll
$frameY$	$\mathbb{R}$	10.0	y-Wert des Fensters, in dem der Graph gezeichnet werden soll
		Weitere	
$nodes$	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.10: Eingabeparameter des FruchtermanReingold-Layouts

So sieht die Berechnung der Abstoßungskräfte zwischen den Knotenpaaren eines Graphen wie in Alg. 15 aus.

Algorithmus 15 FruchtermanReingoldLayout: Berechnung abstoßende Kraft zwischen zwei Knoten s und t

Require: id
1: $distance :=$ distance from s to t
2: **if** $distance == 0$ **then**
3: $\quad distance := 0.001$
4: **end if**
5: $id := \sqrt{|frameX * frameY/numberOfNodes| * scaleFactor}$
6: $factor := id^2/distance$
7: $force := t - s$
8: $force := (1/distance) * force$
9: $force := factor * force$

Alg.16 zeigt die Implementierung der Berechnung der Anziehungskräfte von je zwei verbundenen Knoten.

Algorithmus 16 FruchtermanReingoldLayout: Berechnung anziehende Kraft zwischen zwei Knoten s und t mit Kante e

Require: id
1: $distance :=$ distance from s to t
2: **if** $distance == 0$ **then**
3: $\quad distance := 0.001$
4: **end if**
5: $id := \sqrt{|frameX * frameY/numberOfNodes| * scaleFactor}$
6: $factor := distance^2/id$
7: $force := s - t$
8: $force := (1/distance) * force$
9: $force := factor * force$

DavidsonHarel

Zur Berechnung der Kraft sind die in Tab. 2.11 gelisteten Eingabeparameter erforderlich.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
attractionFactor	$\mathbb{R}$	-0.008	Anziehungskonstante
repulsionFactor	$\mathbb{R}$	0.4	Abstoßungskonstante
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.11: Eingabeparameter des DavidsonHarel-Layouts

Die implementierte Version des Davidson-Harel-Layouts passt sich ganz den kräftebasierten Layout-Algorithmen an, was zur Folge hat, dass lediglich zwei Kriterien zur Layoutdarstellung berücksichtigt werden können: Die Formel zur *uniform distribution* findet sich in der Methode zur Berechnung der abstoßenden Kräfte von Knotenpaaren wieder (Alg. 17), die *fixed edge length* wurde zum Ermitteln der Anziehungskräfte je zwei verbundener Knoten (Alg. 18) herangezogen.

Algorithmus 17 DavidsonHarelLayout: Berechnung abstoßende Kraft zwischen zwei Knoten s und t

Require: $repulsionFactor$
1: $distance$:= distance from s to t
2: **if** $distance == 0$ **then**
3: $\quad distance := 0.001$
4: **end if**
5: $factor := repulsionFactor$ / $distance$
6: $force := t - s$
7: **if** $distance > 0$ **then**
8: $\quad force := (1/distance) * force$
9: **end if**
10: $force := factor * force$

Algorithmus 18 DavidsonHarelLayout: Berechnung anziehende Kraft zwischen zwei Knoten s und t mit Kante e

Require: $attractionFactor$
1: $distance$:= distance from s to t
2: **if** $distance == 0$ **then**
3: $\quad distance := 0.001$
4: **end if**
5: $factor := attractionFactor * \log(distance^2)$
6: $force := s - t$
7: **if** $distance > 0$ **then**
8: $\quad force := (1/distance) * force$
9: **end if**
10: $force := factor * force$

Abbruchbedingung ist hier, wie bei den anderen kräftebasierten Layoutalgorithmen auch,

das Unterschreiten des *accuracy*-Wertes bzw. der maximal gewünschten Anzahl von Iterationen.

Die ebenfalls geforderte Berücksichtigung der Nähe der Knoten zu den Rändern der Zeichenfläche ist durch die bereits beschriebene, und für alle Layoutalgorithmen vorhandene, fit-Funktion abgegolten.

2.4.4 EnergyModel, Touch

EnergyModel-Layout

Alle Eingabeparameter des EnergyModel-Layouts zeigt Tab. 2.12. Besonders wichtig sind die Variablen *gravitationFactor* und *attractionExponent*. Mit ihnen berechnet später der eigentliche Layout-Algorithmus *MinimizerPolyLogBarnesHut* die energieärmsten Positionen der Knoten, wie im Abschnitt 2.2.4 dargestellt.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
gravitationFactor	$\mathbb{R}$	0.15	Schwerkraft, Stärke der Anziehung zum Barycenter
attractionExponent	$\mathbb{R}$	3.0	Anziehungsenergie, das r des PolyLogR-Modells
iterations	$\mathbb{R}$	100	Anzahl der auszuführenden Iterationen
minDistanceX	$\mathbb{R}$	2.5	minimaler waagerechter Abstand der verbundenen Knoten voneinander
minDistanceY	$\mathbb{R}$	2.5	minimaler senkrechter Abstand der verbundenen Knoten voneinander
unconnectedNodeX	$\mathbb{R}$	0.5	minimaler waagerechter Abstand der unverbundenen Knoten voneinander
startWithRandom	boolean	true	True: Knoten bekommen vor Ausführung des Layout-Algorithmus eine per Zufallsfunktion berechnete Position; False: Layout-Algorithmus nimmt bei Beginn der Ausführung aktuelle Knotenpositionen als Grundlage
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.12: Eingabeparameter des EnergyModel-Layouts

Der im Folgenden dargestellte Alg. 19 zeigt die Struktur des Energie-Layoutverfahrens. Es wird unterschieden zwischen unverbundenen und verbundenen Knoten. Während erstere sofort eine neue Position bekommen, werden letztere und ihre zugehörigen Kanten gesammelt und dem *MinimizerLogBarnesHut-Algorithmus* übergeben. Dieser berechnet in endlichen Iterationsschritten die neuen Koordinaten aller Knoten, welche im Anschluss neu gezeichnet werden.

Wie der *MinimizerLogBarnes-Hut-Algorithmus* im Detail funktioniert, wird in [NO03] genau beschrieben. Demnach wird ein Graph im zweidimensionalen Raum iterationsweise in Quadtrees unterteilt. Anhand von einigen Funktionen wird geprüft, ob die neu berechneten Positionen das globale Energieminimum des Graphen minimieren, was es zu erreichen gilt. Die Energie eines einzelnen Knoten m ergibt sich aus der Summe der abstoßenden Kraft des Knotens zum Quadtree, seiner anziehenden Kräfte auf ihn, sowie der Gravitationskraft. Der Algorithmus in Pseudocode-Darstellung ist in Alg. 20 zu finden.

Algorithmus 19 EnergyModelLayout

Require: $gravitationFactor$, $attractionExponent$, $count$, $minDistanceX$, $minDistanceY$, $unconnectedNodeX$
1: $unconnectedCounter := 0$
2: $unconnectedNodeY :=$ biggest value of width of a drawed node
3: $nodePositions = ()$, $edgeWeights = ()$
4: **for all** node n of $nodes$ **do**
5: **if** n has no edges **then**
6: $n.x := unconnectedNodeX$
7: $n.y := unconnectedCounter * fUnconnectedNodeY$
8: $unconnectedCounter := unconnectedCounter + 1$
9: continue
10: **end if**
11: $nodePositions := nodePositions + (n, n.x, n.y)$
12: **end for**
13: $edgeCounter := 0$
14: $distance := 0$
15: **for all** edge e **do**
16: $edgeWeights := edgeWeights + (e, e.weight)$
17: $distance := distance +$ dist(e)
18: $edgeCounter := edgeCounter + 1$
19: **end for**
20: $nodePositions :=$ MinimizerPolyLogBarnesHut($connectedNodesSize$, $edgeWeights$, $nodePositions$, $attractionExponent$, $gravitationFactor$, $count$)
21: **for all** node n of $nodes$ **do**
22: $(n.layoutVarX, n.layoutVarY) := nodePositions(n)$
23: **end for**
24: **for all** node n of $nodes$ **do**
25: $n.x := n.layoutVarX * minDistanceX$
26: $n.y := n.layoutVarY * minDistanceY$
27: **end for**

Da der Algorithmus ursprünglich für dreidimensionale Graphen entwickelt wurde, arbeitet man hier mit *Octtrees*. Daher besitzen alle Positionsangaben jeweils drei Werte. Für zweidimensionale Graphen kann man die dritte Koordinate ignorieren.

Algorithmus 20 MinimizerPolyLogBarnesHut-Algorithmus

Require: $connectedNodesSize$, $edgeWeights$, $nodePositions$, $attractionExponent$, $gravitationFactor$, $iterations$

$repuFactor$:= (sum of all edge weights) / (number of nodes * (number of nodes −1)) / 2
$finalRepuFactor := repuFactor$
$octTree$ = new OctTree(0, pos[0], 1.0f, minPos, maxPos);
$energySum := 0$
for all node m of nodes **do**
    $energySum$ += $getEnergy(n)$
end for
$oldPos$:= new float[3]
$bestDir$:= new float[3]
for $i := 0$ to $iterations$ **do**
    computeBaryCenter()
    buildOctTree()
    if i / $repuStrategy.length < (nrIterations - 20)/repuStrategy.length$ **then**
        $repuFactor := finalRepuFactor$ * repuStrategy($step$ MOD $repuStrategy$.length)^{attrExponent}
    else
        $repuFactor := finalRepuFactor$
    end if
    $energySum := 0$
    for all node m of nodes **do**
        $oldEnergy := getEnergy(i)$
        $getDirection(i, bestDir)$
        oldPos[0] = pos[i][0]; oldPos[1] = pos[i][1]; oldPos[2] = pos[i][2];
        $bestEnergy := oldEnergy$
        $bestMultiple := 0$
        $bestDir$[0] /= 32; bestDir[1] /= 32; bestDir[2] /= 32;
        for $multiple$:= 32 to $multiple$ >= 1 AND ($bestMultiple$ == 0 OR $bestMultiple/2$ == $multiple$) **do**
            $multiple := multiple/2$
            $pos[i][0] = oldPos[0] + bestDir[0] * multiple$
            $pos[i][1] = oldPos[1] + bestDir[1] * multiple$
            $pos[i][2] = oldPos[2] + bestDir[2] * multiple$
            $curEnergy := getEnergy(m)$
            if $curEnergy < bestEnergy$ **then**
                $bestEnergy := curEnergy$
                $bestMultiple := multiple$
            end if
        end for
        for $multiple$:= 64 to ($multiple$ <= 128 AND $bestMultiple == multiple/2$) **do**
            $multiple := multiple$ * 2
            $pos[i][0] := oldPos[0] + bestDir[0] * multiple$
            $pos[i][1] := oldPos[1] + bestDir[1] * multiple$
            $pos[i][2] := oldPos[2] + bestDir[2] * multiple$
            $curEnergy$:= getEnergy(m)
            if $curEnergy < bestEnergy$ **then**
                $bestEnergy := curEnergy$
                $bestMultiple := multiple$
            end if
        end for
        $pos[i][0] := oldPos[0] + bestDir[0] * bestMultiple$
        $pos[i][1] := oldPos[1] + bestDir[1] * bestMultiple$
        $pos[i][2] := oldPos[2] + bestDir[2] * bestMultiple$
        if $bestMultiple > 0$ **then**
            $octTree$.moveNode($oldPos, pos[i], 1.0f$)
        end if
        $energySum := energySum + bestEnergy$
    end for
end for

TouchLayout

Die Eingabeparameter des Touch-Layouts sind in Tab. 2.13 aufgelistet.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
damper	$\mathbb{R}$	1.0	Konstante
count	$\mathbb{N}$	100	Anzahl der auszuführenden Iterationen des Algorithmus
rigidity	$\mathbb{R}$	0.00005	Anziehungskonstante
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll
$maxMotion$	$\mathbb{R}$	0	Anzahl ausgeführter Bewegungen
$lastMaxMotion$	$\mathbb{R}$	0	Anzahl ausgeführter Bewegungen bei vorheriger Iteration

Tabelle 2.13: Eingabeparameter des Touch-Layouts

Das Touch-Layout folgt nicht streng einer bestimmten Kategorie von Algorithmus. In einer vom Benutzer einstellbaren Anzahl von Iterationsschritten berechnet es neue Knotenpositionen zwischen allen verbundenen Knotenpaaren (*relaxEdges*), neue Koordinaten von allen (verbundenen und unverbundenen) Knotenpaaren (*avoidLabels*), sowie führt eine allgemeine Änderung aller Knotenpositionen durch (*moveNodes*). Die Variablen *height* und *width* stehen dabei für die Höhe bzw. Breite des geometrischen Symbols, das den Knoten repräsentiert.

Die Methode *relaxEdges* ist in Alg. 21 dargestellt.

Algorithmus 21 TouchLayout: Methode void relaxEdges

Require: $rigidity$
1: **for all** edge e **do**
2: $dx := (e.target.x - e.source.x) * rigidity * dist(e.source, e.target)$
3: $dy := (e.target.y - e.source.y) * rigidity * dist(e.source, e.target)$
4: $e.target \mathrel{+}= (-dx, -dy)$
5: $e.source \mathrel{+}= (dx, dy)$
6: **end for**

Umso kleiner der *rigidity*-Faktor ausgewählt wird, desto mehr ziehen sich miteinander verbundene Knoten an.

Die Abstoßungskraft zwischen den Knoten steigt mit der Höhe von *repDistance*. Wird sie angemessen hoch gewählt, ergeben sich weniger Überlappungen von Knoten. Alg. 22 zeigt diesen Sachverhalt.

Das abschließende Bewegen aller Knoten ist in Alg. 23 dargestellt. Eine wichtige Funktion besitzt hier der *damper*. In Abhängigkeit von der Geschwindigkeit der Knotenbewegungen zwischen einzelnen Iterationsschritten bewegt er die Knoten mehr oder weniger schnell. Die Variable *motionRatio* vergleicht die Knotenbewegung der letzten Iteration mit der aktuellen

Algorithmus 22 TouchLayout: Methode void avoidLabels

```
1: for all node m of nodes do
2:     for all nodes n of nodes do
3:         vx := m.x − n.x
4:         vy := m.y − n.y
5:         dx := 0
6:         dy := 0
7:         distance := distance from m to n
8:         if distance == 0 then
9:             dx := 0.01
10:            dy := 0.01
11:        else
12:            if distance < 5 then
13:                dx := vx/distance
14:                dy := vy/distance
15:            end if
16:            repX := (MAX(m.width, n.width) ∗ 5)^2
17:            repY := (MAX(m.height, m.height) ∗ 5)^2
18:            randomNumber := random(0,1)
19:            if randomNumber > 0.03 then
20:                m += (dx ∗ repX, dy ∗ repY)
21:                n += (−dx ∗ repY, −dy ∗ repX)
22:            else
23:                m += (dx ∗ repX ∗ 3, dy ∗ repY ∗ 3)
24:                n += (−dx ∗ repX ∗ 3, −dy ∗ repY ∗ 3)
25:            end if
26:        end if
27:    end for
28: end for
```

anhand des Knotens, der am weitesten vom Koordinatenursprung entfernt ist und seiner Entfernung zu ihm. Der *damper* wird nur eingesetzt, wenn sich die Knoten sehr stark bewegen, er schwächt diesen Zustand ab.

Algorithmus 23 TouchLayout: Methode void moveNodes

Require: $damper$, $maxMotion$, $lastMaxMotion$
1: $lastMaxMotion := maxMotion$
2: $maxMotionA := 0$
3: **for all** node m of $nodes$ **do**
4: $position := ((m.x + m.width/2), (m.y + m.height/2))$
5: $dx := m.x * damper/2$
6: $dy := m.y * damper/2$
7: $m.x := dx$
8: $m.y := dy$
9: $distMoved := SQRT(dx^2 + dy^2)$
10: **if** dx ! = 0 OR dy ! = 0 **then**
11: $position.x \mathrel{+}= MAX(-0.05, MIN(0.05, dx))$
12: $position.y \mathrel{+}= MAX(-0.05, MIN(0.05, dy))$
13: $tmpX := MAX(0, (position.x - m.width/2))$
14: $tmpY := MAX(0, (position.y - m.height/2))$
15: $m.x := tmpX$
16: $m.y := tmpY$
17: **end if**
18: $maxMotionA := MAX(distMoved, maxMotionA)$
19: **end for**
20: $maxMotion := maxMotionA$
21: **if** $maxMotion > 0$ **then**
22: $motionRatio := lastMaxMotion/maxMotion - 1$
23: **else**
24: $motionRatio := 0$
25: **end if**
26: **if** $motionRatio <= 0.001$ **then**
27: **if** ($maxMotion < 0.2$ OR ($maxMotion > 1$ AND $damper < 0.9$)) AND $damper > 0.01$ **then**
28: $damper := damper - 0.01$
29: **else**
30: **if** $maxMotion < 0.4$ AND $damper > 0.003$ **then**
31: $damper \mathrel{-}= 0.003$
32: **else**
33: **if** $damper > 0.0001$ **then**
34: $damper := damper - 0.0001$
35: **end if**
36: **end if**
37: **end if**
38: **end if**
39: **if** $maxMotion < 0.001$ **then**
40: $damper := 0$
41: **end if**

2.4.5 Simple Edge-Based, EdgeBased2

Da die Edge-Based-Layoutverfahren zum Berechnen der neuen Kantenpositionen mit der Tiefensuche die Knoten des Graphen durchlaufen, kännte man sie vielleicht auch den hierarchischen Layoutalgorithmen zuordnen. Es gibt eine Oberklasse namens *EdgeBasedLayout.java*. Sie kümmert sich um die Tiefensuche und die Zuweisung der neuen Positionskoordinaten. Hierfür benutzt sie auch die Hilfsfunktionen von *GraphUtilities*. Die Tiefensuche sieht aus wie in Alg. 25 beschrieben. Die Unterscheidung der Kantentypen erfolgt mit Hilfe von Kantenbits. Sie sind binäre Werte, die als Integer-Variablen gespeichert werden, und können mit geeigneten Masken identifiziert werden, was die kantenbasierten Layoutalgorithmen aber nicht machen müssen. Alle Kantentypen sind für sie gleichwertig. Zusammen mit

den ihnen zugewiesenen Kantenlängen werden diese Kantentypen in einer Liste abgelegt, deren Struktur in Alg. 24 dargestellt ist. Die Klasse *EdgeBasedLayout* stellt dafür eine abstrakte Methode namens *getEdgeTypeProperties()* vom Typ *DrawingEdgeTypeProperties* zur Verfügung. Die Implementierung übernehmen die vom *EdgeBased-Layout* abgeleiteten Klassen. Mit der Liste vom Typ *DrawingEdgeTypeProperties*, die diese Methode zurückliefert, können dann die neuen Koordinaten ausgerechnet werden.

Algorithmus 24 EdgeBased: Klasse DrawingEdgeTypeProperties

```
1: Hashtable edgeTypes
2: void addEdgeType(int edgeBits, double direction, double length)
3: double getDirection(int edgeBits)
4: double getLength(int edgeBits)
5: boolean containsEdgeBits(int edgeBits)
```

Weiterhin anzumerken ist, dass es in der Java-Implementierung dieses Algorithmus nötig ist, die Grad-Werte in rad umzurechnen, bevor sie der Sinus- bzw. Cosinus-Funktion übergeben werden können. 180° entsprechen dabei dem Wert π. Alg. 25 zeigt das *EdgeBasedLayout*.

Algorithmus 25 EdgeBasedLayout

```
1: detp := getEdgeTypeProperties(nodes)
2: setAllEdgesAccessed(nodes, false)
3: for all root r do
4:     setNewCoordinatesForNeighbour(r)
5: end for
6: if mirrorLayout then
7:     for node n of nodes do
8:         tempX := n.x
9:         n.x := −n.y
10:        n.y := −tempX
11:    end for
12: end if
```

Mit der Funktion *setAllEdgesAccessed* werden alle Kanten des Graphen als unbesucht markiert.

Alle kantenbasierten Layoutverfahren können in GroIMP durch Auswahl der Variable *mirrorLayout* gespiegelt werden. Waagerechte Kanten werden dann senkrecht gezeichnet, senkrechte waagerecht, usw.

Das eigentliche Berechnen der neuen Positionen der Zielknoten von allen Kanten und damit die Ausrichtung dieser zum Richtungsvektor passiert in der Funktion *setNewCoordinatesForNeighbour*, wie in Alg. 26 dargestellt. Die Variable *detp* ist vom Typ *DrawingEdgeTypeProperties*. Sie enthält alle Kantentypen und für deren Kanten die künftige Kantenlänge.

Nach Berechnung der neuen Koordinaten wird, sofern es sich bei dem aktuell betrachteten Knoten nicht um ein Blatt handelt, diese Methode für all seine verbundenen Knoten, die mit einer eingehenden Kante von dem jetzigen Knoten aus besucht werden kännen, aufgerufen.

Es werden aber nur unbesuchte Kanten vom Algorithmus berücksichtigt. Dies prüft die Methode *isAccessed*. Liefert sie ein *true* zurück, dann wurde diese Kante bereits durchlaufen und die Funktion bricht ab. Bei einem *false* wird die neue Position des Zielknotens berechnet.

Liegen mehrere Kanten des gleichen Kantentyps an einem Knoten an, wird die zweite Kante in entgegengesetzter Richtung zur ersten gezeichnet, alle weiteren werden um ein Vielfaches von einer frei wählbaren Grad-Abweichung von der zuvor gezeichneten Kante des gleichen Typs an diesem Knoten gezeichnet. 1° in rad umgerechnet entsprechen $PI/180$.

Ebenso wird auch verfahren, wenn die neu zu zeichnende Knotenposition bereits von einem anderen Knoten, der in einem früheren Durchgang neue Koordinaten erhalten hatte, belegt ist. Es wird verglichen, ob andere bisher berechnete Knotenpositionen in dem Intervall von aktuell berechneter Knotenposition plus/minus der Breite/Länge des Knoten liegen. Wenn ja, wird diese Knotenposition mit einer kleinen Abweichung neu berechnet. Da auch auf diesen neu berechneten Koordinatenwerten bereits gezeichnete Knoten liegen könnte, muss der Schritt so oft ausgeführt werden, bis ein freier Platz gefunden wurde.

DIV steht für die ganzzahlige Division.

Algorithmus 26 EdgeBasedLayout: Methode void setNewCoordinatesForNeighbour

```
Require: edgeLength, degreeDeviationAgainstOverlapping, Node n
 1: edgeTypeCounter := ()
 2: for all edge e of n do
 3:     if e.isAccessed == true then
 4:         continue //Abbruch
 5:     end if
 6:     e.isAccessed := true
 7:     if edgeTypeCounter enthält nicht e.edgeBits then
 8:         edgeTypeCounter.put(e.edgeBits, 0)
 9:     else
10:         edgeTypeCounter.put(e.edgeBits, +1)
11:     end if
12:     direction := detp.getDirection(e.edgeBits)
13:     if edgeTypeCounter.get(e.edgeBits) MOD 2 == 0 then
14:         direction += edgeTypeCounter.get(bitsOfAnEdge) DIV 2 * (π/180*degreeDeviationAgainstOverlapping)
15:     else
16:         direction += edgeTypeCounter.get(bitsOfAnEdge) DIV 2 * (π/180*degreeDeviationAgainstOverlapping) + PI
17:     end if
18:     e.target.x := e.source.x + (detp.getLength(e.edgeBits) * COS(direction))
19:     e.target.y := e.source.y + (detp.getLength(e.edgeBits) * SIN(direction))
20:     while e.target is hiding another node do
21:         direction := direction + PI / 180 * degreeDeviationAgainstOverlapping
22:         e.target.x := e.source.x + (detp.getLength(e.edgeBits) * COS(direction))
23:         e.target.y := e.source.y + (detp.getLength(e.edgeBits) * SIN(direction))
24:     end while
25:     setNewCoordinatesForNeighbour(e.target)
26: end for
```

Simple Edge-Based

Vom Benutzer einstellbar ist die einheitliche Länge aller Kanten des Graphen, sowie der Abweichungsgrad für den Fall, dass mehrere Kanten gleichen Typs am selben Knoten anliegen.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
edgeLength	$\mathbb{R}$	3.0	Kantenlänge
degreeDeviation	$\mathbb{R}$	10.0	Gradabweichung der Kanten gleichen Typs an einem Knoten
mirrorLayout	*boolean*	false	Wenn true, dann wird das Layout gespiegelt
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.14: Eingabeparameter des Simple Edge-Based-Layouts

Zuerst wird die Anzahl der Kantentypen im Graphen ermittelt. Mit dieser kann dann die Richtung, in der die erste Kante eines bestimmten Typs gezeichnet werden soll, berechnet werden. Diese Werte werden in der Hashtable *detp* gespeichert und an den eigentlichen Layoutalgorithmus in der Oberklasse *EdgeBasedLayout* weitergeleitet.

Die Variable *edgeLength* enthält den Wert der einheitlichen Größe der Kantenlängen im Layout.

Algorithmus 27 Simple EdgeBasedLayout: Methode DrawingEdgeTypeProperties getEdgeTypeProperties

```
1: detp := new DrawingEdgeTypeProperties()
2: for node n of nodes do
3:     for edge e of n do
4:         if e.isAccessed then
5:             continue //Weiter zum nächsten Schleifendurchlauf
6:         end if
7:         e.isAccessed := true
8:         if detp enthält nicht e.edgeBits then
9:             direction := 0
10:            radDistance := PI / total number of different edgeBits
11:            direction := (number of this type of edgeBits) * radDistance
12:            detp.addEdgeType(e.edgeBits, direction, edgeLength)
13:        end if
14:    end for
15: end for
16: return detp
```

Edge-BasedLayout2

Das erweiterte kantenbasierte Layoutverfahren definiert zwei verschiedene Kantenlängen, eine für die zwei häufigsten Kantentypen, die andere für die die Kanten der übrigen Kantentypen.

Name	Datentyp	Vorgabewert	Beschreibung
		Benutzerspezifisch	
shortEdgeLength	$\mathbb{R}$	3.0	Kantenlänge aller Kantentypen außer der zwei häufigsten
longEdgeLength	$\mathbb{R}$	5.0	Kantenlänge der zwei häufigsten Kantentypen
degreeDeviation	$\mathbb{R}$	10.0	Gradabweichung der Kanten gleichen Typs an einem Knoten
mirrorLayout	*boolean*	false	Wenn true, dann wird das Layout gespiegelt
		Weitere	
nodes	Liste		Alle Knoten des Graphen, auf die der Algorithmus angewendet werden soll

Tabelle 2.15: Eingabeparameter des Edge-Based-Layout2

Der Algorithmus, der die Zeichenrichtungen der einzelnen Kantentypen berechnet, ist in Alg. 28 dargestellt. Die beiden am häufigsten vorkommenden Kantentypen müssen ermittelt werden. Sie bekommen die festen Richtungen 0° bzw. 90° zugewiesen. Die Kanten aller anderen Typen erhalten dann, ähnlich wie im Simple EdgeBased-Layout, Richtungen zwischen 0° und 180°, mit Ausnahme der schon vergebenen 0° und 90°. Dabei wird so verfahren, dass der erste eine Richtung zwischen 0° und 90° erhält, der zweite zwischen 90° und 180°, der dritte wieder zwischen 0° und 90°, usw.

Algorithmus 28 EdgeBasedLayout2: Methode DrawingEdgeTypeProperties getEdgeTypeProperties

```
1: edgeTypeCounter = ()
2: for node n of nodes do
3:     for all edge e of n do
4:         if e.isAccessed then
5:             continue //Weiter mit nächstem Schleifendurchlauf
6:         end if
7:         e.isAccessed := true
8:         if edgeTypeCounter contains not e.edgeBits then
9:             edgeTypeCounter.put(e.edgeBits, 1)
10:        else
11:            edgeTypeCounter.put(e.edgeBits, +1)
12:        end if
13:    end for
14: end for
15: mostUsedEdgeBit := numberMost := 0
16: secondUsedEdgeBit := numberSecond := 0
17: for all edgeBits in edgeTypeCounter do
18:    number := edgeTypeCounter.get(e.edgeBits);
19:    if number > numberMost then
20:        numberSecond := numberMost
21:        secondUsedEdgeBit := mostUsedEdgeBit
22:        numberMost := number
23:        mostUsedEdgeBit := edgeBits
24:    else
25:        if number > numberSecond then
26:            numberSecond := number
27:            secondUsedEdgeBit := edgeBits
28:        end if
29:    end if
30: end for
31: detp := new DrawingEdgeTypeProperties()
32: detp.addEdgeType(mostUsedEdgeBit, 0, longEdgeLength)
33: detp.addEdgeType(secondUsedEdgeBit, (PI/2), longEdgeLength)
34: radDistance := (PI / (total number of edgeBits + 1))
35: counter := 1
36: for all edgeBits do
37:    if detp enthält nicht edgeBits then
38:        direction := ((counter+1)/2) * radDistance
39:        if counter MOD 2 == 0 then
40:            direction += PI / 2
41:        end if
42:        detp.addEdgeType(edgeBits, direction, shortEdgeLength)
43:        counter := counter + 1
44:    end if
45: end for
46: return detp
```

2.5 Klassenstruktur

Zum Schluss wird in Abb. 2.3 die Klassenstruktur der Layoutalgorithmen übersichtlich dargestellt. Dabei wurde sich auf die wichtigsten Variablen und Methoden konzentriert, damit das Klassendiagramm einigermaßen übersichtlich bleibt und die Abhängigkeiten voneinander erkennbar sind. Öffentlich sichtbare Variablen sind mit # gekennzeichnet, Variablen, die nur innerhalb der Klasse, in der sie deklariert wurden, sichtbar sind, wurde ein - vorangestellt und die, die lediglich innerhalb ihres Paket verwendet werden können, haben ein +.

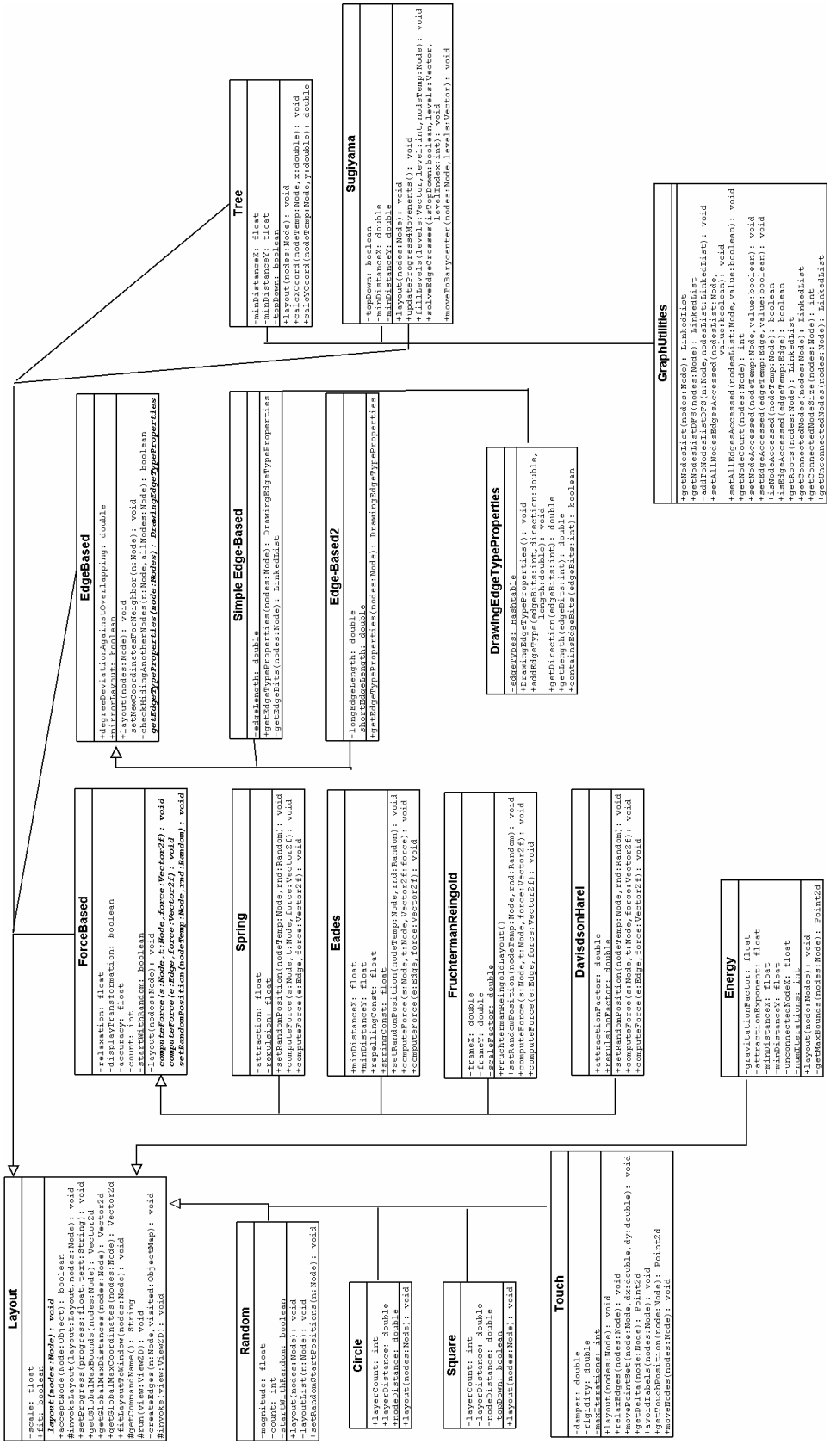

Abbildung 2.3: Übersicht der implementierten Layoutalgorithmen

2.6 Interpolation der Layoutalgorithmen

Manchmal kann es gewünscht sein, dass die Layoutalgorithmen nicht starr auf ein aktuelles Graphenbild angewendet werden, sondern dass die Übergänge vom alten zum neuen Bild fließend verlaufen. Der Benutzer kann mit dem Auge bestimmte Knoten verfolgen, wie sie ihren Platz im neuen Layout einnehmen.

Dies kann man mit einer linearen Interpolation von Layoutalgorithmen erreichen. Eine Interpolation ist eine Berechnung von neuen Werten, die zwischen zwei vorhandenen Datenwerten liegen. Eine lineare Interpolation kann durch folgende Funktion beschrieben werden [WI07b]

Gegeben: f_0, f_1 .. Datenpunkte, durch eine Strecke miteinander verbunden

$$f(x) := f_0 + \frac{(f_1 - f_0)}{x_1 - x_0} * (x - x_0) \tag{2.32}$$

In GroIMP wird immer zwischen dem aktuell gezeichnetem Graphbild und dem Ergebnis des ausgewählten Layoutverfahren interpoliert. Die Interpolation dadurch erreicht, dass der Layoutalgorithmus auf den Graphen angewendet wird, dieser aber nicht sofort neu gezeichnet wird. Stattdessen ruft sich der Algorithmus in *Layout.java* je nach Anzahl der Interpolationsschritte mehrfach auf und und berechnet für den aktuellen Zwischenschritt die Koordinaten aller Knoten. So bewegen sich die Knoten in regelmäßigen Abständen von ihrem Ursprungspunkt zu ihrem Ziel hin.

Über die Anzahl der Zwischenschritte kann der Benutzer in GroIMP frei entscheiden. Jeder Layoutalgorithmus besitzt einen zusätzlichen vom Benutzer einstellbaren Parameter namens *transformationSteps*. Eine 1 bedeutet, dass der Graph direkt und ohne Zwischenschritte gezeichnet werden soll. Die Zeit, die vergehen soll, bis der nächste Schritt gezeichnet wird, wurde auf 100 ms festgelegt, damit eine möglichst fließende Bewegung der Knoten simuliert wird, der Rechner aber nicht überlastet wird.

2.7 Demonstration

Für die Demonstration wurde der GroIMP-Beispielgraph *Busch* nach der 1. Regelanwendung verwendet. Er besitzt 26 Knoten und 25 Kanten. Bei Aufruf des Graphen in GroIMP über das Menü *Panels, 2D, Graph* öffnet sich das Fenster zur Darstellung der zweidimensionalen Graphenstruktur. Beim ersten Aufruf wird das Aussehen der Graphstruktur durch das Square-Layout neu berechnet, wie es in Abb. 2.5 gezeigt wird.

Die nun folgenden Bilder zeigen das Graphlayout nach Anwendung der Algorithmen. Es wurden maximal 100 Iterationen zugelassen. Mit Ausnahme des Touch-Layouts liefert jedes Layoutverfahren schon bei dieser geringen Anzahl von Iterationen eine für sein Layout charakteristische Zeichnung und die Knoten werden mit geeignetem Abstand voneinander positioniert. Das Tree- und das Sugiyamalayout liefern eine klar gegliederte hierarchische Zeichnung. Die kräftebasierten Algorithmen dagegen verdeutlichen jeder auf seine Art die Anziehungskräfte von miteinander verbundenen Knoten und Abstoßungskräfte zwischen allen Knotenpaaren. Ebenfalls gut erkennbar ist die Clusterung von Knotenmengen nach dem Anwenden des Energy-Layouts auf den Graphen.

Das Touch-Layout dagegen weist nach 100 Iterationsschritten noch kein befriedigendes Ergebnis auf. Nach weiteren 900 Iterationen gehen jedoch auch bei diesem Layout die Knoten allmählich auseinander. Allerdings bildet das Layout die Knoten auf keine symmetrische Struktur ab, es wirkt ein wenig chaotisch.

Bei diesem gewählten Beispiel bringen Simple EdgeBased-Layout und EdgeBased2-Layout annähernd die gleichen Ergebnisse. Um die Unterschiede der Auswirkungen der kantenbasierten Layouts zu zeigen, wurde das Busch-Beispiel für die letzten beiden Bilder dieses Kapitels ein wenig abgeändert: Es wurde ein dritter Kantentyp einführt, den einige der Kanten im Graphen nun tragen (erkennbar an der Lila-Färbung dieser Kanten).

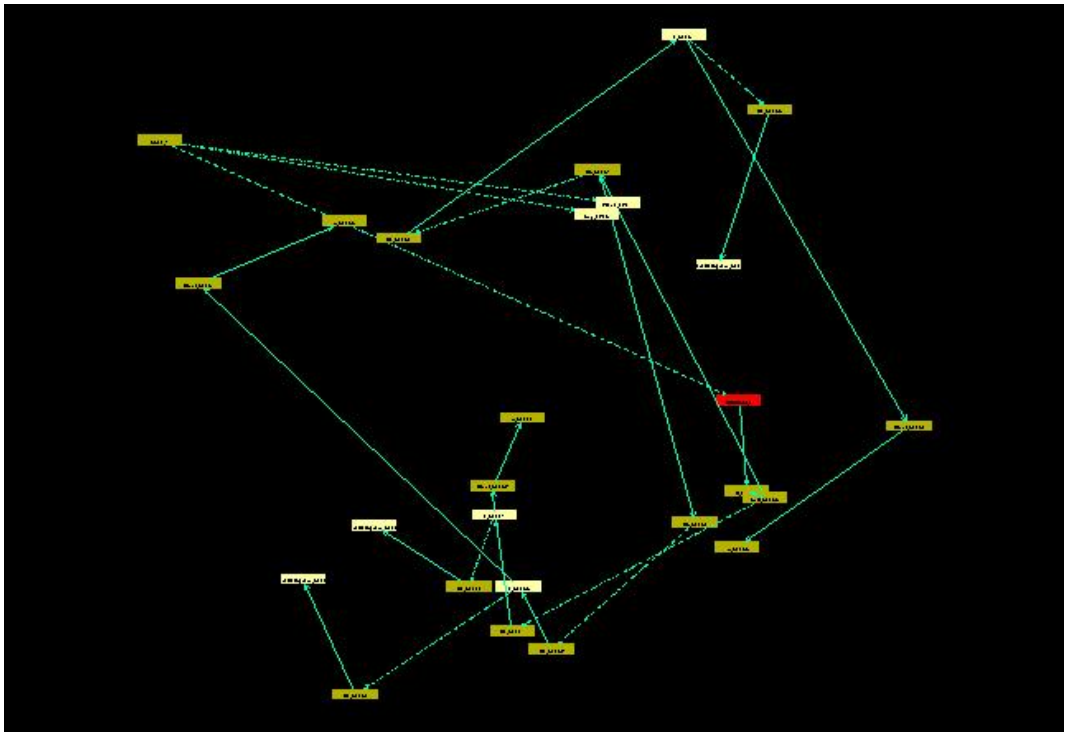

Abbildung 2.4: Random-Layout

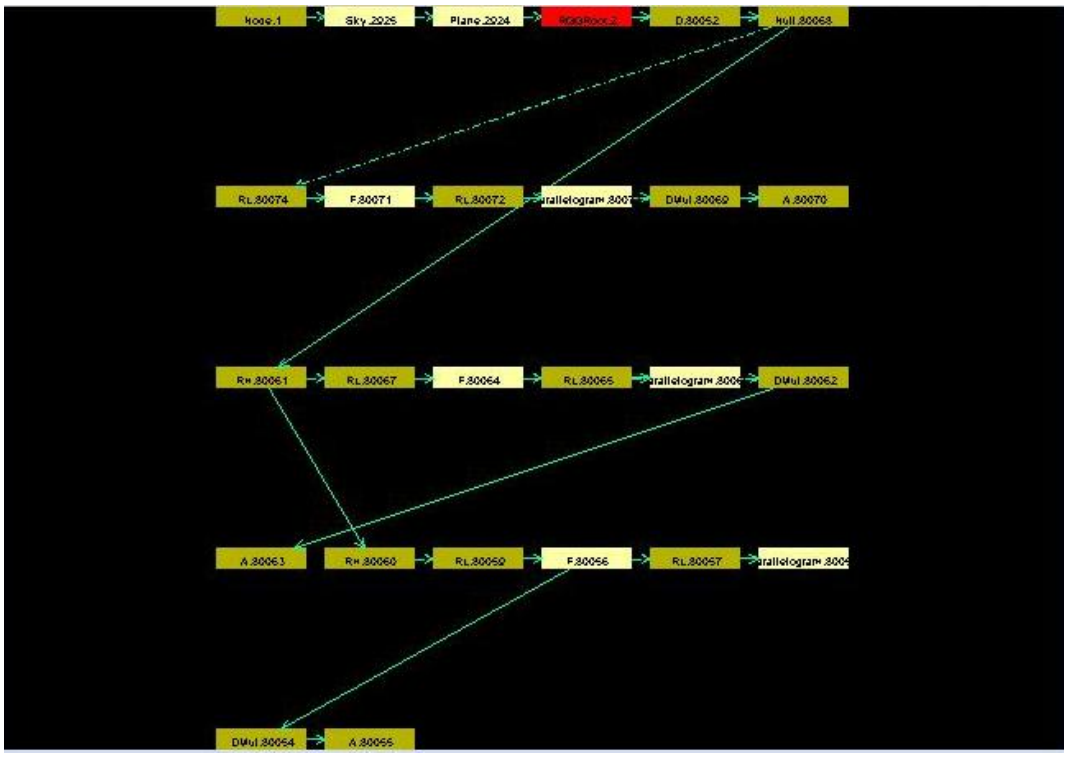

Abbildung 2.5: Square-Layout

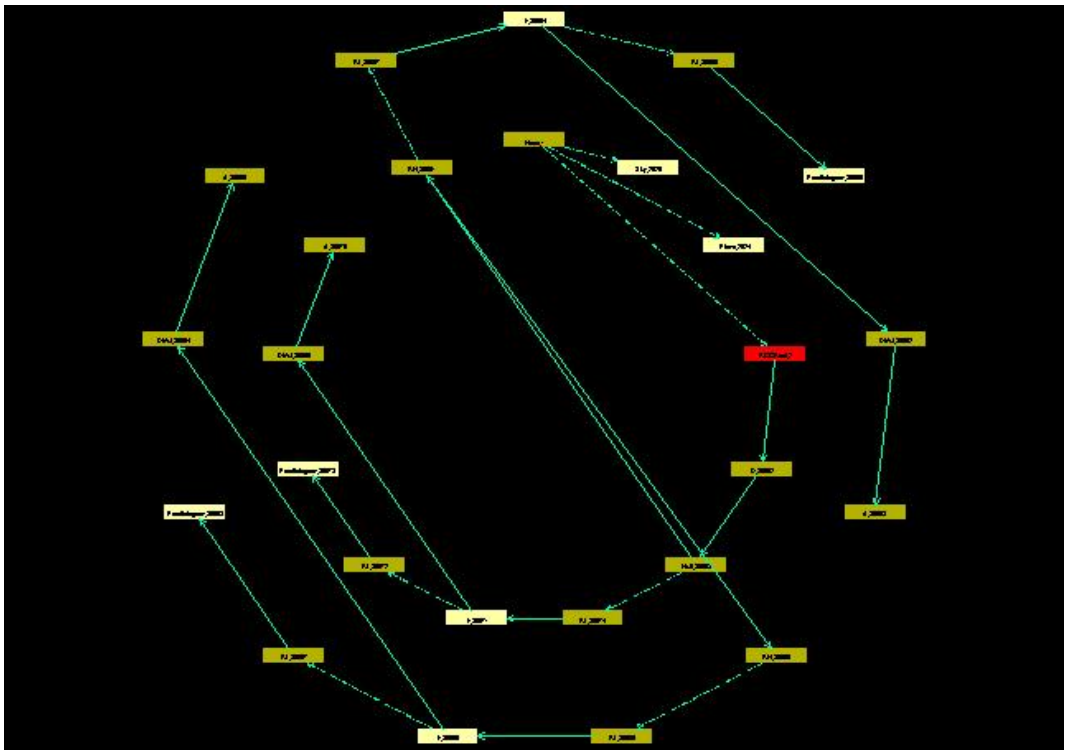

Abbildung 2.6: Circle-Layout

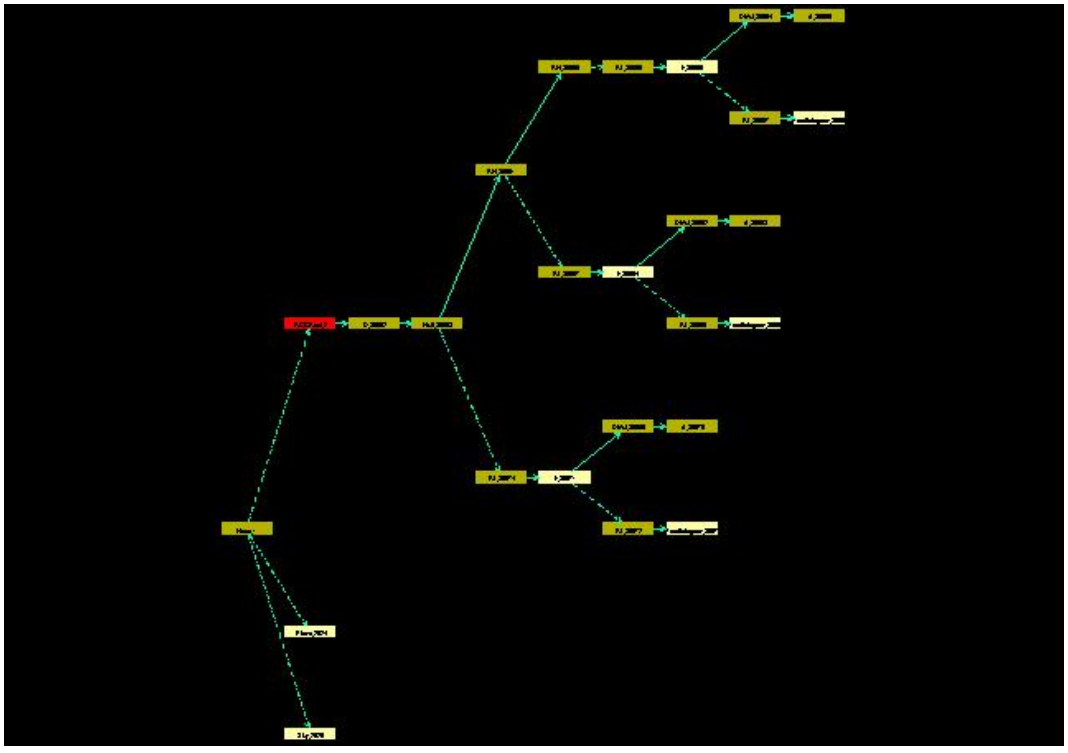

Abbildung 2.7: Tree-Layout

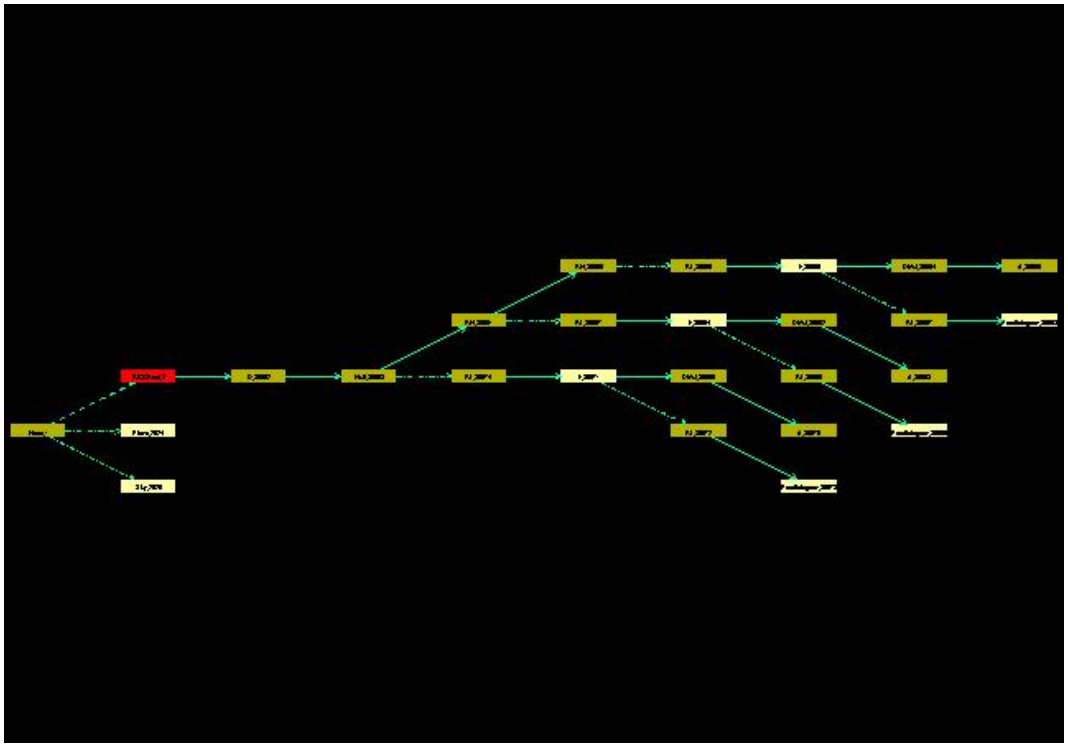

Abbildung 2.8: Sugiyama-Layout

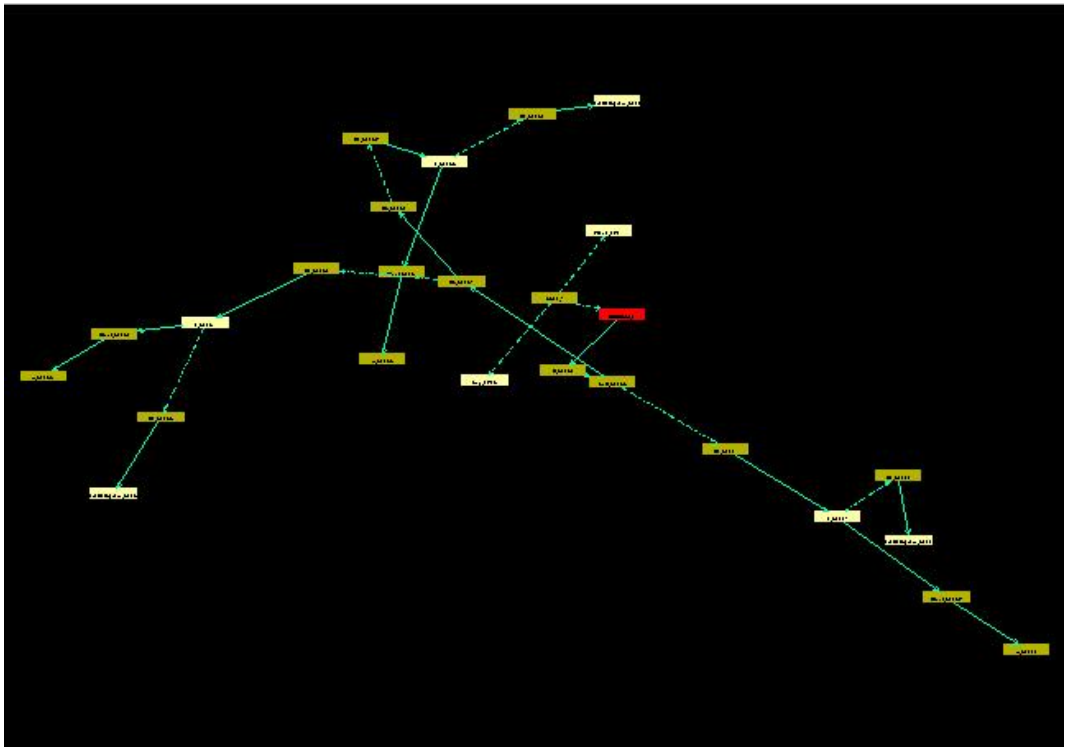

Abbildung 2.9: Spring-Layout

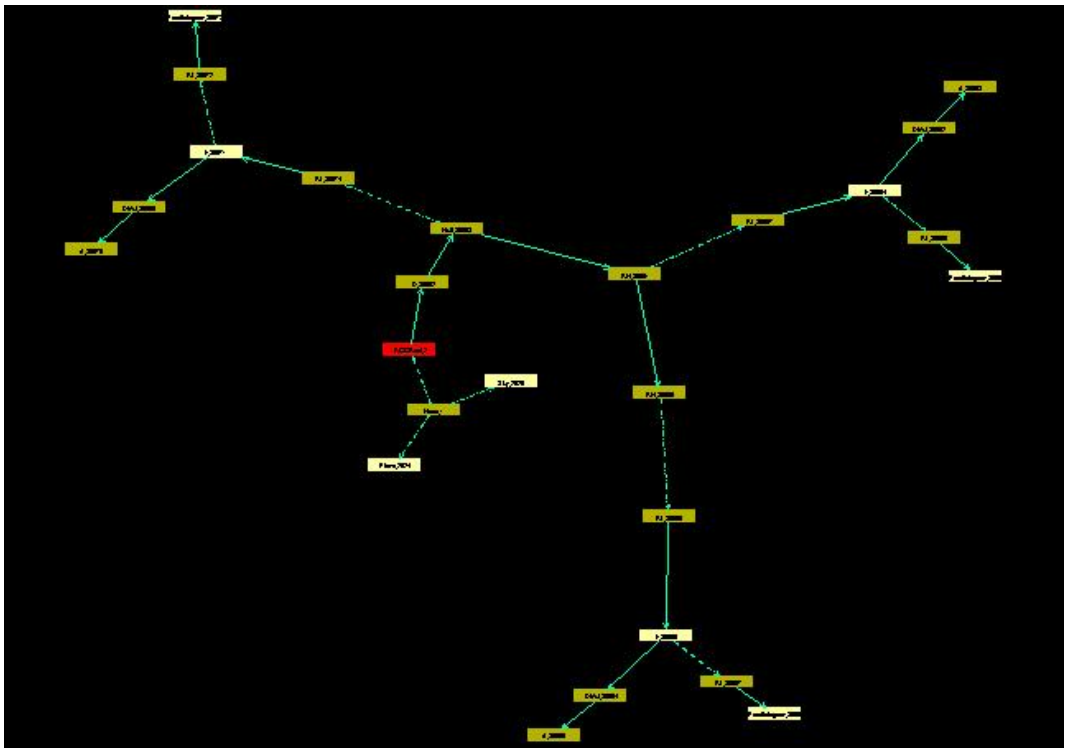

Abbildung 2.10: Eades-Layout

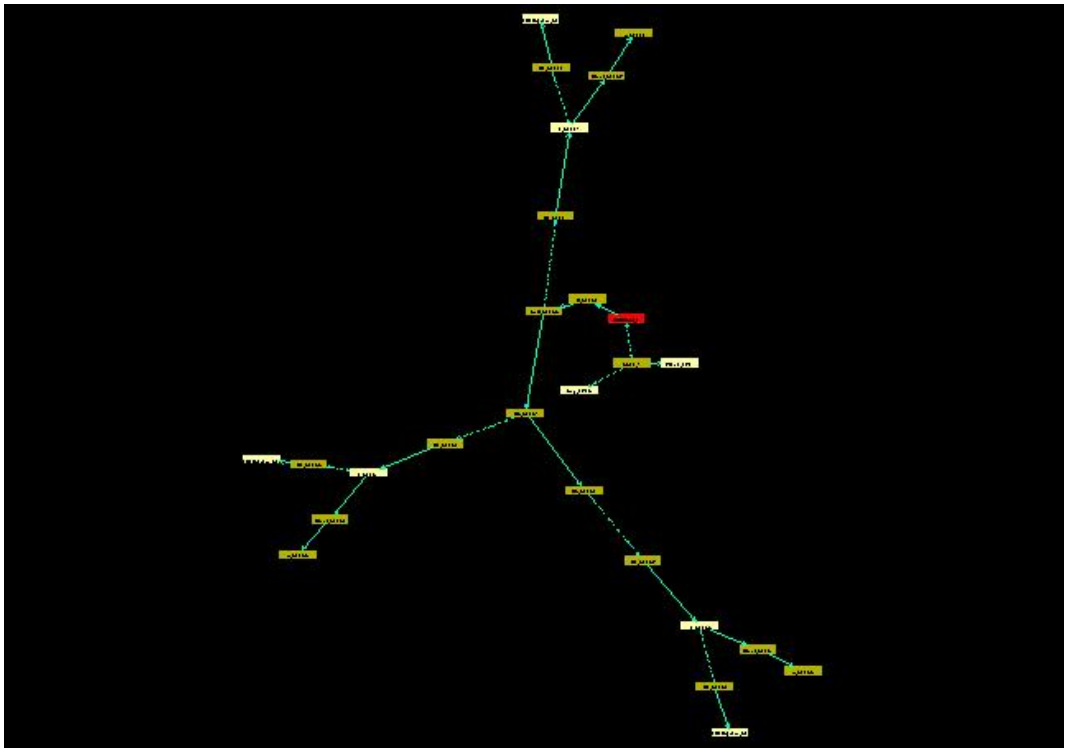

Abbildung 2.11: Fruchterman-Reingold-Layout

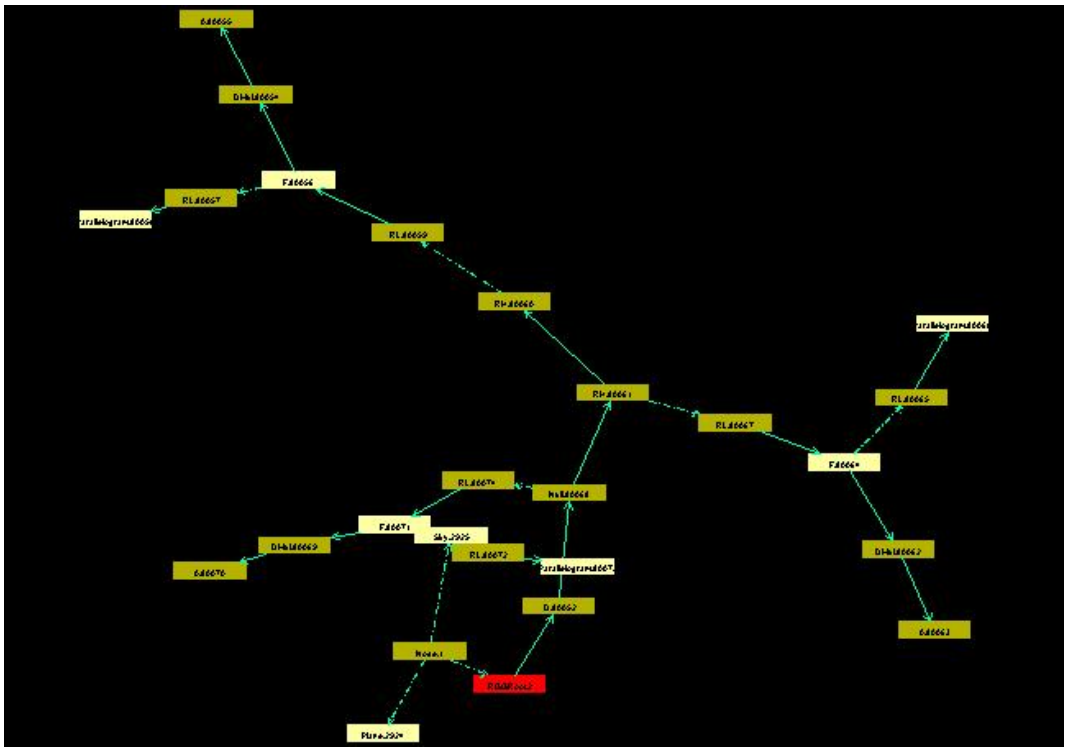

Abbildung 2.12: Davidson-Harel-Layout

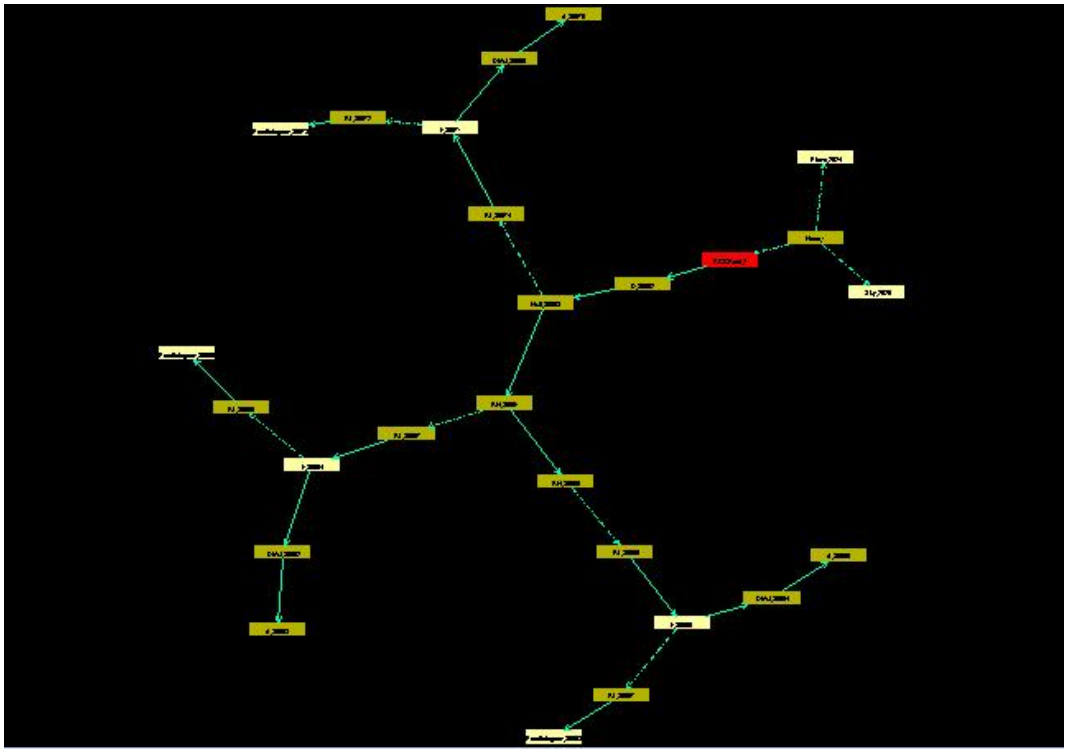

Abbildung 2.13: Energy-Layout

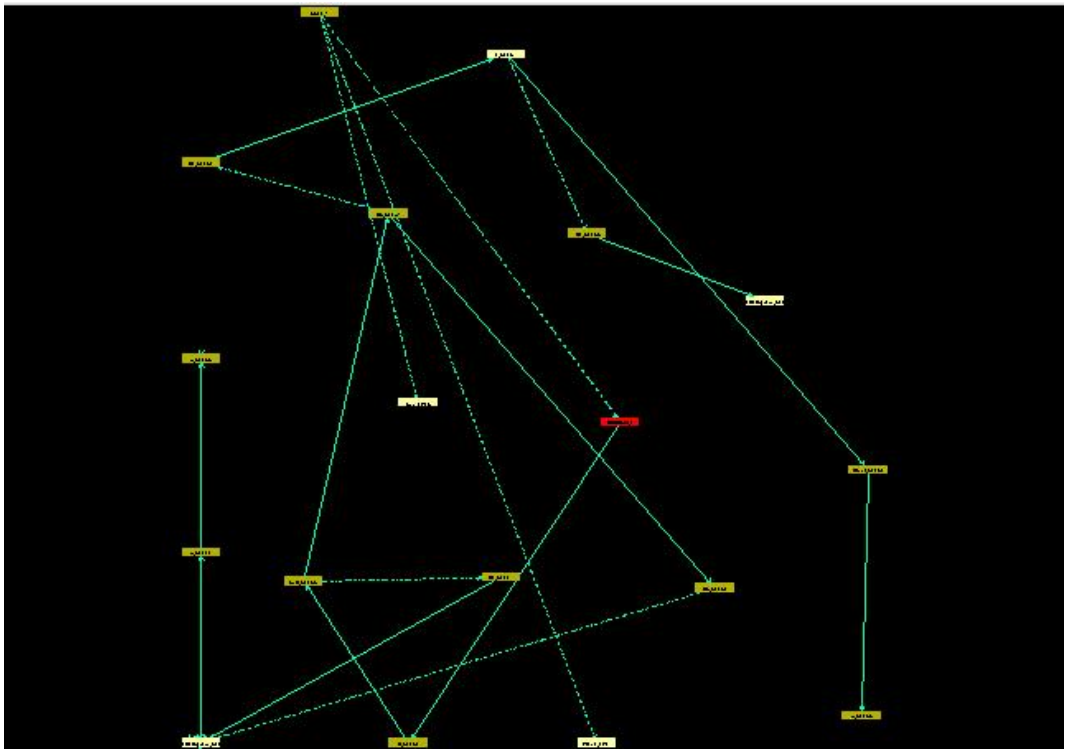

Abbildung 2.14: Touch-Layout nach 1000 Iterationen

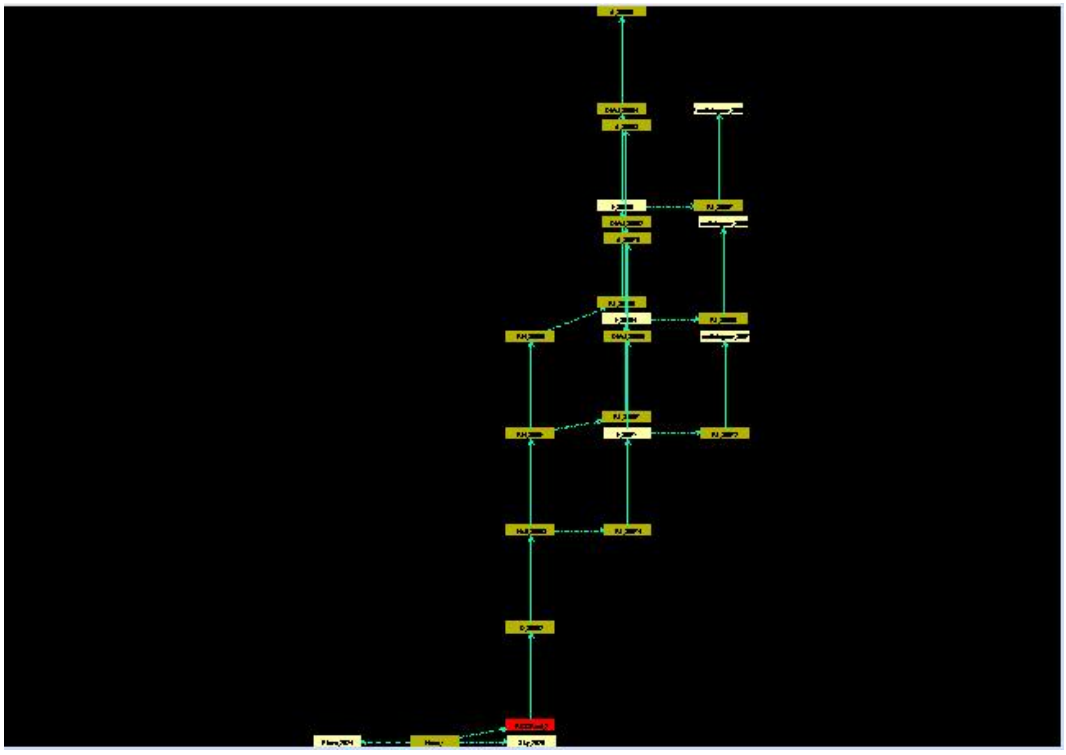

Abbildung 2.15: Simple Edge-Based-Layout

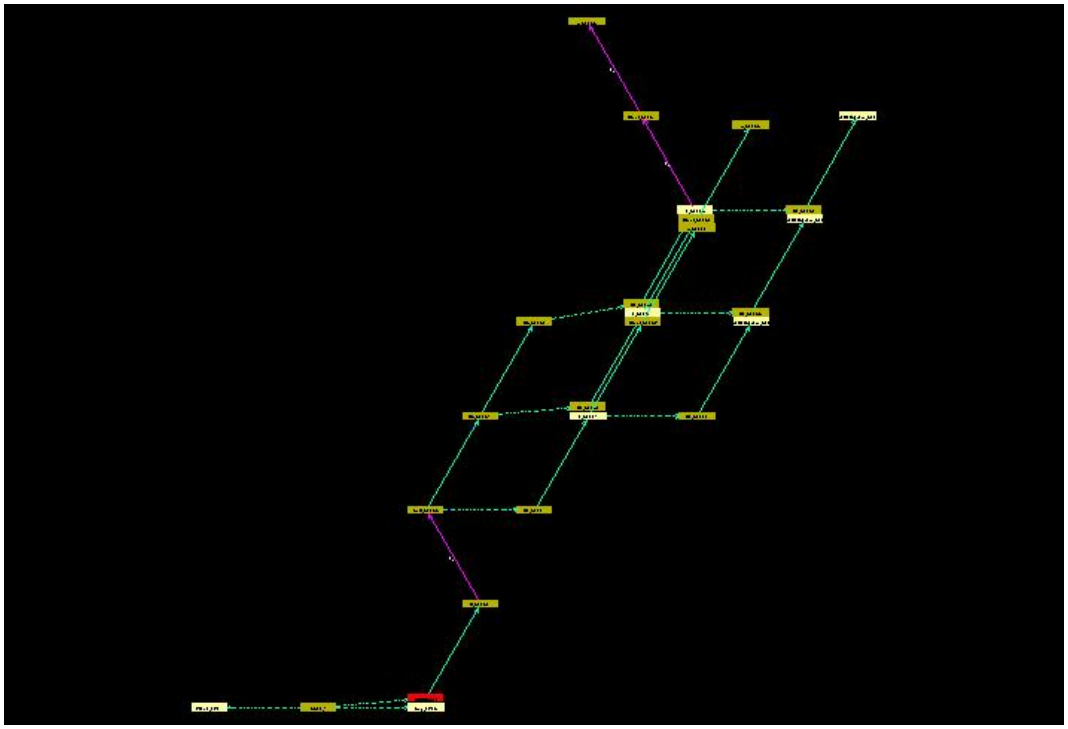

Abbildung 2.16: Simple Edge-Based-Layout mit drei verschiedenen Kantentypen

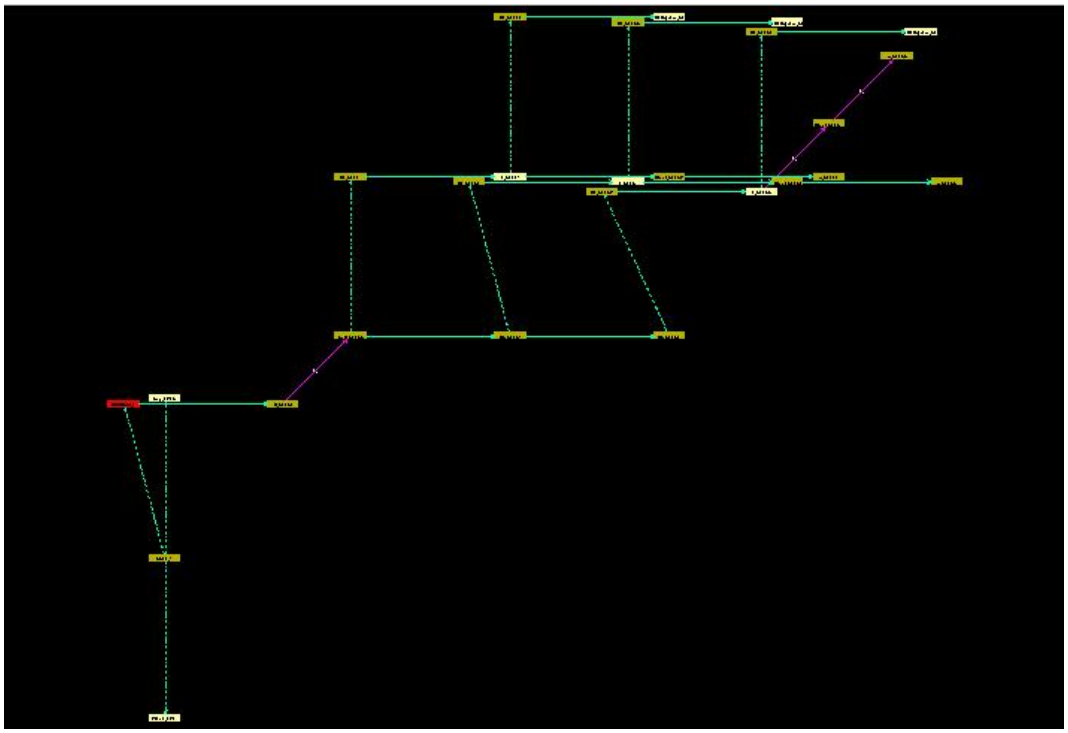

Abbildung 2.17: Edge-Based-Layout2 mit drei verschiedenen Kantentypen

2.8 Vergleich und Diskussion der Layoutalgorithmen

Anhand einiger ausgewählter Kriterien, die in der Einleitung zu diesem Kapitel erklärt wurden, sollen die im Vorfeld besprochenen Layoutalgorithmen nun verglichen werden. Es wird sich zeigen, dass es kein perfektes, universal einsetzbares Layoutverfahren gibt. Jedes hat seine spezifischen Stärken, aber auch Schwächen.

Die größte Stärke der einfachen Layoutalgorithmen ist die geometrische Anordnung der Knoten ohne viel Rechenaufwand. So werden die Knoten beim Circle-Layout auf einem oder mehreren Kreisen positioniert, das Square-Layout ordnet sie in einem rechteckigen Gitter an. Der Benutzer hat so eine übersichtliche Darstellung aller vorhandenen Knoten eines Graphen. Es gibt keine Knotenüberlappungen, und das Zeichenfenster ist gleichmäßig ausgefüllt. Allerdings lassen sich an diesen Layoutergebnissen die Beziehungen der Knoten untereinander nur schlecht ablesen. Die Kanten werden in keinster Weise in der Berechnung des Layouts berücksichtigt. Daher sind auch Kantenüberkreuzungen nicht steuerbar und Kantenlängen von kurz bis lang möglich.

Die kräftebasierten Layoutalgorithmen, zu denen das Spring-, Eades-, FruchtermanReingold- und DavidsonHarel-Layout gehören, ordnen die Knoten eines Graphen in Abhängigkeit von deren Beziehungen untereinander an. Dies ergibt, je nach Einstellung der Parameter und Komplexität des Graphen, erst nach vielen Iterationen ein übersichtliches Bild. Eine gleichmäßige Verteilung der Knoten auf der gesamten Zeichenfläche ist in vielen Fällen nicht gegeben. Es entstehen Häufungen von Knoten, die untereinander viele Kantenbeziehungen haben und sich gegenseitig anziehen, und an anderen Stellen können einzelne Knoten einsam stehen, weil sie durch ihre wenigen Beziehungen von den übrigen Knoten abgestoßen werden. Aber genau dieser Sachverhalt ist bei manchen Graphendarstellungen geradezu erwünscht.

Ähnlich verhält sich auch das Energy-Layout. Zusätzlich minimiert es noch die Kantenüberkreuzungen. Eine geringe Einstellung der Clusterbildung verteilt die Knoten gleichmäßig auf der Zeichenfläche. Bei einer großen Clusterbildung dagegen werden die Kräftebeziehungen in den lokalen Gebieten im Graphen und die aus ihnen heraus zu anderen deutlicher hervorgehoben.

Hierarchiebasierte Layoutverfahren (sowie auch die kantenbasierten Layouts) berücksichtigen als einzige die Richtungen der Kanten und unterscheiden somit Quell- und Zielknoten dieser. Durch das Zeichnen der Knoten in eine Richtung, beginnend von der Wurzel bis zu den Blättern, und durch die einheitlichen Kantenlängen werden Kantenüberkreuzungen weitestgehend vermieden.

Während die einfachen Layoutalgorithmen den Schwerpunkt auf die Darstellung der Kno-

ten an sich legen und die kräftebasierten die Beziehungen der Knoten in den Vordergrund stellen, spielen die Knoten bei den kantenbasierten Layoutverfahren keine Rolle. Den verschiedenen Kantentypen werden Richtungen zugeordnet, in denen die Kanten dieser Typen dann gezeichnet werden. Aus diesem Grund werden die möglichen Layouteigenschaften, die kantenbasiert sind, alle erfüllt. So gibt es eine einheitliche Kantenlänge (beim EdgeBased2-Layout zwei) und dadurch, dass bei der Durchführung des Algorithmus die Knoten in Tiefensuche durchlaufen werden, werden auch die Kantenrichtungen berücksichtigt. Diese Aspekte bringen bei den meisten Graphen eine erhebliche Minimierung der Kantenüberkreuzungen, besonders dann, wenn der Graph viele Kantentypen, aber eine relativ geringe Anzahl von Kanten hat. Bei dem gegenteiligen Fall, also einem Graphen mit vielen Kanten, aber wenig Kantentypen, ist die Überlappung von Knoten ein häufiges Phänomen. Daher zeichnet der Algorithmus auch die sich überlappenden Knoten in einem gewissen Abstand voneinander, was eine Abweichung der zugewiesenen Richtungen zu den Kantentypen mit sich bringt. Daher ergibt sich nicht immer ein symmetrisches und harmonisches Layoutbild.

Eine Sonderstellung nimmt das Random-Layout unter den Layoutverfahren ein. Da dieses Verfahren die Knotenpositionen zufällig auf der Zeichenfläche anordnet, ergibt sich kein symmetrisches Bild. Rein statistisch gesehen müssten die Knoten durch die Zufallskomponente gleichmäßig auf der Zeichenfläche verteilt werden. Dies kann aber genauso zufällig sein. So wie die Knoten unterliegen auch die Kanten keinen festgelegten Grundsätzen. Allerdings ist es möglich, diesen Layoutalgorithmus nach Ausführung eines anderen auf den Graphen anzuwenden, was dazu führt, dass das für das andere Layout so typische Resultat etwas verzerrt werden kann und auch Anwendung finden wird.

Ebenso nutzt auch das Touch-Layout eine Zufallsfunktion, um Knotenüberlappungen zu vermeiden, sie einigermaßen gleichmäßig auf die Zeichenfläche zu verteilen. Die Layouteigenschaften, welche die Kanten berücksichtigen, werden allerdings vernachlässigt.

Alle implementierten Layoutalgorithmen haben miteinander gemein, dass ihre berechneten Layoutbilder optimal an das Zeichenfenster angepasst werden können. Eine globale und für alle Verfahren verfügbare Fit-Funktion macht dies möglich, und zusammen mit dem Setzen des ersten Knoten der Liste auf eine bestimmte Stelle im Zeichenfenster erleichtert es dies dem Betrachter, einen Blick auf die gesamte Graphenstruktur zu werfen.

Den Vergleich der in GroIMP implementierten Layoutalgorithmen fasst Tab. 2.16 noch einmal zusammen (x = Merkmal vorhanden, - = Merkmal nicht vorhanden).

Aus dem Vorhandensein bzw. Fehlen einer bestimmten Eigenschaft lässt allerdings nicht gleichzeitig auf die Qualität eines Layoutverfahrens schließen. Es ist je nach Anwendungsfall zu entscheiden, welche Kriterien beim Zeichnen des Graphen wichtig sind und welche nicht

Eigenschaften / Layout	closeness	smallest separation	fixed edge length	symmetry	uniform distribution	adaption to the frame	crossing minimization	edge directions	Rechenzeit	Laufzeitmessung (ms)
Random	-	-	-	-	(x)	x	-	-	O(n)	867
Square	-	x	-	x	x	x	-	-	O(n)	906
Circle	-	x	-	x	x	x	-	-	O(n)	828
Tree	x	x	x	-	x	x	x	x	O(n)	829
Sugiyama	x	x	x	-	x	x	x	x	$O(n^2)$	1219
Spring	x	x	-	-	-	x	-	-	$O(n^2)$	8062
Eades	x	x	-	-	-	x	-	-	$O(n^2)$	8922
Fruchterman	x	x	-	-	-	x	-	-	$O(n^2)$	10985
Davidson	x	x	-	-	-	x	-	-	$O(n^2)$	8812
Energy	x	x	-	-	(x)	x	x	-	O(n log n)	14891
Touch	x	x	-	-	x	x	-	-	$O(n^2)$	43094
Simple Edge	x	x	x	-	-	x	(x)	x	$O(n^2)$	921
Edge2	x	x	x	-	-	x	(x)	x	$O(n^2)$	953

Tabelle 2.16: Vergleich der Layoutalgorithmen

oder gar vernachlässigt werden können. Um beispielsweise Hierarchien in einem Graphen betonen zu wollen, eignen sich hierarchiebasierte Layouts eher, wie z. B. das Tree-Layout, da die Kantenrichtungen von Bedeutung sind und man den Weg von der Wurzel des Graphen bis zu den Blättern verfolgen möchte. Sind dagegen die Beziehungen der Knoten untereinander wichtig, so wird man die kräftebasierten Layoutverfahren, wie z. B. das Eades-Layout, favorisieren, da es keine uniformen Kantenlängen gibt. Knoten, die miteinander verbunden sind, liegen dicht aneinander, unverbundene weit auseinander.

Aber auch die einfacheren Layoutalgorithmen, wie z. B. das Circle- oder Square-Layout können beim Graphzeichnen Verwendung finden. Bei ihnen ist ein schönes, gleichmäßig ausgefülltes Layoutbild das Resultat. Angewendet auf Graphen mit wenig Knoten kann man sie zum übersichtlichen Darstellen der vorhandenen Knoten benutzen.

Umso mehr Knoten und Kanten ein Graph hat, desto mehr spielt auch der Berechnungsaufwand und virtueller Speicherbedarf für das Layout eine Rolle. Dieser setzt sich aus der eigentlichen Berechnung durch den Algorithmus, sowie aus dem Ausführen von bestimmten Hilfsfunktionen vor oder nach diesem zusammen. Einfache Layoutverfahren, die eine O(n)-Komplexität besitzen, wie z. B. das Circle- oder Square-Layout, bringen auch bei großen Graphen schnelle Ergebnisse. Dagegen werden bei den kräftebasierten Layoutalgorithmen eine bestimmte Anzahl von Iterationen durchgeführt, in denen jeweils einmal die Anziehungskraft aller Kanten ($O(n^2)$), sowie die Abstoßungskraft aller Knotenpaare ($O(n^2)$) berechnet werden, was einen Gesamtaufwand von wiederum $O(n^2)$ ergibt. Die hierarchiebasierten Layoutverfahren haben beim Durchlaufen der Knoten nur eine Komplexität von O(n). Allerdings müssen vorher die Wurzeln des Graphen gesucht werden (O(n)), dann werden, ausgehend von diesen, die Kinderknoten und deren Kinder aufgerufen (O(n)). Beim

Sugiyama-Layout kommt nach dem Anordnen der Knoten auf den Schichten außerdem noch die Umordnung dieser hinzu, um die Kantenüberkreuzungen zu minimieren ($O(n^2)$). Desweiteren sind während des Ablaufs des Layouts diverse andere Funktionen auszuführen, die ebenfalls eine Komplexität von O(n) haben (z. B. das Markieren aller Knoten, und eventuell auch Kanten, als unbesucht). Ebenso verhält es sich auch mit den kantenbasierten Layoutalgorithmen. Zum Ausrichten der Kanten müssen nur die Knoten eines Graphen und deren zugehärige Kanten aufgerufen und neu gesetzt werden ($O(n^2)$). Die vorherige Berechnung der Richtungsvektoren umfasst aber ein Ermitteln aller vorhandenen Kantentypen (O(n)), mehrmaliges Setzen der Knotenmarkierungen auf unbesucht, usw. Das erweiterte kantenbasierte Layoutverfahren ist durch das Ermitteln der zwei häufigsten Kantentypen auch ein wenig komplexer als das Simple Edge-Based-Layout, weshalb es Sinn machte, beide Layoutideen voneinander zu trennen und nicht das Simple Edge-Based-Layout in dem Edge-Based-Layout2-Verfahren zu integrieren.

Es ist aus den o.g. Gründen daher nachvollziehbar, dass zwei Layoutverfahren mit der gleichen Komplexität nicht unbedingt gleich schnell in der Berechnung des neuen Layouts sein müssen. Daher gibt Tab. 2.16 ebenfalls die benötigte Zeit für das Zeichnen eines Graphen mit 100 Knoten je Layout an. Jede Layoutberechnung wurde dreimal durchgeführt und anschließend der Mittelwert aus den Ergebnissen gebildet. Getestet wurde auf einem Intel Core2 mit 1,83 GHz.

Allerdings stellt sich die Frage, ob das Anzeigen und Layouten von Graphen mit mehr als 1000 Knoten überhaupt Sinn macht. Denn Layoutbilder sind ausschließlich für den menschlichen Betrachter interessant, und ein zwar übersichtlich gezeichneter Graph, der allerdings mehr als 1000 Knoten besitzt, wird dennoch für den Benutzer unübersichtlich sein. Hierfür bieten sich Filter an, die uninteressante Teilstrukturen des Graphen ausblenden, hervorheben oder zusammenfassen können, welche ebenfalls im Rahmen dieser Diplomarbeit entwickelt wurden und im nächsten Kapitel ausführlich beschrieben werden.

3 Graph-Filter

3.1 Vorbemerkungen

Graph-Filter sind Methoden, die auf Graphstrukturen angewendet werden können, um für den Benutzer bedeutsame Substrukturen von den unwichtigen Informationen graphisch trennen zu können.

Graph-Filter können zur Darstellung bzw. Hervorhebung wesentlicher Teile des Graphen genutzt werden. Die für den Benutzer unwesentlichen Informationen könnnen dabei ausgeblendet oder auch zusammengefasst dargestellt werden. Die Entscheidungskriterien, nach denen bestimmte Substrukturen gefiltert werden, sind für alle Filter gleich. Diese könnten beispielsweise sein:

- Knotenbezeichnungen
- Knoten- oder Kantentypen
- Darstellungseigenschaften von Knoten
- Layerzugehörigkeit der Knoten

Es ist dem Geschmack des Benutzers überlassen, welcher Filter letztendlich zum Zuge kommt. Kombinationen der Filter sind ebenso möglich. Es wurden drei Filterarten definiert, die im Folgenden beschrieben werden.

3.2 Filterarten

3.2.1 Filter zum Ausblenden bestimmter Substrukturen

Manchmal ist es wichtig, wichtige Informationen von den unwichtigen zu trennen. Dabei müssen diese eindeutig unterscheidbar sein, sonst kann keine rechengestützte Auswertung erfolgen. Der Filter zum Ausblenden bestimmter Substrukturen kann die für die Darstellung unwichtigen Knoten und deren Beziehungen zueinander unsichtbar machen, um nur den interessanten Teil des Graphen darzustellen.

3.2.2 Filter zum Hervorheben bestimmter Substrukturen

Einen gegenteiligen Weg geht der Hervorhebungs-Filter. Hier werden nicht die Knoten ausgeblendet, welche den definierten Kriterien entsprechen, sondern mit geeigneten Darstellungsmitteln graphisch hervorgehoben. Dieser Filter kann Anwendung finden, wenn es sinnvoll ist, bestimmte Substrukturen des Graphen in Beziehung mit den Knoten und Kanten des gesamten Graphen zu sehen.

Die hervorzuhebenden Knoten können in ihrem Aussehen durch Farbe, Form oder Beschriftung verändert werden.

3.2.3 Filter zum Zusammenfassen bestimmter Substrukturen

Wenn es gewünscht ist, bestimmte Graph-Substrukturen auszublenden, sie aber nicht einfach nur verschwinden zu lassen, sondern durch einen Signalknoten anzeigen zu lassen, dass etwas ausgeblendet wurde, dann ist der Zusammenfassungs-Filter eine geeignete Methode. Anhand von festgelegten Kriterien werden bestimmte Knoten aus dem gezeichneten Graphen-Bild entfernt und stattdessen ein Zusammenfassungsknoten eingeblendet, der alle Kanten zu nicht-zusammengefassten Knoten besitzt, welche ursprünglich an den ausgeblendeten Knoten anlagen.

Desweiteren gibt es die Möglichkeit, diesen Zusammenfassungsknoten graphisch hervorzuheben, um ihn von echten Knoten des Graphen unterscheiden zu können. Ähnlich wie beim Hervorhebungs-Filter können Farbe, Form und/oder Beschriftung dieses Knotens verändert werden.

3.3 Implementierung in GroIMP

Es gibt in GroIMP bereits eine Klasse *GraphFilter*. Diese gibt die darzustellenden Knoten und Kanten des Graphen weiter an den Zeichenalgorithmus. Durch Aufruf der Methode *getLifeCycleState*, die als Parameter den zu untersuchenden Knoten oder die Kante verlangt, kann *GraphFilter* entscheiden, ob dieses Objekt angezeigt (Rückgabewert *PERSISTENT*) oder ausgeblendet (Rückgabewert *TRANSIENT*) werden soll. Diese Funktion muss von den erbenden Filterklassen implementiert werden.

Die Knoten eines Graphen in GroIMP besitzen bestimmte Attribute, wie z. B. Name, Beschriftung, Zugehörigkeit zu einer Klasse, Darstellungsgröße, -form oder -farbe. Sie können von Filtern mit der Methode *getObject* verändert werden. Diese Methode bekommt den Knoten, ein Attribut dieses Knotens und den aktuellen Graphstatus übergeben und kann als Rückgabewert den neuen Wert des Attributes zurückgeben. Der *GraphFilter* wertet dieses Attribut aus. Da es aber von Graph zu Graph unterschiedlich sein kann, welche und wie viele Attribute die Knoten besitzen, beschränken sich die im Folgenden beschriebenen Filterarten auf die wichtigsten und für die Filterung sinnvollen Attribute. Da sich GroIMP in ständiger Weiterentwicklung und Verbesserung befindet, kann nicht ausgeschlossen werden, dass in Zukunft weitere nützliche Knotenattribute hinzukommen werden. In solchen Fällen würde es keine große Mühe machen, diese ebenfalls einzubinden.

3.3.1 XML-Datenformat

Es musste ein Datenformat gefunden werden, welches so flexibel wie möglich die Definition von Filterkriterien und graphischen Darstellungseigenschaften erlaubt, eine Mehrfachausführung und Kombination der Filter nicht verbietet und nicht nur manuell erstellbar ist, sondern gegebenenfalls auch für größere Projekte dynamisch defniniert werden kann. Hier bot sich das XML-Format **??** geradezu an.

3.3.2 Beispiel

Anhand einer beispielhaften XML-Datei sollen im Folgenden die Möglichkeiten der Filterdefinitionen erklärt werden. Abb. 3.1 zeigt ein solches Beispiel.

Die Datei zum Definieren der Graph-Filter muss mit dem Tag

```
<HighlightFilter_Specification>
```

beginnen und mit dem entsprechenden Ende-Tag

```
</HighlightFilter_Specification>
```

aufhören.

Zwischen diesen beiden Tags können die Filter definiert werden. Dabei ist es unerheblich, ob nur ein, zwei, alle drei oder mehrere der gleichen Art definiert werden. Sie werden in der gleichen Reihenfolge, in der sie in der Datei stehen, auf den Graphen angewandt.

Die Beispiel-XML-Datei ist in 3 Sektionen unterteilt, je eine für die im Folgenden beschriebenen Filter. Jede Sektion hat einen Bereich zum Definieren der Kriterien, der Hervorhebungsfilter zusätzlich einen zum Festlegen, wie die Knoten, die mit den Kriterien überein-

```
<HighlightFilter_Specification>                                                    1
<Filter_Hide>                                                                      2
  <Criterion>                                                                      3
    <de.grogra.imp.caption compare = "=">Plane.2924</de.grogra.imp.caption>        4
  </Criterion>                                                                     5
</Filter_Hide>                                                                     6
<Filter_Highlight>                                                                 7
  <Criterion>                                                                      8
    <de.grogra.imp.caption compare="=">Sky*</de.grogra.imp.caption>                9
  </Criterion>                                                                     10
  <Criterion>                                                                      11
    <de.grogra.imp.caption compare ="=">*z</de.grogra.imp.caption>                 12
  </Criterion>                                                                     13
  <Result>                                                                         14
    <de.grogra.imp2d.shape>Octagon</de.grogra.imp2d.shape>                         15
    <de.grogra.imp.caption>TEST</de.grogra.imp.caption>                            16
    <de.grogra.imp.fillColor>0.11,0.22,0.33</de.grogra.imp.fillColor>              17
    <de.grogra.graph.layer>2</de.grogra.graph.layer>                               18
  </Result>                                                                        19
</Filter_Highlight>                                                                20
<Filter_Folding>                                                                   21
  <Criterion>                                                                      22
    <de.grogra.imp.caption compare="=">.</de.grogra.imp.caption>                   23
  </Criterion>                                                                     24
  <FoldedNode>                                                                     25
    <de.grogra.imp2d.shape>Rectangle</de.grogra.imp2d.shape>                       26
    <de.grogra.imp.caption>Folded</de.grogra.imp.caption>                          27
  </FoldedNode>                                                                    28
</Filter_Folding>                                                                  29
</HighlightFilter_Specification>                                                   30
```

Abbildung 3.1: Beispiel für eine XML-Datei zum Einlesen

stimmen, dargestellt werden sollen. Der Zusammenfassungsfilter braucht ebenfalls Angaben der Darstellungsart des Knoten, in denen alle Knoten, die den Suchkriterien entsprechen, zusammengefasst werden. Ausführlicheres dazu in den folgenden Unterkapiteln.

Das Suchen und auch Verändern der Knoten geschieht über deren definierte (zweidimensionale) Eigenschaften, wie z. B. Namen, zugehörige Knotenklasse, Farbe, Form, Größe, usw. Zur Zeit können folgende berücksichtigt werden:

- Alle Eigenschaften, die Zeichenketten (java.lang.String) als Werte haben, wie z. B.
  - de.grogra.imp.caption
  - de.grogra.imp.name
- Alle Eigenschaften, die ganze Zahlen, also Integerwerte (int) haben, wie z. B.
  - de.grogra.graph.layer
- Farbeigenschaften (javax.vecmath.Color3f), z. B.
  - de.grogra.imp.fillColor

- Formeigenschaften (java.awt.Shape), z. B.
  - de.grogra.imp2d.shape

Definition der Kriterien

Es gibt ein einheitliches Schema zum Angeben der Suchkriterien für alle Filter, die zu Beginn einer jeden Filterdefinition festgelegt werden. Diese Kriterien werden im folgenden Tag gebündelt:

```
<Criterion>
  Kriterium 1
  Kriterium 2
  ...
</Criterion>
```

Alle Kriterien, die so zusammengefasst wurden, sind mit einer UND-Verknüpfung verbunden, d.h. dass die zu suchenden Knoten im Graphen alle festgelegten Kriterien erfüllen müssen, um auf ihnen den Filter anwenden zu können. Im Beispiel 3.3.2 sind im *Filter_Highlight* zwei Criterion-Tags angegeben. Der Inhalt von mindestens einem dieser muss erfüllt sein, damit der Filter wirken kann. Mehrere Criterion-Tags entsprechen also einer ODER-Verknüpfung.

Ein Kriterium ist folgendermaßen aufgebaut:

```
<Attributname [compare="[=|!=|<|>|<=|>=]"]>Suchwert</Attributname>
```

Im *Filter_Hide* wird nach der Knotenbezeichnung gesucht, deren Name gleich *Plane.2924* sein soll. In diesem Fall ist der Attributname *de.grogra.imp.caption*, der Suchwert gleich *Plane.2924*. Die compare-Option funktioniert nur bei Attributen, die eine Zeichenkette oder eine Zahl zurückliefern. Als Vergleichsoperatoren für Zahlen sind das Gleichheitszeichen ('='), die Kleiner- ('<') und Größer-Als-Operatoren ('>'), sowie deren zusammengesetzte Operatoren ('<=', '>=') erlaubt. Als Vergleichsoperatoren für Zeichenketten sind Gleichheitszeichen ('=') und Ungleichheitszeichen ('!=') möglich.

Sollen, z. B. in der Knotenbezeichnung, nicht nur ganze Zeichenketten gesucht werden, sondern vielleicht nur bestimmte Wortbausteine, kann man den Platzhalter '*', der für eine beliebige Zeichenkette steht, benutzen. Für den *Filter_Highlight* wird im Beispiel nach Knotenbezeichnungen gesucht, die mit den Buchstaben '*Sky*' beginnen, oder mit '*z*' enden.

Anwendung des Ausblenden-Filters

Die Definition des Filters zum Ausblenden von bestimmten Knoten ist, wie in Abb. 3.1 gezeigt, so aufgebaut:

```
<Filter_Hide>
  Definition der Suchkriterien
</Filter_Hide>
```

Alle Knoten, die mit den Werten der Suchkriterien übereinstimmen, werden ausgeblendet, sind also innerhalb der Graphendarstellung nicht mehr sichtbar. Die den Suchkriterien entsprechenden Knoten bekommen in GroIMP den Status *TRANSIENT* zugewiesen, die so gekennzeichnet von der Darstellung auf der Zeichenfläche ausgenommen werden. Alle Kanten, die an diesen Knoten anliegen, werden ebenfalls entfernt.

Anwendung des Hervorhebungs-Filters

Dieser Filter hat folgende Struktur:

```
<Filter_Highlight>
  Definition der Suchkriterien
  <Result>
    <Attributname>Attributwert</Attributname>
    ...
  </Result>
</Filter_Highlight>
```

Im Result-Tag werden die graphischen Änderungen der Knoten, die den Suchkriterien entsprechen, definiert. Es können ein oder mehrere Attributveränderungen angegeben werden. Jeder hervorzuhebende Knoten bekommt alle Werte zugewiesen, die unter *Result* angegeben wurden; diese Werte sind also durch eine UND-Verknüpfung miteinander verbunden. Das Beispiel in Abb. 3.1 hat einen Hervorhebungsfilter definiert. Es wird geprüft, ob die Knotenbezeichnungen mit der Zeichenkette '*Sky*' beginnen oder mit dem Buchstaben '*z*' enden. Gibt es Knoten, die diese Kriterien erfüllen, werden deren graphische Eigenschaften verändert: Sie bekommen eine achteckige Form, erhalten die Bezeichnung '*Test*', werden mit einer bestimmten Farbe eingefärbt und werden dem Layer 2 zugeordnet.

Anwendung des Zusammenfassen-Filters

Der Bereich, der den Filter zum Zusammenfassen bestimmter Substrukturen definiert, ist folgendermaßen aufgebaut:

```
<Filter_Folding>
  Definition der Suchkriterien
  <Folded_Node>
    <Attributname>Attributwert</Attributname>
  </Folded_Node>
</Filter_Folding>
```

Alle Knoten, die mit den Werten der Suchkriterien übereinstimmen, werden in einem Knoten zusammengefasst. Das heißt, dass diese Knoten in der Graphdarstellung ausgeblendet werden und gleichzeitig ein neuer, mit Eigenschaftswerten, die im Folded-Node-Tag definiert worden sind, eingefügt wird, der die Kanten der unsichtbaren Knoten zu anderen noch vorhandenen Knoten erhält. Das Aussehen des neuen zusammengefassten Knoten kann genauso wie das Aussehen der gefundenen Knoten im Hervorhebungs-Filter verändert werden, nur diesmal anstatt eines *Result*-Tags ein *Folded_Node*-Tag.

Der zusammengefasste Knoten ist den anderen gegenüber gleichwertig. So lange der Filter aktiv ist, können auch Layoutalgorithmen die Knoten inklusive der Folding-Knoten neu anordnen, und machen keine Unterscheidung zwischen ihnen.

Ein Beispiel für den Zusammenfassungsfilter findet sich in Abb. 3.1. Alle Knoten, die einen Punkt in ihrer Bezeichnung haben, werden in einem neuen Knoten zusammengefasst, welcher rechteckig ist und die Aufschrift *Folded* trägt.

Filteranwendung auf einer Baumstruktur ab einem bestimmten Knoten

Ein besonderes Feature unterstützen alle drei Filterarten. Durch Angabe des Tags

```
<Subnodes/>
```

in einem der Filter wird festgelegt, dass nicht nur der Filter auf die Knoten angewendet wird, die den definierten Kriterien entsprechen, sondern auch auf deren Kinder, Enkel und alle weiteren Knoten bis zu den Blättern. Dies ist besonders dann sinnvoll, wenn ganze Teilbäume eines Graphen unwichtig sind und aus dem Layout entfernt werden können, bzw. wichtig sind und gemeinsam mit gleichen Attributen hervorgehoben werden sollen.

3.4 Klassenstruktur

Das folgende Diagramm zeigt in Form eines Klassendiagramms die Hierarchie und Abhängigkeiten der Filterklassen untereinander, sowie die Einordnung dieser Klassen zur Filterschnittstelle von GroIMP.

Öffentlich sichtbare Variablen sind mit # gekennzeichnet, Variablen, die nur innerhalb der Klasse, in der sie deklariert wurden, sichtbar sind, wurde ein - vorangestellt und die, die lediglich innerhalb ihres Paket verwendet werden können, haben ein +.

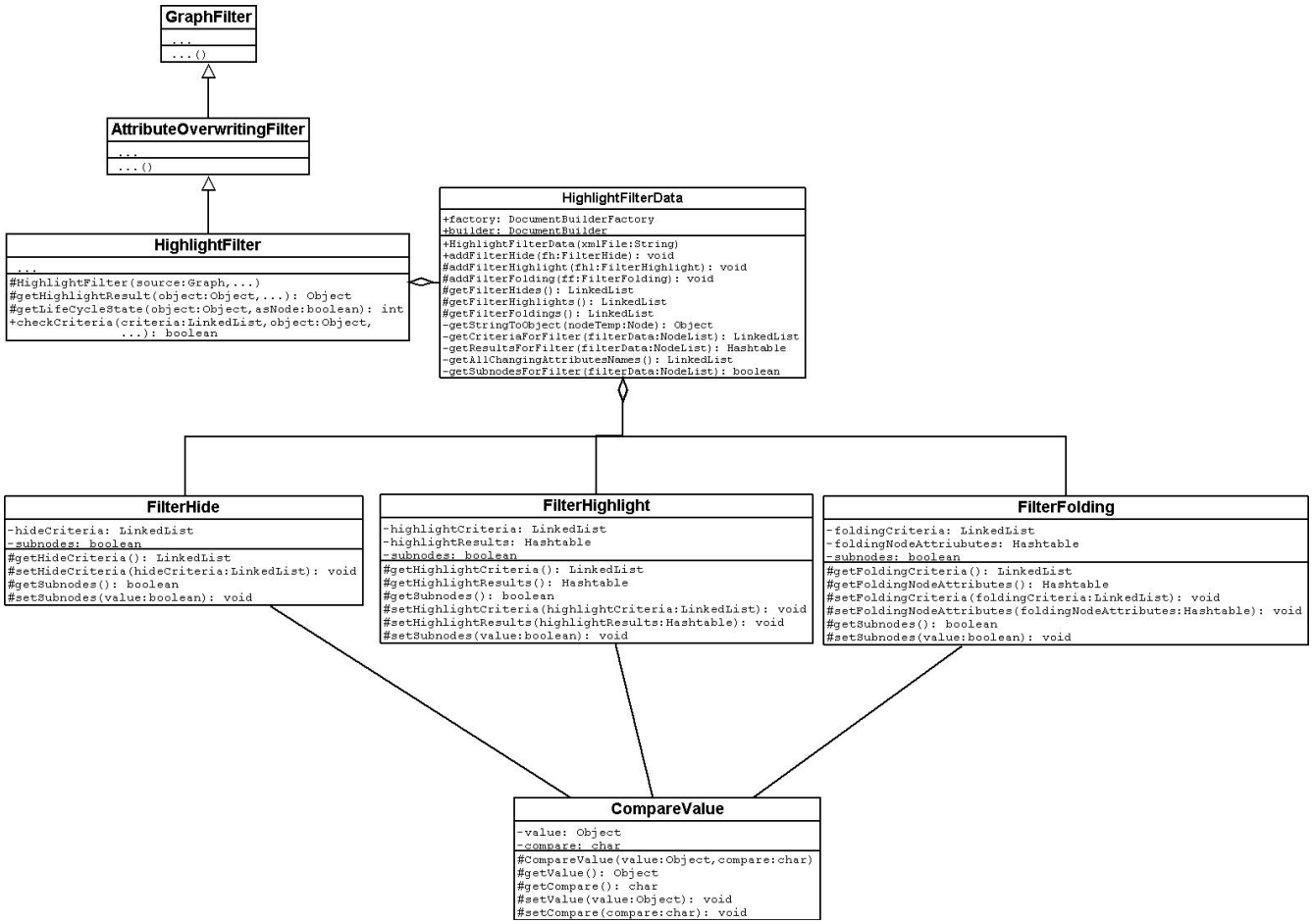

Abbildung 3.2: Übersicht der Filterklassen

3.5 Demonstration

Anhand eines Vorher-Nachher-Bildes soll die Wirkungsweise der Ausblenden- und Hervorhebungsfilter demonstriert werden. Die im vorhergehenden Abschitt diskutierte Spezifikations-XML-Datei wurde auf das Buschbeispiel ohne Regelanwendung ausgeführt. Der Knoten mit der Aufschrift *Plane.2924* verschwindet, der Knoten *Sky.2925* bekommt eine achteckige Form, wird blaugefärbt und bekommt die Bezeichnung '*Test*'.

Das Layout des Graphen wurde mit dem Square-Layout berechnet.

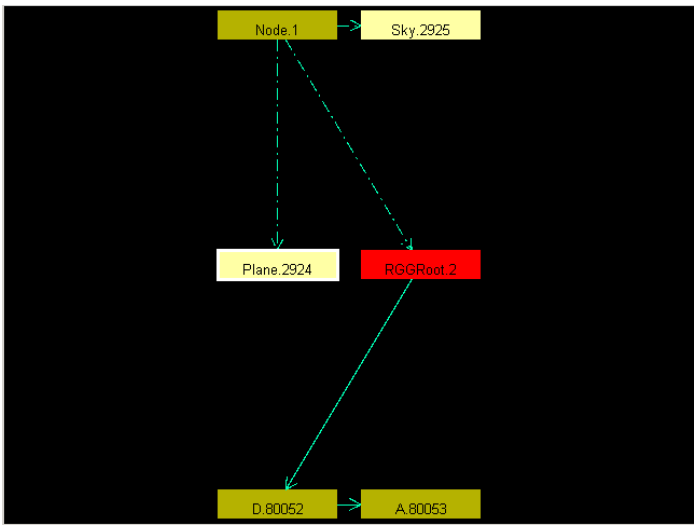

Abbildung 3.3: Graph vor Filteranwendung

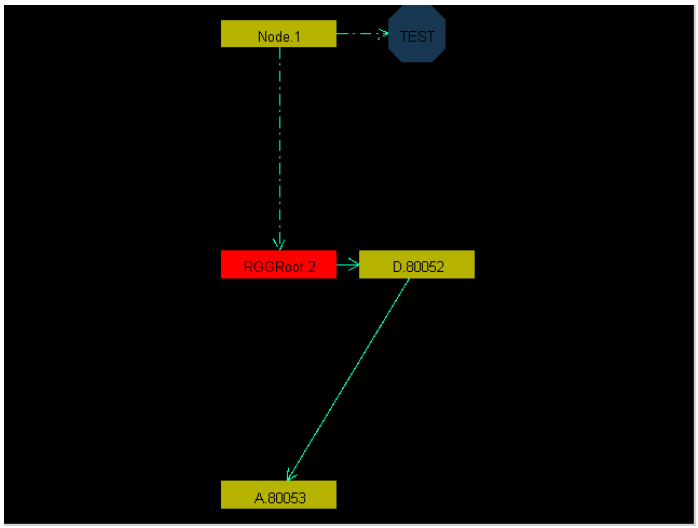

Abbildung 3.4: Graph nach Filteranwendung

3.6 Diskussion der Filterverfahren

Die vorgestellten Filter sind einfache, aber dennoch sehr nützliche Werkzeuge zur Verbesserung der Übersichtlichkeit des Graphenlayouts.

Die Struktur der XML-Datei, die die Filter definiert, ist einleuchtend und kann schnell an die eigenen Bedürfnisse angepasst werden. Bei sehr umfangreichen Filterdefinitionen könnte allerdings das Eintragen der Kriterien und Resultatergebnisse von Hand relativ ermüdend werden. Außerdem birgt das manuelle Ändern der XML-Datei auch die Gefahr von Syntaxfehlern, die erst zur Laufzeit von GroIMP vom Filterverfahren erkannt, aber natürlich nicht behoben werden können. Es gibt allerdings mittlerweile recht gute XML-Editoren auf dem Markt, die durchaus auf syntaktische Korrektheit prüfen können, wenn zuvor das XML-Schema definiert wurde [WI07a].

Durch die einheitliche Definition der Kriterien in allen drei Filterverfahren können diese sehr schnell untereinander ausgetauscht werden. Der *Result*-Bereich des Hervorhebungsfilters und der *FoldedNode*-Bereich des Folding-Filters sind ebenfalls in ihrer Struktur gleich.

Vor der Entwicklung der Filterverfahren wurde geprüft, welche Attributeigenschaften der Knoten zum Suchen (als Kriteriendefinition) und Verändern des Aussehens dieser bedeutsam sind. Es musste eine Einschränkung getroffen werden, da nur die wenigsten Knotenattribute vom Typ her einfache Datentypen, wie z. B. String, sind, und damit leicht implementierbar sind. Eine Berücksichtigung aller Attribute hätte nicht nur den Implementierungsaufwand extrem vergößert sondern auch die Ausführung der Filter träge gemacht. Dennoch ist es möglich, neue Typen von Attributen berücksichtigen zu lassen.

Eine interessante Folding-Möglichkeit, die im Rahmen dieser Diplomarbeit leider nicht umgesetzt werden konnte, ist das Zusammenfassen von Knoten mit gleichen Suchkriterien, die sich in dem gleichen Suchbaum befinden. Andere Knoten, die ebenfalls den Kriterien entsprechen, sich aber in einem anderen Teilbaum befinden, werden in einem neuen Knoten zusammengefasst.

4 Ausblick

In dieser Diplomarbeit wurden unterschiedliche Arten von Layoutalgorithmen besprochen, Beispiele ihrer Gattungen gezeigt und auch in GroIMP für die 2D-Ansicht der Graphen implementiert. Es steht nun eine breite Auswahl an Verfahren zur Verfügung, damit möglichst viele verschiedene Sichtweisen auf den darzustellenden Graphen gegeben sind. Bei der Anwendung in der Praxis hatte sich allerdings gezeigt, dass die reine Implementierung der Algorithmen selten ausreicht um ein befriedigendes Layoutergebnis zu erzielen. Aus diesem Grund entstanden z. B. die Fit-Funktion, Graphenrotationen (in einigen der Verfahren), aber auch das Setzen des *Node.1*-Knoten auf eine feste Position im Zeichenfenster. Es ist daher nicht auszuschließen, dass die Layoutalgorithmen auch in Zukunft weiter an die Bedürfnisse der Anwender angepasst werden müssen. Insbesondere eine große Anzahl von Kantenüberkreuzungen können sich störend auf das Gesamtbild des Graphen auswirken. Eine Layoualgorithmen versuchen diese bereits zu minimieren (vgl. Kap. 2.7). Die anderen Layoutalgorithmen könnten ebenfalls dahingehend optimiert werden, oder man implementiert neue Verfahren, die dieses Feature unterstützen, wie z. B. Branch-and-Cut-Algorithmen [ZI01]. Hierfür bietet GroIMP mit der Klasse *Layout* eine gute Implementierungsschnittstelle an.

Die implementierten Filterverfahren sind sehr allgemein gehalten, damit sie ohne viel Wissen über die Bedeutung der einzelnen Knoten und die Struktur des Graphen auf diesen angewendet werden können. Es ist zu überlegen, ob sie nicht dahingehend erweitert werden könnten, damit sie auf Wunsch auch spezielle Eigenschaften der Graphen von relationalen Wachstumsgrammatiken berücksichtigen können. Beispielsweise könnten sie zwischen normalen Knoten und 3D-Transformationsknoten unterscheiden.

Ebenso könnte man darüber nachdenken, ob man nicht durch eine Software die Spezifikations-XML-Datei automatisch generieren lassen kann. Sie könnte die Möglichkeiten der Filterverfahren aufzeigen und dem Benutzer zwischen den Varianten auswählen lassen. So könnte man syntaktisch falsche Angaben, die durch manuelles Editieren entstehen, vermeiden.

Literaturverzeichnis

[BE98] Frank Becker. *Formeln und Tabellen für die Sekundarstufe I und II*, Paetec GmbH, 6. Auflage, Berlin, 1998.

[BR99] Ilja Bronstein, Konstantin Semendjajew, Gerhard Musiol, Heiner Mühlig. *Taschenbuch der Mathematik*. Harr Deutsch, 4. Auflage, Frankfurt am Main, 1999.

[DB99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[DH96] Ron Davidson, David Harel. *Drawing Graphs Nicely Using Simulated Annealing*. In: ACM Transactions on Graphics, Vol. 15, Nr. 4, 1996, S. 301-331.

[FR91] Thomas M. J. Fruchterman, Edward M. Reingold. *Graph Drawing by Force-Directed Placement*. In: Software - Practice and Experience, Vol. 21, Nr. 11, 1991, S. 1129-1164.

[GW96] Bernhard Ganter, Rudolf Wille. *Formale Begriffsanalyse*. Springer, Berlin, 1996

[JG07] *JGraph*. URL: http://www.jgraph.com 20.05.2007.

[KBK06] Winfried Kurth, Gerhard Buck-Sorlin, Ole Kniemeyer. *Relationale Wachstumsgrammatiken: Ein Formalismus zur Spezifikation multiskalierter Struktur-Funktions-Modelle von Pflanzen*. In: Modellierung pflanzlicher Systeme aus historischer und aktueller Sicht. Symposium zu Ehren von Prof. Dr. Dr. h.c. Eilhard Alfred Mitscherlich, Schriftenreihe des Landesamtes für Verbraucherschutz, Landwirtschaft und Flurneuordnung Brandenburg, Reihe Landwirtschaft, Band 7, 2006, S. 36-45.

[KBK07] Winfried Kurth, Gerhard Buck-Sorlin, Ole Kniemeyer. *GroIMP*. URL: http://www-gs.informatik.tu-cottbus.de/grogra.de/software/groimp/, Lehrstuhl Grafische Systeme, BTU Cottbus, 06.05.2007.

[KK89] T. Kamada, S. Kawai. *An Algorithm for Drawing General Undirected Graphs*. In: Information Processing Letters, Vol. 31, Nr. 1, 1989, S. 7-15.

[KN06] Ole Kniemeyer. *Rule Based Modelling With the XL/GroIMP Software*. Lehrstuhl Grafische Systeme an der Brandenburgischen Technischen Universität Cottbus, 2004.

[LA03] Britta Landgraf. *BibRelEx: Erschließung bibliographischer Datenbasen durch Visualisierung von annotierten inhaltsbasierten Beziehungen*. Dissertation, Fakultät für Informatik der Technischen Universität München, 2003.

[NO03] Andreas Noack. *Energy Models for Drawing Clustered Small-World Graphs*. Computer Science Report, 07/03, BTU Cottbus, 2003.

[NW06] Martin Nöllenburg, Alexander Wolff. *A Mixed-Integer Program for Drawing High-Quality Metro Maps*, Proc. 13th Int. Symposium on Graph Drawing (GD 05), In: Lecture Notes in Computer Science, Vol. 3843, Springer Verlag, 2006, S. 321-333.

[SU02] Kozo Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*. World Scientific Publishing, Singapore, 2002,

[WI07a] Wikipedia. *XML-Editor*. URL: http://de.wikipedia.org/wiki/XML-Editor, 07.06.2007.

[WI07b] Wikipedia. *Lineare Interpolation*. URL: http://de.wikipedia.org/wiki/Interpolation, 07.06.2007.

[XM07] *XML*. URL: http://www.w3.org/XML, 20.05.2007.

[ZH06] Dexu Zhao. *Simulation und Visualisierung der Struktur und Dynamik metabolischer Netzwerke mit relationalen Wachstumsgrammatiken*. Diplomarbeit, Lehrstuhl Grafische Systeme an der Brandenburgischen Technischen Universität Cottbus, 2006.

[ZI01] Thomas Ziegler. *Crossing Minimization in Automatic Graph Drawing*. Dissertation, Technische Fakultät der Universität des Saarlandes, Saarbrücken, 2001.

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle wörtlich und sinngemäß übernommenen Textstellen als solche kenntlich gemacht habe.

Cottbus, 09.06.2007