Algorithmen zum schönen Zeichnen von Graphen und deren Integration in das RELVIEW-System

Diplomarbeit
von
Stefan Meier

Aufgabenstellung
Prof. Dr. Rudolf Berghammer

Betreuung
Prof. Dr. Rudolf Berghammer
Peter Schneider

Christian-Albrechts-Universität zu Kiel
Technische Fakultät
Institut für Informatik und Praktische Mathematik

15. Februar 1996

Inhaltsverzeichnis

1 Einleitung **3**

2 Begriffe der Graphentheorie **7**

3 Schöne Graphen **9**

4 Zeichnen azyklischer Graphen **12**
4.1 Einführung 12
4.1.1 Aesthetische Kriterien 12
4.1.2 Überblick über den Algorithmus 13
4.2 Optimale Rang-Einteilung 13
4.2.1 Was soll die Rang-Einteilung leisten? 13
4.2.2 Die Network-Simplex-Methode 14
4.2.3 Details der Implementierung 18
4.3 Die Ordnung der Knoten in den Schichten 20
4.4 X-Koordinaten 24
4.4.1 Was soll diese Phase des Algorithmus leisten? 24
4.4.2 Optimale X-Koordinaten-Zuweisung 25
4.4.3 Details der Implementierung 26
4.5 Erweiterung auf beliebige Graphen 27
4.6 Beispiele 27

5 Zeichnen von Bäumen **31**
5.1 Einführung 31
5.2 Ein naiver Baumzeichenalgorithmus 31
5.3 Knuths Algorithmus 34
5.4 Der Algorithmus von Wetherell und Shannon 35
5.5 Beispiele 39

6 Der Algorithmus KK **41**
6.1 Einführung 41
6.2 Das Federmodell 41
6.3 Lokale Minimierung der Gesamtenergie 42
6.4 Der Algorithmus 44
6.5 Beispiele 45

7 Der Algorithmus FR **48**
7.1 Einführung 48
7.2 Der Algorithmus 49
7.3 Wahl der Funktionen f_a, f_r und *cool* 51
7.4 Zeichnen nicht zusammenhängender Graphen 53
7.5 Beispiele 53

8 Relationen und RELVIEW **56**
8.1 Relationenalgebra 56
8.1.1 Basisoperationen auf Relationen 56
8.1.2 Spezielle homogene Relationen 57
8.1.3 Spezielle heterogene Relationen 58
8.1.4 Relationale Beschreibung von Teilmengen 58
8.1.5 Hüllen 58
8.1.6 Residuen und Quotienten 59
8.2 Das RELVIEW-System 59
8.2.1 Allgemeines 59
8.2.2 Das RELVIEW-Menü-Fenster 61
8.2.3 Das Directory-Fenster 63
8.2.4 Das Graphfenster 64
8.2.5 Das Relationenfenster 65
8.3 Eine Anwendung von RELVIEW 67
8.3.1 Schnittvervollständigung einer geordneten Menge 67
8.3.2 Relationaler Zugang zur Schnittvervollständigung 67
8.3.3 Beispiel 70

9 Abschließende Bemerkungen **75**

Kapitel 1

Einleitung

Wer mit Relationen- und Graphentheorie arbeitet, muß häufig Beispielrechnungen mit Relationen durchführen. Die Ausführung solcher Berechnungen mit Papier und Bleistift ist mühsam und fehleranfällig. Das RELVIEW-System bietet die Möglichkeit Berechnungen und Tests völlig interaktiv und bildschirmorientiert auf dem Rechner durchzuführen. Damit wird die Arbeit natürlich erleichtert, und die Korrektheit der Ergebnisse ist garantiert. Nun stellt sich aber noch ein weiteres Problem. In RELVIEW werden Relationen als Boolesche Matrizen dargestellt (nähere Einzelheiten zu RELVIEW siehe Kapitel 8). Hat man nun eine Berechnung mit RELVIEW durchgeführt, so erhält man die Ergebnisrelation in der folgenden Form (hier zum Beispiel eine Matrix der Dimension 30×30):

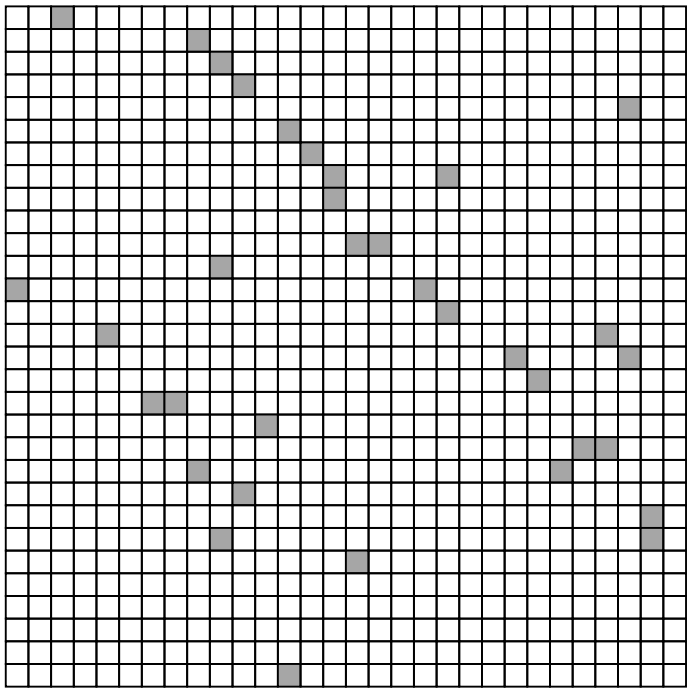


Abbildung 1.1: Eine Relation als Boolesche Matrix

Will man nun wissen, ob etwa das Feld in Zeile 13 und Spalte 19 schwarz oder

weiß ist, so ist dies schlecht auf den ersten Blick zu erkennen. Man muß also mühsam die Zeilen und Spalten abzählen, um das gewünschte Feld zu finden. Muß man dies für mehrere Felder tun, so ist das eine recht langwierige Aufgabe. Wesentlich einfacher wird die Aufgabe, wenn man das folgende Bild vorliegen hat, in dem die Relation als Graph dargestellt wird:

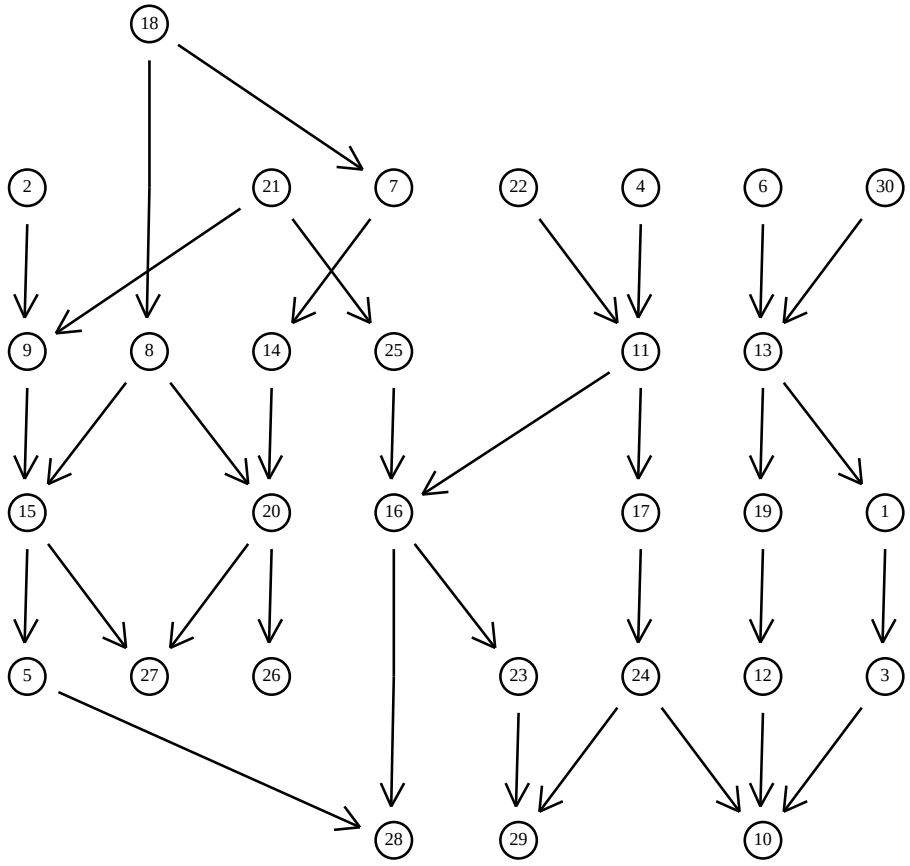


Abbildung 1.2: Eine Relation als Graph

Dieses Bild entspricht genau der Matrix aus Abb. 1.1. Das bedeutet, in Abb. 1.2 geht ein Pfeil vom Knoten x zum Knoten y, genau dann wenn in Abb. 1.1 das Feld in Zeile x und Spalte y schwarz ist. Hier läßt sich nun leicht erkennen, ob ein Pfeil von Knoten 13 zu Knoten 19 führt (ja, es gibt so einen Pfeil, also ist das Feld in Zeile 13 und Spalte 19 in Abb. 1.1 schwarz). Natürlich gibt es viele Möglichkeiten, diesen Graphen zu zeichnen. Man könnte ihn zum Beispiel auch wie in Abb. 1.3 darstellen. Diese Darstellung ist natürlich weit weniger übersichtlich als die in Abb. 1.2. Man kann schlecht sehen, ob ein Pfeil von Knoten a zu Knoten b geht, und zu erkennen, ob es einen Pfad von Knoten a zu Knoten b gibt, ist sehr schwer. Man würde deshalb sicherlich die erste Art der Darstellung bevorzugen.

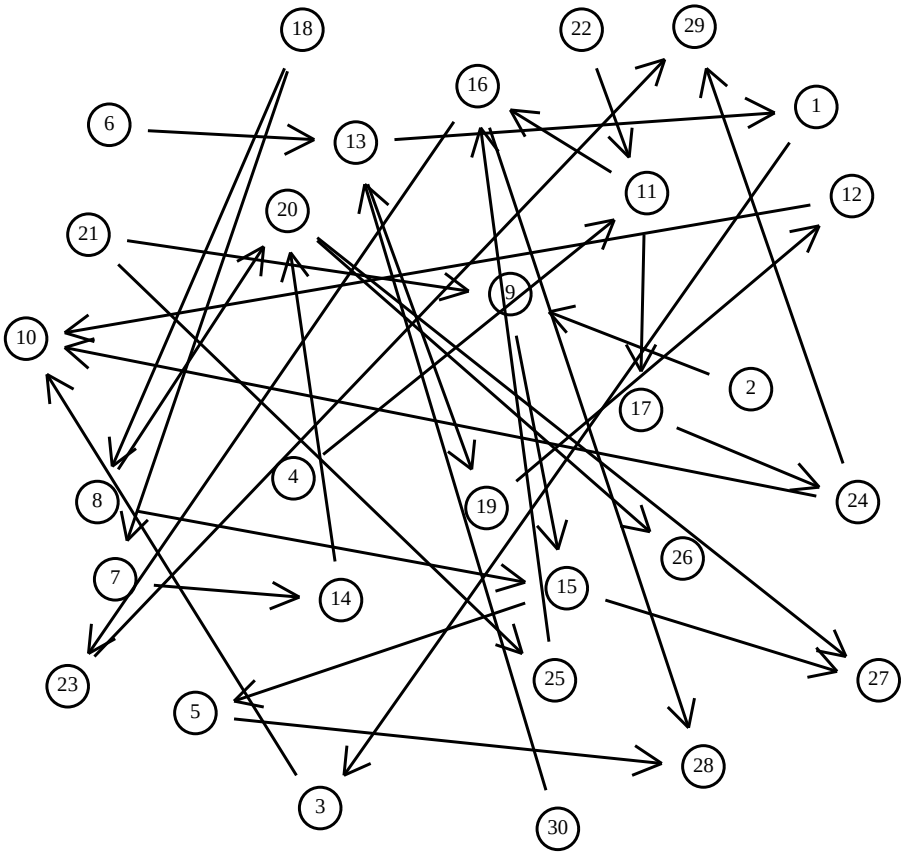


Abb. 1.3: Eine andere Darstellung des Graphen

Aufgabe der vorliegenden Arbeit ist es nun, Algorithmen auszuwählen, mit denen man Graphen schön zeichnen kann (was der Begriff "schön" hier genau bedeutet sehen wir in Kapitel 3). Die Algorithmen sollen erstens hier vorgestellt und zweitens ins RELVIEW-System eingebaut werden, um die Möglichkeit zu schaffen, quadratische Relationen als Graphen darzustellen.

Die Arbeit ist folgendermaßen gegliedert:

Kapitel 2 führt zunächst einige Begriffe aus der Graphentheorie ein, die im Verlauf der Arbeit benutzt werden (auch die bereits hier verwendeten Begriffe wie Graph oder Knoten).

In Kapitel 3 werden dann allgemeine Bemerkungen zum Thema "Schönes Zeichnen von Graphen" gemacht. Wie oben bereits gesehen, kann man Graphen ja auf verschieden Weise in der Ebene darstellen. Vergleicht man nun Abb. 1.2 und Abb. 1.3, so wird wohl jederman zustimmen, daß das Bild in 1.2 schöner ist als das in 1.3. In Kapitel 3 betrachten wir nun generelle Aussagen darüber, wann die Darstellung eines Graphen schön ist.

Die Kapitel 4–7 stellen die verschiedenen Algorithmen vor, die auch ins RELVIEW-System eingebaut sind.

In Kapitel 4 wird ein Algorithmus vorgestellt, der Graphen in Schichten zeichnet. Das heißt, die Knoten des Graphen werden auf Schichten verteilt und Knoten, die in einer Schicht liegen, werden auf ein und derselben Höhe gezeichnet. Mit diesem Algorithmus wurde auch Abb. 1.2 erzeugt.

Kapitel 5 beschreibt einen Algorithmus, der nur auf Bäume bzw. Wälder (d.h. Graphen, deren Zusammenhangskomponenten Bäume sind) anwendbar ist. Auch dieser Algorithmus stellt den Graphen, wie bei Bäumen üblich, in Schichten dar. Dies ist der einzige der hier vorgestellten Algorithmen, dessen Anwendung auf eine bestimmte Klasse von Graphen beschränkt ist.

Der Algorithmus aus Kapitel 6 zeichnet die Graphen nicht in Schichten, sondern verteilt sie gleichmäßig auf der Zeichenfläche. Dadurch werden die Bilder einerseits kompakter, andererseits aber auch unübersichtlicher.

In Kapitel 7 wird ein Algorithmus beschrieben, dessen Idee ähnlich ist wie die des Algorithmus aus Kapitel 6. Er läuft einerseits deutlich schneller, macht aber auch etwas schlechtere Bilder als der Algorihtmus aus Kapitel 6.

Das Kapitel 8 gibt eine kurze Beschreibung von RELVIEW. Dazu werden zuerst Begriffe der Relationentheorie eingeführt (RELVIEW ist ja zum Arbeiten mit Relationen entwickelt worden). Danach folgt eine kurze Beschreibung der Benutzung von RELVIEW, wobei insbesondere die Graphein- und Graphausgabe, berücksichtigt wird. Als letztes betrachten wir noch ein Beispiel für die Anwendung von RELVIEW.

Den Abschluß bildet Kapitel 9 mit einem Ausblick in mögliche Erweiterungen des Systems.

Kapitel 2

Begriffe der Graphentheorie

Dieser Abschnitt dient dazu, einige Begriffe zu klären, die im Verlaufe dieser Arbeit verwendet werden. In den folgenden Abschnitten wird dann die Bedeutung der hier eingeführten Begriffe nicht mehr explizit erläutert.

Ein ungerichteter Graph ist ein Paar $G = (V, E)$, wobei V eine nichtleere Menge und E eine Teilmenge von $\{\{x, y\} | x, y \in V \text{mit} x \neq y\}$ ist. Die Elemente von V heißen Knoten und die Elemente von E heißen Kanten von G. Sei nun $G = (V, E)$ ein ungerichteter Graph. Seien $u, v \in V$.

- Eine Folge $p = (v_0, v_1, \ldots, v_n)$ mit $v_0 = u$, $v_n = v$ und $\{v_i, v_{i+1}\} \in E$ für alle $i \in \{0, 1, \ldots, n-1\}$ heißt Pfad von u nach v in G der Länge n.
- Ein ungerichteter Graph heißt zusammenhängend, wenn es einen Pfad von u nach v gibt für alle $u, v \in V$.
- G heißt Baum, wenn G zusammenhängend ist und $|V| = |E| + 1$ gilt.

Sei die Relation $\sim$ auf V folgendermaßen definiert:

$$u \sim v \text{ gdw. es gibt einen Pfad von } u \text{ nach } v \text{ in } G.$$

Dann ist $\sim$ eine Äquivalenzrelation auf V. Seien $V_1, V_2, \ldots, V_m$ die Äquivalenzklassen von $\sim$ und seien $E_i = \{\{x, y\} \in E | x, y \in V_i\}$ für $i = 1, 2, \ldots, m$. Dann bilden die E_i eine Partition von E. Die Graphen $G_i = (V_i, E_i)$, $i = 1, 2, \ldots, m$ heißen die Zusammenhangskomponenten von G.

Ein gerichteter Graph ist ein Paar $G = (V, E)$, wobei V eine nichtleere Menge und E eine Teilmenge von $V \times V$ ist. Sei $e = (u, v) \in E$. Dann heißt u Anfangspunkt und v Endpunkt von e. Ist eine Unterscheidung in Anfangs- und Endpunkt nicht wichtig, so bezeichnet man auch beide Punkte als die Endpunkte der Kante e. Im folgenden bezeichnen wir gerichtete Graphen einfach als Graphen.

Sei $G = (V, E)$ ein Graph. Seien $u, v \in V$.

- Der G zugrundeliegende ungerichtete Graph ist definiert als (V, E') mit

$$E' = \{\{x, y\} | (x, y) \in E \text{ oder } (y, x) \in E \text{ und } x \neq y\}.$$

- Eine Folge $p = (v_0, v_1, \ldots, v_n)$ mit $v_0 = u$, $v_n = v$ und $(v_i, v_{i+1}) \in E$ für alle $i \in \{0, 1, \ldots, n-1\}$ heißt Pfad von u nach v in G der Länge n.
- Ist $u = v$ und $n \geq 1$, so nennt man den Pfad p einen Kreis.
- G heißt azyklisch, wenn es in G keinen Kreis gibt.
- G geißt zusammenhängend, wenn sein zugrundeliegender ungerichteter Graph zusammenhängend ist.
- Seien V_i die Knotenmengen der Zusammenhangskomponenten des G zugrundeliegenden ungerichteten Graphen und $E_i = \{(x, y) \in E | x, y \in V_i\}$. Dann sind die Graphen $G_i = (V_i, E_i)$ die Zusammenhangskomponenten von G.
- G heißt Baum, wenn G zusammenhängend ist, jeder Knoten nur Endpunkt höchstens einer Kante ist und es genau einen Knoten gibt, der nicht Endpunkt einer Kante ist.
- G heißt Wald, wenn seine Zusammenhangskomponenten Bäume sind.

Ein Graph $G = (V, E)$ heißt endlich, wenn V endlich ist. Im Rest der Arbeit werden wir es nur mit endlichen Graphen zu tun haben. Endliche Graphen lassen sich in der Ebene darstellen, indem man Knoten aus V durch Punkte und Kanten aus E durch Kurven zwischen den entsprechenden Punkten repräsentiert. Der Zweck von Graphausgabealgorithmen ist, das sie für endliche Graphen eine Darstellung in der Ebene liefern.

Kapitel 3

Schöne Graphen

Graphen sind ein oft benutztes Mittel zur Repräsentation aller Arten von Daten. Wir benutzen sie, um konkrete Relationen übersichtlicher darzustellen als z.B. durch Matrizen. Es gibt nun eine Fülle von Algorithmen zum Schönen Zeichnen von Graphen (siehe [1]), von denen hier vier vorgestellt werden sollen, die auch ins RELVIEW-System eingebaut sind.

Was sind nun schöne Graphen? In der Literatur gibt es viele verschiedene Kriterien, nach denen man beurteilt, ob das Bild eines Graphen schön ist. Diese hängen natürlich davon ab, mit welchen Mitteln der Graph dargestellt werden soll, für welche Zwecke das Bild benutzt wird und was für eine Art von Graph gezeichnet werden soll.

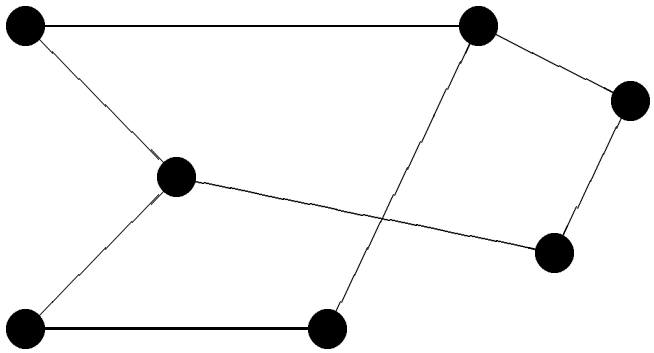


Abbildung 3.1: Geradlinige Darstellung

Es gibt verschiedene Standards zur Darstellung von Graphen in der Ebene. Üblicherweise werden die Knoten durch Kreise oder Kästen dargestellt, und eine Kante (u, v) wird durch eine Linie zwischen den Symbolen, die u und v darstellen, repräsentiert. Nach der Art, wie die Linien, die die Kanten repräsentieren, gezeichnet werden, kann man die Bilder zum Beispiel in folgende Kategorien einteilen. In einer geradlinigen Darstellung werden die Kanten nur durch gerade Linien repräsentiert, stellt man jede Kante durch einen Polygonzug dar, so spricht man von einer Polygondarstellung, und wenn die einzelnen Segmente

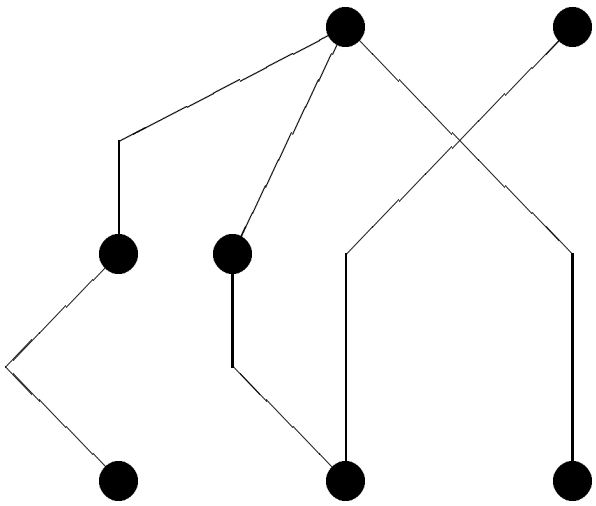


Abbildung 3.2: Polygondarstellung

des jeweiligen Polygonzuges nur horizontal oder vertikal verlaufen, spricht man von einer Orthogonaldarstellung (siehe dazu Abb. 3.1 – 3.3).

Die in dieser Arbeit dargestellten Algorithmen liefern Bilder, die entweder eine Polygon- oder eine geradlinige Zeichnung sind. Wann ist nun eine Darstellung eines Graphen schön? Auf jeden Fall möchte man, daß ein Graph gut lesbar ist, das heißt, daß man schnell erkennen kann, welche Knoten durch Kanten miteinander verbunden sind bzw. zwischen welchen Knoten es einen Pfad in dem Graphen gibt. Daraus ergeben sich zum Beispiel die folgenden Forderungen an das Bild eines Graphen.

- Die Kanten sollten möglichst kurz und gerade sein,
- Knoten sollten nicht zu dicht nebeneinander liegen und
- die Kanten sollten sich möglichst wenig schneiden.

Weitere mögliche Wünsche sind,

- daß im Graph vorhandene Symmetrien sich auch im Bild wiederspiegeln,
- daß isomorphe Teilgraphen stets gleich dargestellt werden, egal an welcher Stelle sie im Graphen auftreten und
- daß das Bild so wenig Platz wie möglich beansprucht.

Zusätzlich kommen für spezielle Arten von Graphen noch spezifische Anforderungen hinzu. So will man bei gerichteten Bäumen gerne den Vater über seinen Söhnen zentriert haben, bei azyklischen Graphen sollen alle Kanten die gleiche Richtung haben und planare Graphen sollen auf jeden Fall auch planar, das heißt, ohne daß sich zwei Kanten schneiden, dargestellt werden. Es ist nicht zu erwarten, daß ein Algorithmus alle diese Anforderungen an die Darstellung von Graphen erfüllen kann, und so gibt es auch eine Fülle von Algorithmen zum

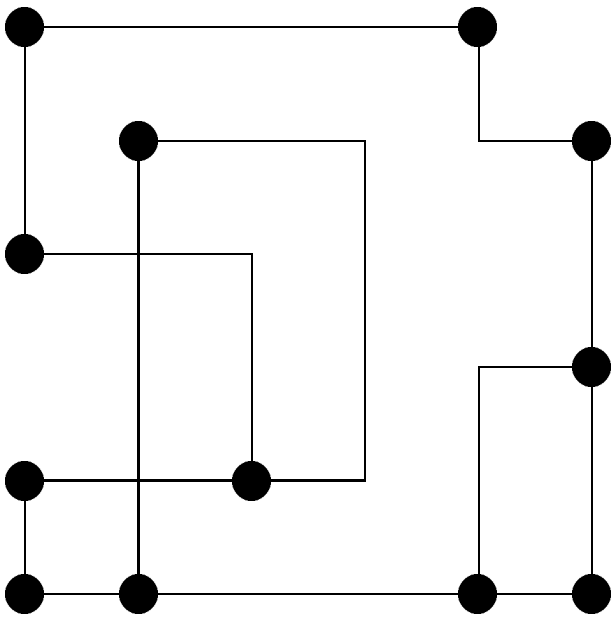


Abbildung 3.3: Orthogonaldarstellung

Zeichnen von Graphen, die dann jeweils bei einigen der genannten Kriterien ihre Schwerpunkte setzen und andere ganz oder teilweise vernachlässigen. Und selbst bei den Schwerpunkten der Algorithmen ist nicht immer Optimalität der Ergebnisse zu erwarten, da zum Beispiel die Minimierung von Kantenkreuzungen NP–vollständig ist [5], also mit effizienten Algorithmen nicht immer ein optimales Ergebnis erreicht werden kann. Bei den Algorithmen, die hier vorgestellt werden, wird jeweils erwähnt, welche Kriterien bei der Entwicklung des Algorithmus eine Rolle spielten.

Kapitel 4

Zeichnen azyklischer Graphen

4.1 Einführung

Der hier vorgestellte Algorithmus stammt hauptsächlich aus einer Arbeit von Gansner et al. [10].

4.1.1 Aesthetische Kriterien

Gerichtete azyklische Graphen (kurz dags - directed acyclic graphs) haben die Eigenschaft, daß man sie so darstellen kann, daß alle Kanten des Graphen eine gemeinsame Richtung haben, wie zum Beispiel von links nach rechts oder von oben nach unten (wie die meisten Beispielgraphen hier). Diese Tatsache hat zur Entwicklung einiger Algorithmen zum Zeichnen von dags geführt, die auf den folgenden Prinzipien basieren:

A1: Stelle den Graph als eine Hierarchie dar, sorge insbesondere dafür, daß alle Kanten dieselbe Richtung haben. Dies hilft, Pfade zu finden, und hebt Quellen und Senken besonders hervor.

A2: Vermeide Anomalien im Bild, die es unübersichtlich machen, wie zum Beispiel Kreuzungen von Kanten oder besonders stark geknickte Kanten.

A3: Male die Kanten möglichst kurz. Das macht es leichter benachbarte Knoten zu finden, und ist außerdem im Sinne von A2.

A4: Achte auf Symmetrie und Ausgewogenheit im Bild. Dieser Punkt spielt an manchen Stellen in dem hier vorgestellten Algorithmus nur eine untergeordnete Rolle.

Wie bereits in 3 erwähnt, ist es natürlich nicht möglich, den Graph bezüglich all dieser Punkte optimal darzustellen. Daher werden an einigen Stellen vereinfachende Annahmen gemacht und Heuristiken verwendet, die schnell laufen und in den meisten Fällen gute Ergebnisse liefern.

```
1. procedure draw_graph()
2. begin
3.    rank();
4.    ordering();
5.    position();
6. end
```

Abbildung 4.1: Gesamtalgorithmus

4.1.2 Überblick über den Algorithmus

Die Eingabe, auf der der Algorithmus arbeitet, ist ein Graph $G = (V, E)$, dabei ist V die Knotenmenge und E die Kantenmenge. Wir nehmen an, daß der Graph zusammenhängend ist. Ist dies nicht der Fall, so können die einzelnen Zusammenhangskomponenten getrennt dargestellt werden. Der Algorithmus hat, wie in Abb. 4.1 zu sehen, drei Phasen. In der ersten Phase wird jedem Knoten ein Rang zugewiesen. Knoten mit gleichem Rang werden auf einer Höhe, sozusagen in einer Schicht gezeichnet. Das heißt, hiermit werden auch gleichzeitig die Y-Koordinaten bestimmt. In der zweiten wird die Ordnung der Knoten innerhalb der Schichten festgelegt, so daß Kreuzungen von Kanten möglichst vermieden werden, und in der letzten Phase werden schließlich die X-Koordinaten der Knoten festgelegt.

4.2 Optimale Rang-Einteilung

4.2.1 Was soll die Rang-Einteilung leisten?

Diese erste Phase des Algorithmus weist jedem Knoten $v \in G$ einen ganzzahligen Rang $\lambda(v)$ zu, so daß er mit den zu v inzidenten Kanten verträglich ist. Das bedeutet, daß für jede Kante $e = (v, w) \in E$ gilt $l(e) \geq \delta(e)$, wobei die Länge $l(e)$ von $e = (v, w)$ als $\lambda(w) - \lambda(v)$ definiert ist, und $\delta(e)$ die Mindestlänge der Kante e angibt. $\delta(e)$ ist normalerweise gleich 1, wird aber aus technischen Gründen, wie wir später sehen werden, bei einigen Gelegenheiten programmintern anders gesetzt.

A3 aus 4.1.1 besagt, daß die Kantenlängen möglichst klein sein sollen. Abgesehen davon, daß dies schönere Bilder liefert, verkürzt sich mit der Kantenlänge auch die Laufzeit der noch folgenden Phasen des Algorithmus, da diese von der Gesamt-Kantenlänge abhängt. Somit ist es natürlich wünschenswert, eine optimale Rang-Einteilung zu finden, das heißt eine bei der die Summe der gewichteten Kantenlängen minimal wird.

Das Finden einer optimalen Rang-Einteilung kann auch durch das folgende ganzzahlige lineare Programm beschrieben werden:

$$\min \sum_{(v,w)\in E} \omega(v, w)(\lambda(w) - \lambda(v))$$

$$\text{wobei } \lambda(w) - \lambda(v) \geq \delta(v, w) \text{ für alle } (v, w) \in E$$

Die Gewichtsfunktion ω ist hierbei wiederum in dem Ausgangsgraphen gleich 1 für alle Kanten, wird aber intern später anders gesetzt. Die Gewichtsfunktion ω und die Funkton δ, die die minimale Länge angibt, bilden die Menge der Kanten E auf die nichtnegativen rationalen Zahlen bzw. auf die natürlichen Zahlen ab. Es gibt nun verschiedene Wege, dieses ganzzahlige Programm zu lösen. Eine Möglichkeit wäre zum Beispiel, das äquivalente lineare Programm zu lösen und dann diese Lösung in eine ganzzahlige zu transformieren (in polynomialer Zeit).

4.2.2 Die Network-Simplex-Methode

Dies ist ein einfacher Zugang zu dem Problem, der auf dem Network-Simplex-Algorithmus aus [3] basiert. Obwohl seine Komplexität nicht als polynomial bewiesen wurde, braucht er in der Praxis nur wenige Iterationen und läuft sehr schnell.

Für die Beschreibung des Algorithmus benötigen wir zunächst folgende Definitionen: Eine Rang-Einteilung ist zulässig, wenn sie die Bedingung $l(e) \geq \delta(e)$ erfüllt. Die Spanne $slack(e)$ einer Kante e ist die Differenz zwischen ihrer Länge und ihrer minimalen Länge, also $slack(e) = l(e) - \delta(e)$. Eine Kante heißt passend, wenn ihre Spanne Null ist.

Ein spannender Baum induziert eine Rang-Einteilung, oder besser eine Familie von Rang-Einteilungen. (Beachte, daß der spannende Baum nicht unbedingt ein gerichteter Baum ist, und nur auf dem zu Grunde liegenden ungerichteten Graphen basiert.) Man erhält diese Rang-Einteilung, indem man zuerst einem beliebigen Knoten irgendeinen Rang zuweist, und dann die Rang-Einteilung für die restlichen Knoten nach folgendem Schema durchführt: Sei v ein Knoten, der noch keinen Level hat, und w ein mit v durch eine Baumkante verbundener Knoten, dem berets ein Rang zugewiesen wurde. Der Wert, der v als Rang zugewiesen wird, ist dann der Rang von w erhöht bzw. vermindert um die Mindestlänge der Kante zwischen v und w, und zwar erhöht, wenn die Kante von w nach v geht, und vermindert, wenn sie von v nach w geht. Dies führt man so lange fort, bis man allen Knoten einen Rang zugewiesen hat. Ein spannender Baum heißt zulässig, wenn er eine zulässige Rang-Einteilung induziert. Durch diese Konstruktion der Rang-Einteilung sind automatisch alle Baumkanten passend.

Mit einem gegebenen spannendem Baum können wir jeder Baumkante e einen ganzzahligen Schnittwert wie folgt zuordnen: Löscht man die Kante e aus dem Baum, so zerfällt der Baum in zwei Zusammenhangskomponenten, die Vorder- und die Hinterkomponente. Die Vorderkomponente enthält den End- und die Hinterkomponente den Anfangsknoten der Kante e. Der Schnittwert von e ist definiert als die Summe der Gewichte aller Kanten, die von der Hinter- zur Vorderkomponente gehen (das heißt auch der Kante e selbst) minus der Summe der Gewichte aller Kanten, die von der Hinter- zur Vorderkomponente gehen.

Ein negativer Schnittwert einer Baumkante deutet darauf hin (wegen Degeneriertheit muß dies aber nicht immer der Fall sein), daß die Summe der gewichteten Kantenlängen reduziert werden kann, indem diese Kante so weit wie möglich gestreckt wird. So weit wie möglich bedeutet, bis eine der Kanten, die von der Vorder- zur Hinterkomponente gehen, passend wird. Dies entspricht

```
1. procedure rank()
2.   feasible_tree();
3.   while (e = leave_edge()) ≠ nil do
4.     f = enter_edge(e);
5.     exchange(e,f);
6.   end
7.   normalize();
8.   balance();
9. end
```

Abbildung 4.2: Network-Simplex

dem Ersetzen der Baumkante mit dem negativen Schnittwert durch die neue dann passende Kante, um so einen neuen zulässigen spannenden Baum zu erhalten. Diese Feststellung ermöglicht es uns, das Problem der Rang-Einteilung graphentheoretisch statt mit algebraischen Mitteln zu lösen. Baumkanten mit negativen Schnittwerten werden sukzessive durch geeignete Nicht-Baumkanten ersetzt, bis alle Baumkanten nicht-negative Schnittwerte haben. Theoretisch muß man, um die Terminierung des Algorithmus zu garantieren, natürlich eine Technik verwenden, die das Kreisen verhindert. In der Praxis ist dies jedoch nicht unbedingt erforderlich, da Fälle, in denen das Kreisen auftritt, nur sehr schwer zu konstruieren sind, und deshalb praktisch nie auftreten. Für genauere Informationen über den Network-Simplex-Algorithmus, was Terminierung und Optimalität der Lösung betrifft, verweise ich auf die Literatur [3, 4].

Abb. 4.2 zeigt die Version des Network-Simplex-Algorithmus von Gansner et al.:

2: Die Funktion `feasible_tree` erzeugt einen zulässigen spannenden Baum. Diese Funktion wird gleich noch ausführlich beschrieben. Die Simplex-Methode beginnt mit einer zulässigen Lösung und behält dies bis zum Schluß bei.

3: `leave_edge` liefert eine Baumkante mit negativem Schnittwert, falls eine solche vorhanden ist, oder `nil`, falls nicht, was bedeutet, daß die Lösung optimal ist. Jede Kante mit negativem Schnittwert kann als die zu ersetzende Kante ausgewählt werden.

4: `enter_edge` findet eine Kante, die nicht zum Baum gehört, durch die `e` ersetzt wird. Dazu wird die Kante `e` aus dem Baum gelöscht, wodurch der Baum in eine Vorder- und eine Hinterkomponente geteilt wird. Von allen Kanten, die von der Vorder- zur Hinterkomponente gehen, wird eine mit minimaler Spanne ausgewählt. Dies ist nötig, damit der entstehende Baum wieder zulässig ist.

```
1.  procedure  feasible_tree()
2.    init_rank();
3.    tree_init();
4.    while treesize < |V| do
5.      e = eine Nicht-Baumkante, die einen Endknoten im Baum
6.            und minimale Spanne hat;
7.      delta = slack(e);
8.      if der Baumknoten von e ist Endknoten von e then
9.        delta = -delta;
10.     for v in tree do v.rank = v.rank + delta;
11.     füge den Nicht-Baumknoten von e in den Baum ein
12.     mit e als neuer Baumkante;
13.     treesize = treesize + 1;
14.   end
15. init_cutvalues();
16. end
```

Abbildung 4.3: Die Prozedur feasible_tree

5: Die Kanten `e` und `f` werden ausgetauscht. Dazu wird der Baum mit den Schnittwerten entsprechend geändert.

7: Die Funkton `normalize` setzt den niedrigsten Rang auf Null.

8: Knoten, bei denen die Summe der Kantengewichte aller Kanten, die zu ihnen hingehen, gleich der Summe der Kantengewichte aller Kanten, die von ihnen weggehen, ist und die mehrere zulässige Ränge haben, werden auf einen zulässigen Rang mit den wenigsten Knoten verschoben. Dies soll im Sinne von Prinzip A4 aus 4.1.1 verhindern, daß die Knoten sich mehr als nötig in einer Schicht häufen. Diese Korrekturen verändern die Kosten der Rang-Einteilung nicht. Die Knoten werden einfach nach einem gierigen Algorithmus verschoben, was zu zufriedenstellenden Ergebnissen führt.

Abb. 4.3 beschreibt den Algorithmus zum Konstruieren eines initialen zulässigen Baumes:

2: Hier wird eine initiale zulässige Rang-Einteilung konstruiert. Dies geschieht folgendermaßen: Die Knoten werden in einem FIFO-Puffer verwaltet. Knoten werden in den Puffer getan, wenn sie keine unmarkierten Eingangskanten haben, gleichzeitig werden ihre Ausgangskanten markiert. Wenn alle Knoten im FIFO-Puffer sind, werden sie in umgekehrter Reihenfolge wieder entnommen, wobei jedem Knoten der niedrigste Rang zugeordnet wird, der der Mindestlänge seiner Eingangskanten genügt.

3: `tree_init` stellt einen initialen Baum her, der aus einem beliebigen Knoten besteht. `treesize` wird auf 1 gesetzt.

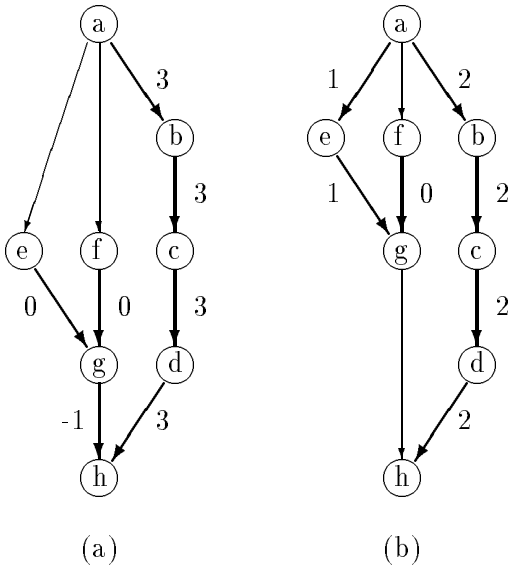


Abbildung 4.4: Beispiel zum Network-Simplex-Algorithmus

5-13: Hier wird eine Nicht-Baumkante gesucht, die in den Baum hineinführt, das heißt, die genau einen Endknoten im Baum hat. Dann wird der Rang der Baumknoten so verändert, daß die neue Baumkante passend wird. Da die Kante so gewählt wurde, daß sie eine minimale Spanne hat, ist die neue Rang-Einteilung immer noch zulässig. In jeder Iteration wächst der Baum also um einen Knoten, und am Schluß erhält man einen zulässigen spannenden Baum. Diese Technik wird ähnlich auch von Sugiyama in [2] beschrieben.

15: Die Funktion `init_cutvalues` berechnet die Schnittwerte der Baumkanten. Für jede Baumkante erhalten die Knoten dazu eine Marke, die angibt, ob der Knoten zur Vorder- oder Hinterkomponente gehört. Dann wird Summe der vorzeichenbehafteten Kantengewichte der Kanten gebildet, deren Anfangs- und Endknoten in verschiedenen Komponenten liegen, wobei das Vorzeichen negativ ist, wenn die Kante von der Vorder- zur Hinterkomponente geht.

Ein kleines Beispiel für den Network-Simplex-Algorithmus ist in Abb. 4.4 zu sehen. Nicht-Baumkanten sind durch dünne Linien dargestellt, das Gewicht und die minimale Länge aller Kanten ist gleich 1. Bild (a) zeigt den Graph nach der initialen Rang-Einteilung mit den entsprechenden Schnittwerten. Zum Beispiel ist der Schnittwert der Kante (g, h) gleich -1. Dies ergibt sich aus dem Gewicht der Kante (g, h) (von der Hinter- zur Vorderkomponente) minus der Kantengewichte von (a, e) und (a, f) (von der Vorder- zur Hinterkomponente). In Bild (b) wurde die Kante (g, h) mit negativem Schnittwert durch die Kante (a, e) ersetzt, wobei sich die gezeigten neuen Schnittwerte ergeben. Da alle Schnittwerte nichtnegativ sind, ist die Lösung optimal, und der Algorithmus terminiert.

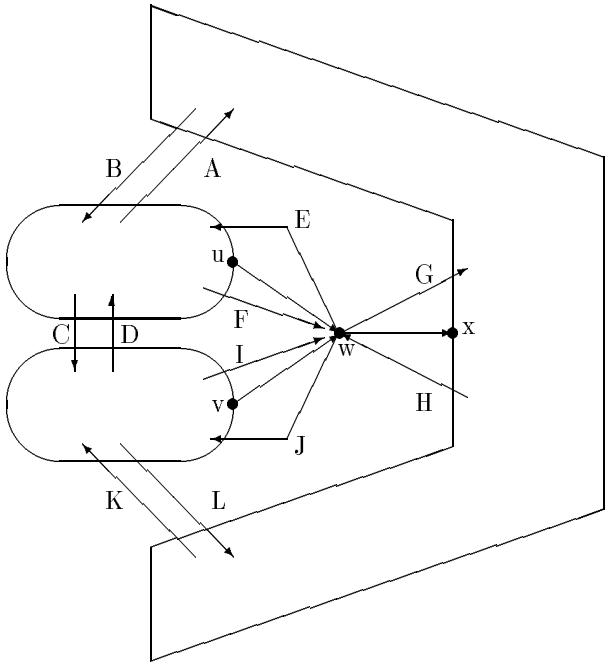


Abbildung 4.5: Berechnung der Schnittwerte

4.2.3 Details der Implementierung

Hier werden einige spezielle Punkte angesprochen, die bei einer Implementierung des Network-Simplex-Algorithmus wichtig sind, um die Laufzeit zu verringern. Diese Verbesserungen sind hier ganz nützlich, werden aber unbedingt notwendig in der dritten Phase des Algorithmus, wenn die Network-Simplex-Methode auf sehr viel größere Graphen angewandt wird.

Die Berechnung des initialen zulässigen Baumes und der ersten Schnittwerte benötigt oft einen sehr großen Anteil an der Gesamtrechenzeit zur Lösung eines Problems mithilfe des Network-Simplex-Algorithmus. Tatsächlich ist es in der Praxis häufig so, daß die Anfangslösung nicht weit von der Optimalen entfernt ist, und es nur noch einiger weniger Iterationen bedarf, um diese zu finden. In einer naiven Implementierung kann man die Schnittwerte berechnen, indem man für jede Kante alle Knoten markiert, je nachdem ob sie zur Vorder- oder zur Hinterkomponente gehören, und dann die Summe berechnet, wie vorhin bereits beschrieben.

Um die Kosten dieser Berechnung zu reduzieren, hilft uns die Feststellung, daß zur Berechnung der Schnittwerte nur zur jeweiligen Kante lokale Informationen nötig sind (genauer, man braucht nur Kanten betrachten, die einen der Endpunkte der Baumkante enthalten), wenn man bei den Blättern des Baumes anfängt und dann weiter nach innen geht. Es ist einfach, die Schnittwerte einer Kante zu berechnen, die ein Blatt des Baumes als Endknoten hat, da entweder die Vorder- oder die Hinterkomponente nur aus einem einzigen Knoten besteht. Hat man nun zu einem Knoten die Schnittwerte aller Kanten, die ihn enthalten, bis auf eine berechnet, so ist der Schnittwert der letzten Kante die Summe der Schnittwerte der anderen Kanten plus einem Term, der jetzt nur noch von den

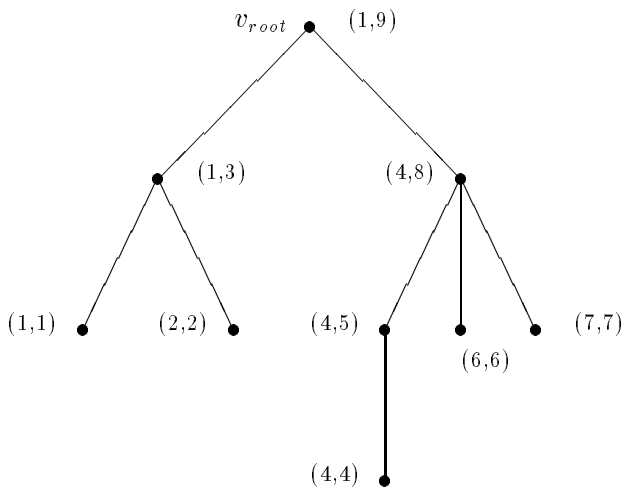


Abbildung 4.6: Postorder-Durchlauf mit Knotenmarkierungen (low,lim)

Kanten, die diesen Knoten enthalten, abhängt.

Diese Berechnung wird in Abb. 4.5 illustriert, für den speziellen Fall, wo ein Knoten w Endpunkt von genau drei Baumkanten ist. Wir nehmen an, daß die Schnittwerte von (u,w) und (v,w) bereits bekannt sind. Die Kanten, die mit Großbuchstaben beschriftet sind, stehen für die Menge aller Kanten in der dargestellten Richtung, deren Endpunkte in den entsprechenden Komponenten liegen. Die Schnittwerte von (u,w) und (v,w) berechnen sich wie folgt:

$$c_{(u,w)} = \omega(u,w) + A + C + F - B - E - D$$

beziehungsweise

$$c_{(v,w)} = \omega(v,w) + L + I + D - K - J - C \ .$$

Der Schnittwert von (w,x) ist demnach

$$\begin{aligned} c_{(w,x)} &= \omega(w,x) + G - H + A - B + L - K \\ &= \omega(w,x) + G - H + (c_{(u,w)} - \omega(u,w) \\ &\quad -C - F + E + D) + (c_{(v,w)} - \omega(v,w) - I - D + J + C) \\ &= \omega(w,x) + G - H + c_{(u,w)} - \omega(u,w) + c_{(v,w)} \\ &\quad -\omega(v,w) - F + E - I + J \ , \end{aligned}$$

ein Ausdruck, der nur die bereits bekannten Schnittwerte und lokale Informationen über die betrachtete Kante (w,x) benötigt. Berechnet man die Schnittwerte auf diese Weise, so wird jede Kante nur zweimal betrachtet, nämlich genau dann, wenn bei der Berechnung des Schnittwertes einer Baumkante einer der Endpunkte die Funktion des Knotens w aus obigem Beispiel hat.

Eine weitere Verbesserung, die einer Technik aus [3] entspricht, läßt sich durch die Durchführung eines Postorder-Durchlaufs auf dem spannenden Baum erreichen. Dabei beginnt man mit irgendeinem festen Knoten v_{root} als Wurzel, und markiert jeden Knoten v mit der Postorder-Nummer $lim(v)$, die angibt,

als wievielter Knoten er besucht wurde, mit $low(v)$, der niedrigsten Postorder-Nummer aller Nachfolger von v, und mit $parent(v)$, der Kante, über die v erreicht wurde (siehe Abb. 4.6). Dies versetzt uns in die Lage, sehr einfach testen zu können, ob ein Knoten in der Vorder- oder in der Hinterkomponente einer Baumkante liegt, und somit, ob eine Nicht-Baumkante die beiden Komponenten verbindet. Sei zum Beispiel $e = (v, w)$ eine Baumkante und v_{root} in ihrer Vorderkomponente (das heißt $lim(u) < lim(v)$). Dann liegt ein Knoten w genau dann in der Hinterkomponente von e, wenn gilt $low(u) \leq lim(w) \leq lim(u)$. Diese Parameter können auch benutzt werden, um den Baum nach einem Schritt im Network-Simplex-Algorithmus wieder zu aktualisieren(in `exchange(e,f)`). Wenn $f = (w, x)$ die neue Baumkante ist, dann müssen nur die Schnittwerte auf dem Pfad zwischen w und x (im Baum) korrigiert werden. Dieser Pfad ist leicht zu finden, indem man die $parent$-Kanten von w und x bis zum ersten gemeinsamen Vorgänger zurückverfolgt, das heißt, bis zu dem ersten Knoten l mit $low(l) \leq lim(w), lim(x) \leq lim(l)$. Natürlich müssen die Postorder-Parameter beim Austausch der Kanten auch korrigiert werden, aber nur für Knoten unterhalb von l.

Die Geschwindigkeit des Network-Simplex-Algorithmus wird außerdem stark durch die Wahl der auszutauschenden Kante mit negativem Schnittwert beeinflußt. Die Praxis zeigt, daß es viele Iterationen sparen kann, wenn man die Baumkanten zyklisch durchsucht, statt immer wieder am Anfang der Kantenliste zu beginnen.

4.3 Die Ordnung der Knoten in den Schichten

Nach der Rang-Einteilung werden Kanten zwischen Knoten, deren Rang sich um mehr als eins unterscheidet, durch Ketten von Kanten der Länge 1 ersetzt. Dazu werden Hilfsknoten, sogenannte virtuelle Knoten, auf den Schichten, die zwischen den beiden Knoten der ursprünglichen Kante liegen, eingeführt. Dadurch gibt es in dem entstehendem Graph nur noch Kanten zwischen benachbarten Schichten.

Die Anordnung der Knoten in den Schichten bestimmt die Anzahl der Kantenkreuzungen in der Darstellung des Graphen. Eine gute Anordnung der Knoten ist also eine, die wenige Kreuzungen produziert. Hierzu müssen wir auf Heuristiken zurückgreifen, da das Problem der Minimierung von Kantenkreuzungen selbst für Graphen mit nur zwei Schichten NP-vollständig ist.

Einige wichtige Heuristiken zur Reduzierung von Kantenkreuzungen in Graphen, die in Schichten gezeichnet werden, sind auf der Grundlage des folgendem Schemas, das zuerst von Warfield [6] vorgestellt wurde, entwickelt worden. Zuerst wird für jede Schicht eine initiale Ordnung hergestellt. Dann wird in einer Folge von Iterationen versucht, die Ordnung zu verbessern. Jede Iteration besucht zunächst die erste Schicht und geht dann der Reihe nach alle Schichten bis zur letzten durch, oder umgekehrt. Wird eine Schicht besucht, so wird jeder ihrer Knoten ein Gewicht zugeordnet, das von den relativen Positionen der mit ihm durch eine Kante verbundenen Knoten der vorhergehenden Schicht abhängt (das heißt entweder eine Schicht höher oder eine Schicht tiefer, je nachdem in

```
1.  procedure ordering()
2.    order = init_order();
3.    best = order;
4.    for i = 0 to Max_iterations do
5.      wmedian(order,i);
6.      transpose(order);
7.      if crossing(order) < crossing(best) then
8.        best = order;
9.    end
10.   return best;
11. end
```

Abbildung 4.7: Ordnungs-Algorithmus

welche Richtung die aktuelle Iteration läuft). Dann werden die Knoten neu geordnet, indem sie nach ihren Gewichten sortiert werden.

Zwei bekannte Methoden, die Knoten zu gewichten, sind die Barycenter-Methode [2] und die Median-Methode [7]. Sei v ein Knoten und P die Liste der Positionen der zu ihm inzidenten Knoten in der gerade betrachteten benachbarten Schicht. Die Position eines Knotens ist seine Nummer in der Ordnung der Schicht, in der er sich befindet. Die Barycenter-Methode definiert das Gewicht von v als das arithmetische Mittel der Elemente aus P, die Median-Methode dagegen als das mittlere der Elemente aus P. Wenn die Anzahl der Elemente in P gerade ist, gibt es zwei Medians. Das führt zu zwei verschiedenen Median-Methoden: benutze stets den linken Median oder benutze stets den rechten Median. Die Median-Methode erweist sich in der Praxis als besser geeignet und hat einen leichten theorethischen Vorteil. Eades und Wormald [7] haben nämlich gezeigt, daß die Anzahl der Kantenkreuzungen im Bild eines Graphen mit zwei Schichten, das nach der Median-Methode gezeichnet wurde (das heißt, bei dem die Knoten nach der Median-Methode geordnet wurden), höchstens drei mal so groß ist wie die minimale Anzahl der Kantenkreuzungen. Solch eine Schranke ist für die Barycenter-Methode bisher nicht bekannt.

Die hier verwendete Heuristik zum Ordnen der Knoten ist eine Verfeinerung der Median-Methode bei der Gansner et al. zwei Neuerungen eingeführt haben. Die erste ist, daß in dem Fall, daß es zwei Medians gibt, ein Wert zwischen diesen Beiden genommen wird, der mehr in die Richtung geht, auf der die Knoten dichter zusammenliegen (wie genau, das sehen wir später in Abb. 4.8). Die zweite Neuerung ist eine zusätzliche Heuristik, die die Anzahl der Kreuzungen verringert, nachdem die Knoten nach ihren Gewichten sortiert wurden und die eine bezüglich Vertauschung zweier benachbarter Knoten lokal optimale Ordnung liefert. Diese zusätzliche Heuristik reduziert in den meisten Fällen die Anzahl der Kreuzungen noch einmal um weitere 20-50%. Detaillierte Statistiken zu diesem Thema findet man in der entsprechenden Literatur [8].

```
1.  procedure wmedian(order,iter)
2.    if iter mod 2 = 0 then
3.      for r = 1 to Max_rank do
4.        for v in order[r] do
5.          median[v] = median_value(v,r-1);
6.        sort(order[r],median);
7.      end
8.    else . . .
9.    endif
10. end
```

Abbildung 4.8: Die gewichtete Median-Heuristik

Abb. 4.7 zeigt den Ordnungs-Algorithmus:

2: `init_order` liefert eine initiale Ordnung. Dazu wird eine Tiefensuche auf dem Graphen durchgeführt, wobei man stets bei denjenigen Knoten beginnt, die keine Vorgänger haben. Den Knoten wird eine Positon in ihrer Schicht in der Reihenfolge von links nach rechts zugewiesen, das heißt, wird ein Knoten besucht, so bekommt er die nächste freie Position in seiner Schicht. Diese Strategie sorgt dafür, daß die initiale Ordnung eines Baumes keine Kreuzungen hat. Dadurch werden schon einige leicht vermeidbare Kantenkreuzungen unterbunden.

4-9: `Max_Iterations` ist die Anzahl der Iterationen, die durchgeführt werden (bei Gansner et al. ist `Max_Iterations` 24). Bei jeder Iteration wird die neue Ordnung übernommen, wenn sich die Anzahl der Kreuzungen verringert hat. Es wäre auch denkbar eine andere Strategie zu verwenden, die keine absolute Zahl von Iterationen vorgibt. Zum Beispiel könnte man die Iterationen so lange durchführen, wie die Lösung sich in den letzten x Iterationen noch um y Prozent verbessert hat, wobei man dann x und y nach Belieben setzen kann. `wmedian` ordnet die Knoten neu gemäß der gewichteten Median-Methode. `transpose` vertauscht die benachbarten Knoten in jeder Schicht, wenn dadurch die Zahl der Kantenkreuzungen reduziert werden kann. Diese beiden Funktionen werden gleich noch ausführlich beschrieben.

Der gewichtete Median-Algorithmus ist in Abb. 4.8 dargestellt. Je nachdem, ob die aktuelle Iteration eine gerade oder eine ungerade Nummer hat, werden die Schichten von oben nach unten oder umgekehrt durchgegangen. Um die Darstellung zu vereinfachen, wird in Abb. 4.8 nur eine Richtung im Detail dargestellt:

1-10: Beim Durchlaufen in Vorwärtsrichtung beginnt die Hauptschleife mit 1 und endet mit der Nummer der höchsten Schicht. In jeder Schicht wird den Knoten ein Median abhängig von den zu ihm inzidenten Knoten in der nächst niedrigeren Schicht zugewiesen. Dann werden die Knoten in

```
1.  procedure median_value(v,adj_rank)
2.    P = adj_position(v,adj_rank);
3.    m = |P|/2;
4.    if |P| = 0 then
5.      return -1.0;
6.    elseif |P| mod 2 = 1 then
7.      return P[m];
8.    elseif |P| = 2 then
9.      return (P[0] + P[1])/2;
10.   else
11.     left = P[m-1] - P[0];
12.     right = P[|P|-1] - P[m];
13.     return (P[m-1]*right+P[m]*left)/(left+right);
14.   endif
15. end
```

Abbildung 4.9: Die Funktion median_value

der Schicht nach ihren Medians sortiert. Eine wichtige Frage ist noch, was man mit den Knoten macht, die keine inzidenten Knoten in der vorigen Schicht haben. Bei Gansner et al. behalten solche Knoten ihre bisherige Position bei und die anderen Knoten werden dann in die verbleibenden Positionen sortiert.

In Abb. 4.9 wird die Funktion `median_value` beschrieben:

1-15: Der Median eines Knotens ist definiert als die Position des mittleren seiner benachbarten Knoten, wenn diese eindeutig ist. Sonst wird ein Wert zwischen den beiden Medians genommen, der sich danach richtet, wie eng die Knoten beieinander liegen. Generell wird der Wert zu der Seite verschoben, auf der die Knoten dichter zusammen liegen.

2: Die Funktion `adj_position` liefert ein geordnetes Feld mit den Positionen der zu `v` benachbarten Knoten in der gegebenen Schicht `adj_rank`.

4-5: Wenn Knoten keine Nachbarn in dieser Schicht haben, bekommen sie den Median-Wert -1. Dies wird in der `sort`-Funktion benutzt, um anzuzeigen, daß diese Knoten ihre bisherige Position behalten sollen.

Abb. 4.10 zeigt die Transpositionsheuristik:

3-15: Dies ist die Hauptschleife, die so lange läuft, wie die Anzahl der Kantenkreuzungen durch das Vertauschen benachbarter Knoten verringert werden kann. Wie in der Funktion `ordering` könnte man auch hier eine andere Abbruchbedingung wählen, zum Beispiel um die Schleife zu beenden, wenn die Verringerung nur noch ein genügend kleiner Teil der Anzahl der Kreuzungen ist.

```
1.  procedure transpose(rank)
2.    improved = True;
3.    while improved do
4.      improved = False;
5.      for r = 0 to Max_rank do
6.        for i = 0 to |rank[r]|-2 do
7.          v = rank[r][i];
8.          w = rank[r][i+1];
9.          if crossing(v,w) > crossing(w,v) then
10.           improved = True;
11.           exchange(rank[r][i],rank[r][i+1]);
12.         endif
13.       end
14.     end
15.   end
16. end
```

Abbildung 4.10: Transpositionsheuristik zur Reduzierung der Kreuzungen

7-12: Jedes Paar benachbarter Knoten wird untersucht. Ihre Positionen werden getauscht, wenn dies die Anzahl der Kreuzungen verringert. Die Funktion `crossing (v,w)` zählt einfach die Anzahl der Kreuzungen, die entstehen, wenn `v` links von `w` plaziert wird, genauer, die Anzahl der Kreuzungen, die durch Kanten entstehen, die `v` oder `w` als Endpunkte haben. Auf Kreuzungen, die durch andere Kanten entstehen, hätte das Vertauschen von `v` und `w` natürlich keinen Einfluß.

Wenn die Knoten nach ihren Median-Werten sortiert werden, oder wenn man den Transpositions-Algorithmus anwendet, kann es vorkommen, daß beide Knoten die gleichen Median-Werte haben, bzw. daß die Anzahl der Kreuzungen die gleiche ist, egal in welcher Reihenfolge die Knoten auftreten. Es hat sich als hilfreich herausgestellt, und außerdem als passend zu A4 aus 4.1.1, wenn man Knoten mit gleichen Werten in der Sortier- oder Transpositionsphase bei jedem zweiten Durchlauf vertauscht.

4.4 X-Koordinaten

4.4.1 Was soll diese Phase des Algorithmus leisten?

In dieser Phase werden den Knoten die X-Koordinaten zugewiesen (die Y-Koordinaten sind durch die Einteilung in Schichten ja bereits gegeben). Einige ältere Arbeiten behandeln dies als eine der Barycenter-Methode nachgeschaltete Phase, die nur lokale Verbesserungen vornimmt, um allzu schlechte Bilder zu vermeiden. Gansner et al. sehen die Zuweisung der X-Koordinaten als ein eigenständiges, wohldefiniertes Problem an, was zu besseren Bildern führt und

die Möglichkeit zur späteren Verbesserung oder Erweiterung des Algorithmus offenhält. Entsprechend der ästhetischen Kriterien aus 4.1.1, sind kurze, gerade Kanten in einem Bild eher erwünscht als lange, geknickte. Dies führt uns dazu, die X-Koordinatenzuweisung als das folgende ganzzahlige Optimierungs-Problem zu betrachten:

$$\min \sum_{e=(v,w)} \Omega(e)\omega(e)|x_w - x_v|$$

$$\text{wobei } x_b - x_a \geq 1$$

für alle a, b mit a ist linker Nachbar von b in derselben Schicht.

Das Gewicht $\Omega(e)$, ein interner Wert und nicht zu verwechseln mit dem ursprünglichen Kantengewicht $\omega(e)$, wird so definiert, daß die Kosten gering sind, wenn lange Kanten gerade gezeichnet werden. Da Kanten zwischen zwei benachbarten Schichten stets als gerade Linien gezeichnet werden können, ist es wichtiger, den horizontalen Abstand zwischen virtuellen Knoten möglichst gering zu halten. Dann können die Ketten von virtuellen Knoten, durch die die ursprünglichen Kanten ersetzt wurden, nämlich vertikal angeordnet werden, und der entstehende Polygonzug, durch den die Kante dargestellt wird, hat dann keine Knicke. Dies ist sehr wichtig, da das Bild eines Graphen, dessen Kanten viele Knicke aufweisen, sehr unübersichtlich wird. Dementsprechend werden die Kanten in drei verschiedene Klassen, abhängig von der Art ihrer Endknoten, aufgeteilt.

1. Beide Knoten sind echte Knoten (das heißt keine virtuellen Knoten).
2. Ein Knoten ist ein echter und einer ein virtueller Knoten.
3. Beides sind virtuelle Knoten.

Sei e eine Kante der Klasse 1, f eine Kante der Klasse 2 und g eine Kante der Klasse 3. Dann ist $\Omega(e) \leq \Omega(f) \leq \Omega(g)$. Gansner et al. benutzen hier entsprechend die Werte 1, 2, und 8.

4.4.2 Optimale X-Koordinaten-Zuweisung

Um den Knoten die X-Koordinaten zuzuweisen, benutzen wir noch einmal den Network-Simplex-Algorithmus aus 4.2.2, indem wir die X-Koordinaten als den Rang der Knoten ansehen. Dafür müssen wir uns einen Hilfsgraph konstruieren, wie in Abb. 4.11 dargestellt. Die Knoten des Hilfsgraphen G' sind die Knoten des Ausgangsgraphen G plus eines Knotens n_e für jede Kante e aus G (mit Ausgangsgraph ist hier der Graph nach dem Ordnungs-Algorithmus gemeint, das heißt, in G sind bereits die virtuellen Knoten enthalten). In G' gibt es nun zwei Arten von Kanten. Die einen enthalten die Information über die Kosten der ursprünglichen Kanten. Jede Kante $e = (u, v)$ in G wird ersetzt durch zwei Kanten $e_u = (n_e, u)$ und $e_v = (n_e, v)$ mit $\delta = 0$ und $\omega = \omega(e)\Omega(e)$. Die anderen Kanten trennen Knoten, die in derselben Schicht liegen. Sei v der linke Nachbar von w. Dann hat G' eine Kante $e_{(v,w)} = (v, w)$ mit $\delta(e_{(v,w)}) = 1$ und $\omega(e_{(v,w)}) = 0$. Diese Kante sorgt dafür, daß die Knoten u und v nicht dieselben Koordinaten

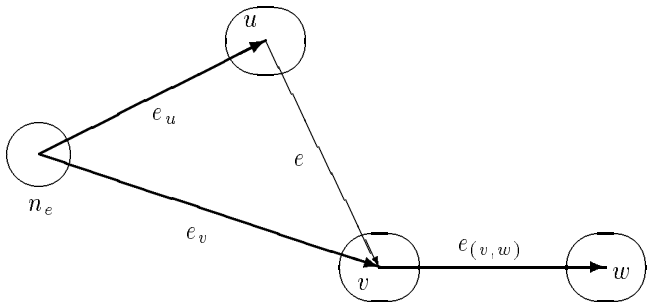


Abbildung 4.11: Ein Hilfsgraph

erhalten, hat aber keinen Einfluß auf die Kosten. Wir betrachten nun das Rang-Einteilungsproblem für den Graphen G', das mithilfe der Network-Simplex-Methode gelöst werden kann. Jede X-Koordinaten-Zuweisung von G entspricht einer Rang-Einteilung von G' mit den gleichen Kosten. Diese erhält man, indem man jedem Knoten u aus G als Rang in G' den Wert zuweist, den er in G als X-Koordinate hat, und jedem Knoten n_e mit $e = (u, v)$ den Wert $min(x_u, x_v)$, wobei x_u und x_v die X-Koordinaten von u bzw. v in G sind. Umgekehrt induziert jede Rang-Einteilung in G' eine X-Koordinaten-Zuweisung in G. Zusätzlich gilt, daß in einer optimalen Rang-Einteilung stets eine der Kanten e_u oder e_v die Länge 0 haben muß und die andere die Länge $|x_u - x_v|$. Das heißt, die Kosten der ursprünglichen Kante (u, v) in G ist gleich der Summe der Kosten der beiden Kanten e_u und e_v in G' und somit haben insgesamt beide Lösungen, die der Rang-Einteilung von G' und die der X-Koordinaten-Zuweisung von G, dieselben Kosten. Also bedeutet Optimalität für G' auch Optimalität für G, und eine Lösung für G' liefert uns eine Lösung für G.

4.4.3 Details der Implementierung

Der Hilfsgraph ist erheblich größer als der ursprüngliche Graph. Hat der ursprüngliche Graph V Knoten, E Kanten und S Schichten, so hat der Graph mit den zusätzlich eingeführten virtuellen Knoten $V + D$ Knoten und $E + D$ Kanten, wobei D die Anzahl der virtuellen Knoten sei. Der Hilfsgraph hat dann $V+E+2D$ Knoten und $V+2E+3D-S$ Kanten. Dieser Graph braucht deutlich mehr Zeit bei der Anwendung des Network-Simplex-Algorithmus. Demzufolge sind die Verbesserungen, die in 4.2.3 angeführt wurden, in dieser Phase des Algorithmus äußerst wichtig.

Es ist außerdem noch eine weitere Verbesserung möglich. Wenn man sich die spezielle Struktur des Hilfsgraphen anschaut, stellt man fest, daß es sehr leicht ist, hierfür einen zulässigen spannenden Baum zu konstruieren, und zwar auf folgende Weise. Man benutzt alle Kanten, die Knoten in ein und derselben Schicht verbinden. Für je zwei benachbarte Schichten wählt man eine Kante $f = (u, v)$ zwischen den beiden Schichten und fügt dann beide Kanten f_u und f_v zum Baum hinzu. Dadurch wird für alle Knoten in diesen beiden Schichten die Positon relativ zueinander festgelegt. Zum Schluß fügt man für jede Kante

$e = (w, x)$ die Kante e_w oder e_x zum Baum hinzu, je nachdem ob w oder x weiter links plaziert ist (was durch die Wahl der Kante f festgelegt wurde). Ohne die genannten Verbesserungen braucht der Network-Simplex-Algorithmus 5–10 mal länger. So läuft er jedoch genauso schnell wie einige Heuristiken, die erstens schwerer zu programmieren sind und zweitens keine optimalen Lösungen produzieren.

4.5 Erweiterung auf beliebige Graphen

Der Algorithmus, wie bisher beschrieben, ist nur auf kreisfreie Graphen anwendbar. Wir wollen ihn jedoch auch auf nicht-kreisfreie Graphen anwenden. Dazu sind folgende Modifikationen notwendig. Vor Anwendung des bisherigen Algorithmus wird der Graph kreisfrei gemacht. Dazu werden zunächst alle Schleifen, das heißt Kanten (v, v) mit $v \in V$, gelöscht. Zusätzlich führen wir eine Tiefensuche auf dem Graphen durch, bei der den Knoten ihre DFS-Parameter (DFS = depth first search) zugewiesen werden, nämlich der Wert $d[v]$, wenn der Knoten besucht wird und der Wert $f[v]$, wenn der Knoten verlassen wird, also, wenn alle seine Nachfolger besucht wurden. Dies funktioniert so, daß bei der Tiefensuche ein Zähler mitgeführt wird, dessen Wert jeweils der nächste DFS-Parameter wird (d oder f). Nach jeder solchen Zuweisung wird der Zähler um 1 erhöht, das heißt, jeder Wert zwischen 1 und $2|V|$ kommt nur einmal unter den d- oder f-Werten vor. Nun gibt es unter den Kanten, die nicht zum DFS-Baum gehören, drei verschiedene Sorten, die sich durch die d- und f-Werte ihrer Anfangs- und Endpunkte unterscheiden lassen (u bezeichne im Folgenden stets den Anfangs- und v den Endpunkt einer Kante).

1. Die Vorwärtskanten, das heißt Kanten, die von einem Knoten zu einem seiner Nachfahren im DFS-Baum gehen. Für diese gilt $d[u] < d[v]$ und $f[u] > f[v]$.

2. Die Rückwärtskanten, das heißt Kanten, die von einem Knoten zu einem seiner Vorfahren im DFS-Baum gehen. Für diese gilt $d[u] > d[v]$ und $f[u] < f[v]$.

3. Die Querkanten, das heißt Kanten, die weder Vorwärtskanten noch Rückwärtskanten sind. Für sie gilt $d[u], f[u] < d[v], f[v]$ oder $d[u], f[u] > d[v], f[v]$.

Offenbar gilt: Gibt es keine Rückwärtskanten, so ist der Graph kreisfrei. Also werden alle Rückwärtskanten vorübergehend umgedreht, und man erhält einen kreisfreien Graph. Auf diesem Graph läßt man nun den Algorithmus laufen, führt die zuvor gelöschten Schleifen wieder ein und zeichnet dann die gedrehten Kanten wieder in ihrer ursprünglichen Orientierung.

4.6 Beispiele

Die hier abgebildeten Darstellungen von Graphen sind direkt vom RELVIEW-System gezeichnet worden. Anhand dieser Bespiele werden wir sehen, daß man

im allgemeinen über diesen Algorithmus zwei Aussagen treffen kann. Zum Einen ist er sehr gut geeignet, Graphen gut lesbar darzustellen, auch wenn sie relativ komplex sind. Zum Anderen können aber in Einzelfällen recht einfache Graphen auch wesentlich knapper dargestellt werden, als dieser Algorithmus es tut, ohne daß dabei die Lesbarkeit beeinträchtigt wird.

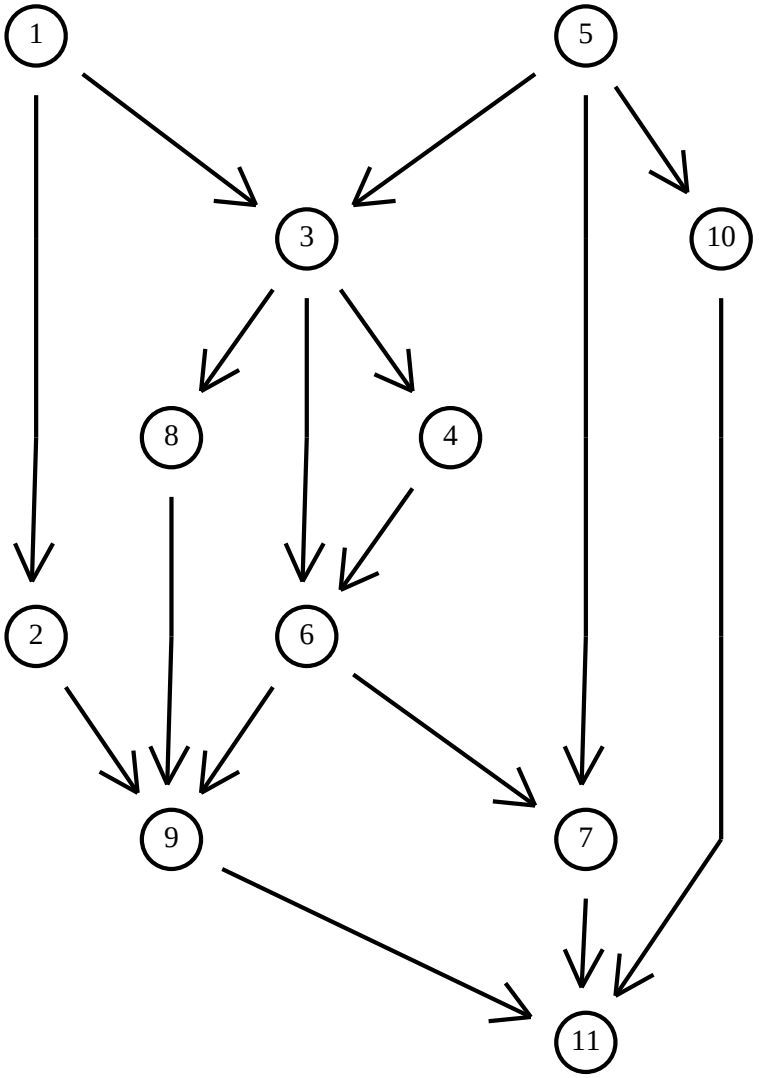


Abbildung 4.12: Beispiel 1

Der in Abb. 4.12 dargestellte Graph ist sehr übersichtlich und gut lesbar gezeichnet. Für diesen Fall ist der Algorithmus also gut geeignet.

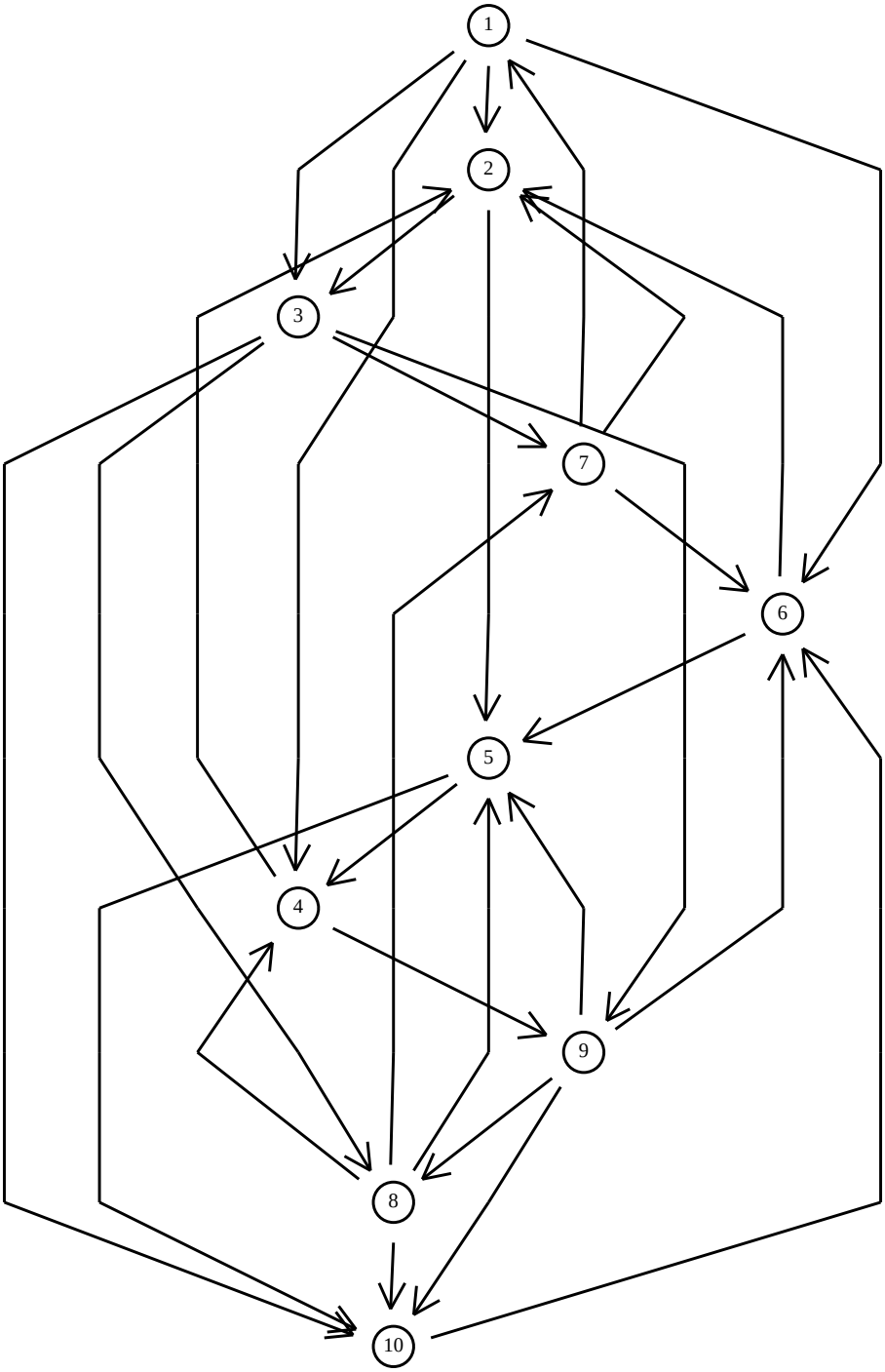


Abbildung 4.13: Beispiel 2

In Abb. 4.13 sehen wir einen Graphen, der etwas komplexer ist. Aber auch dieses Bild ist ziemlich übersichtlich (vergleiche hierzu Abb. 6.5 und Abb. 7.5)

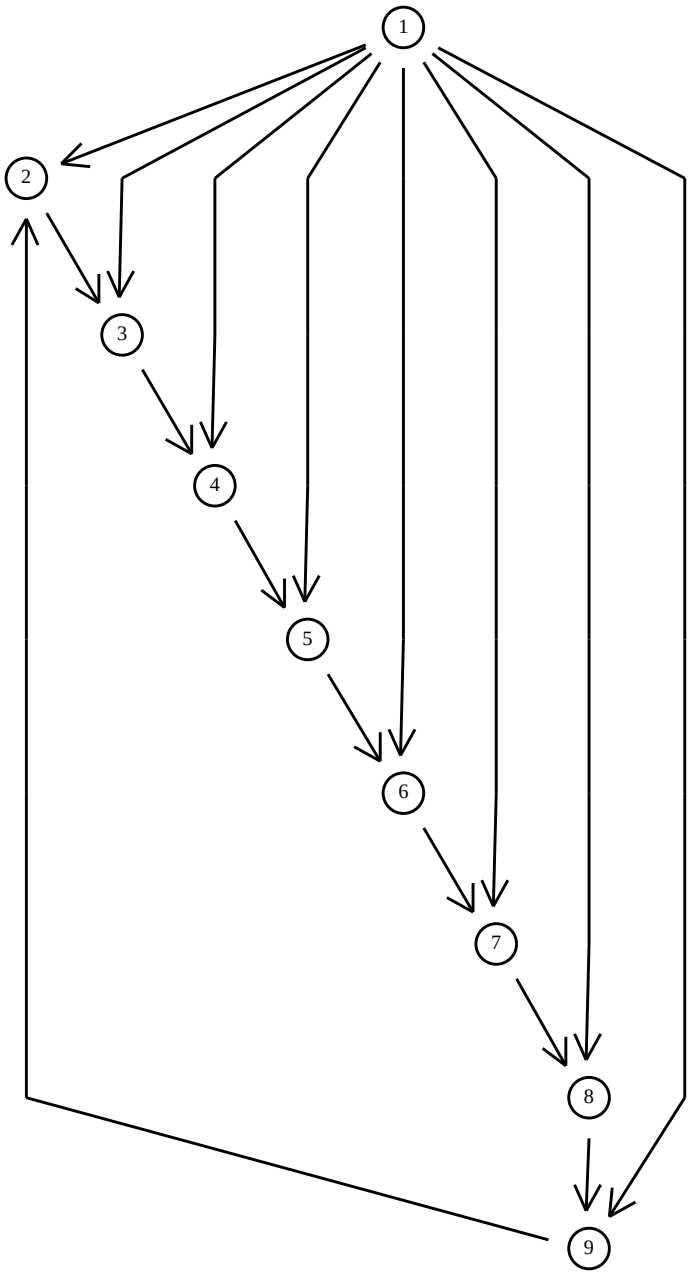


Abbildung 4.14: Beispiel 3

Der Graph aus Abb. 4.14 könnte allerdings besser gezeichnet werden. Es wäre hier besser, den Graph nicht in Schichten zu zeichnen, da hierdurch die Kanten, die von Knoten 1 ausgehen, unötig lang werden. Eine bessere Möglichkeit wäre zum Beispiel, die Knoten 2–9 auf einem Kreis um Knoten 1 zu plazieren (vgl. hierzu Abb. 7.2).

Kapitel 5

Zeichnen von Bäumen

5.1 Einführung

Der in diesem Kapitel beschriebene Algorithmus stammt aus einer Arbeit von Wetherell und Shannon [11]. Die Darstellung des Algorithmus wird so aussehen, daß zunächst ein naiver Algorithmus präsentiert wird, der noch nicht besonders schöne Bilder liefert, und ausgehend von diesem ein Algorithmus zum schönen Zeichnen von Bäumen entwickelt wird. Die Kriterien, die hier angewandt werden, um zu beurteilen, wann ein Baum schön ist, werden in der jeweiligen Phase, das heißt, wenn der Algorithmus dieses Kriterium erfüllt, genauer erwähnt. Es seien hier nur einige Punkte erwähnt, die einen schönen Baum ausmachen und wohl auf allgemeine Zustimmung treffen.

- Bäume sind planare Graphen, also sollten Kanten sich nicht schneiden.
- Im Baum hat jeder Knoten eine bestimmte Entfernung von der Wurzel. Diese sollte auch im Bild zum Ausdruck kommen, das heißt, kein Knoten soll dichter an der Wurzel liegen als einer seiner Nachfolger.
- In einem Binärbaum soll der linke Sohn links und der rechte Sohn rechts vom Vater liegen.

Wie groß in der Darstellung eines Baumes die Knoten sein sollen, wird hier nicht betrachtet. Um die Knoten zu trennen, wird bei der Vergabe von Positionen auf einer Schicht zwischen zwei Knoten mindestens ein Platz frei gelassen. Die Ausgabe des Algorithmus ist dann eine Angabe der relativen Positionen der Knoten zueinander, deren absolute Werte nach Belieben variiert werden können.

5.2 Ein naiver Baumzeichenalgorithmus

Für diesen ersten Algorithmus haben wir zwei ästhetische Kriterien, die das Bild erfüllen soll.

A1: Die Knoten des Baumes, die dieselbe Entfernung von der Wurzel haben, sollen alle auf einer geraden Linie liegen, und alle solchen Linien sollen parallel verlaufen.

```
1.  procedure naive_tree()
2.  begin
3.    for i = 0 to max_height do
4.      next_x[i] = 1;
5.    root.status = 0;
6.    current = root;
7.    while current ≠ nil do
8.      if current.status = 0 then
9.        current.x = next_x[current.height];
10.       next_x[current.height] = next_x[current.height] + 2;
11.       for v ist Sohn von current do
12.         v.status = 0;
13.       current.status = 1;
14.     elseif 1 ≤ current.status ≤ current.♯_of_sons then
15.       current.status = current.status + 1;
16.       current = current.son[current.status−1];
17.     else (* current.status > current.♯_of_sons *)
18.       current = Vater von current;
19. end
```

Abbildung 5.1: Naiver Baumzeichenalgorithmus

A2: Das Bild eines Baumes soll so wenig Platz wie möglich verbrauchen, das heißt, die Breite soll möglichst gering sein (Die Höhe des Baumes ist ja durch den Baum selber festgelegt).

Die Höhe eines Knotens bestimmt seine Y-Koordinate. Die Höhe des Baumes wird gebraucht, um Speicherplatz für Hilfsfelder zu reservieren. Also ist das erste, was man tun muß, die Bestimmung der Höhe der Knoten, was durch irgendeine Durchlaufstrategie, bei der Väter vor ihren Söhnen besucht werden, erledigt werden kann. Die einfachste Art, die Kriterien A1 und A2 zu erfüllen, ist, die Knoten in jeder Schicht so weit wie möglich links zu plazieren. In Abb.

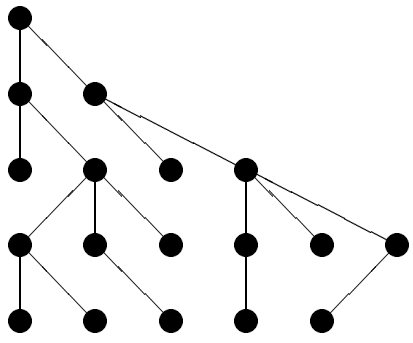


Abbildung 5.2: Baum gemalt mit naivem Algorithmus

```
1.  procedure binary_tree()
2.  begin
3.    next_number = 1;
4.    root.status = first_visit;
5.    current = root;
6.    while current ≠ nil do
7.      case current.status of
8.        first_visit: begin
9.          current.status = left_visit;
10.         if current hat linken Sohn then
11.           current = linker Sohn von current;
12.           current.status = first_visit;
13.         endif
14.       end
15.       left_visit: begin
16.         current.x = next_number;
17.         next_number = next_number +1;
18.         current.status = right_visit;
19.         if current hat rechten Sohn then
20.           current = rechter Sohn von current;
21.           current.status = first_visit;
22.         endif
23.       end
24.       right_visit: current = Vater von current;
25.     endcase
26. end
```

Abbildung 5.3: Knuths Algorithmus

5.2 ist ein Baum zu sehen, der mit diesem Algorithmus gezeichnet wurde.
Abb.5.1 zeigt diesen ersten Algorithmus:

3–5: Das Feld `next_x` wird initialisert, `root.status` wird auf Null und `current` gleich `root` gesetzt. `current` ist der Knoten, der gerade besucht wird, und `next_x[i]` gibt die nächste freie Position für Knoten der Höhe `i` an. Der Status eines Knotens sagt aus, ob er selbst und wieviele seiner Söhne schon besucht wurden.

9–13: Dem aktuellen Knoten wird die nächste freie Position auf seiner Höhe zugewiesen, und der entsprechende Eintrag im Feld `next_x` wird um 2 erhöht (um 2, damit zwischen zwei Knoten immer ein Platz frei bleibt). Der Status der Söhne von `current` wird auf 0 gesetzt und der Status von `current` selbst auf 1, was bedeutet, daß er schon besucht wurde.

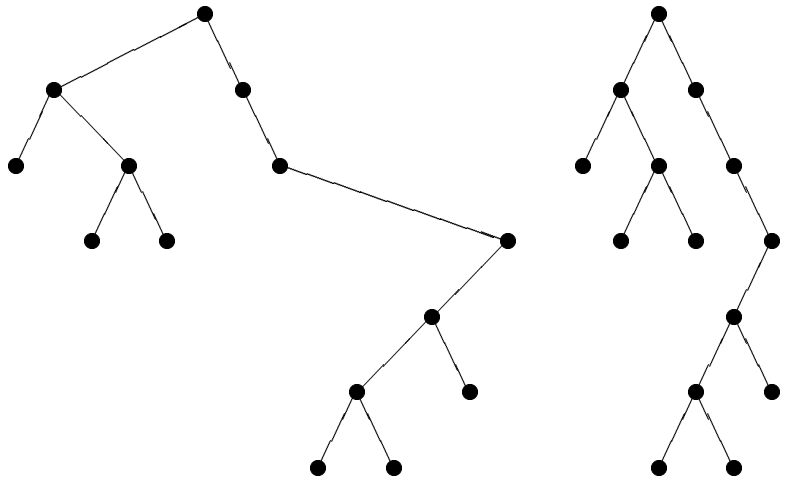


Abbildung 5.4: Zwei Bilder von einem Baum und zwar links von Knuths Algorithmus und rechts eine bessere Version

15–16: `current.status` wird um 1 erhöht, und im folgenden Durchlauf durch die while–Schleife wird der nächste Sohn von `current` besucht.

18: Alle Söhne wurden besucht, und `current` wird gleich Vater von `current` gesetzt (wobei der Vater von `root` als `nil` definiert ist).

5.3 Knuths Algorithmus

Wie in Abb. 5.2 zu sehen, ist es bei diesem Algorithmus möglich, daß Väter links, rechts oder zentriert über ihren Söhnen gezeichnet werden. Um einen beseren Algorithmus zu finden, betrachten wir zunächst nur Binärbäume. Für diese wollen wir das folgende Kriterium erfüllt haben:

A3: In einem Binärbaum sollen linke Söhne stets links und rechte Söhne stets rechts von ihren Vätern liegen.

Der Algorithmus von Knuth [9] in Abb. 5.3 stellt Binärbäume so dar, daß sie A1 und A3 erfüllen:

9–12: Der Knoten `current` wird hier zum ersten Mal besucht. Falls vorhanden, wird der linke Sohn besucht, und der Status wird entsprechend gesetzt.

16–21: Der linke Sohn wurde bereits besucht, und `current` erhält nun die nächste freie Nummer als X-Koordinate. Danach wird `next_number` um eins hochgezählt und der rechte Sohn besucht, falls vorhanden.

24: Beide Söhne wurden besucht, und es geht beim Vater von `current` weiter.

Den Knoten wird dort als X-Koordinate also ihr Index in einer In-Order-Numerierung zugewiesen. Dadurch sind zwar linke Söhne immer links und rechte Söhne immer rechts von ihren Vätern plaziert, aber dadurch, daß in jeder Spalte nur ein Knoten liegen kann, können Bilder entstehen, die sehr viel mehr Platz als nötig verbrauchen (siehe Abb. 5.4), das heißt, A2 wird durch Knuths Algorithmus verletzt.

```
1. procedure tidy_tree()
2. begin
3.    for i = 0 to max_height do
4.       modifier[i] = 0;
5.       next_pos[i] = 1;
6.    endfor;
7.    first_pass();
8.    modifying_pass();
9. end
```

Abbildung 5.5: Algorithmus von Wetherell und Shannon

5.4 Der Algorithmus von Wetherell und Shannon

Die beiden bisher vorgestellten Algorithmen haben den Nachteil, daß sie jeweils nur eines der beiden Kriterien A2 und A3 beachten und das andere vollkommen vernachlässigen. Der naive Algorithmus plaziert die Knoten, ohne dabei die Positionen der Söhne zu beachten, und Knuths Algorithmus benutzt für jeden Knoten eine eigene Spalte, so daß in dieser Spalte keine anderen Knoten mehr plaziert werden können, wodurch es zu einem zu großen Platzverbrauch kommt. Der folgende Algorithmus verbindet nun die Ideen der beiden Vorherigen. Abb. 5.5 zeigt den groben Aufbau des Algorithmus von Wetherell und Shannon. Es finden hier zwei Durchläufe statt. Im Ersten wird den Knoten in einer Post-Order Durchlaufstrategie das Maximum der Position `place` und der nächsten freien Position in der entsprechenden Schicht zugewiesen. Dabei ist `place` die Position, die der Knoten haben sollte, um zwischen seinen Söhnen zu liegen. Liegt `place` zu weit links, das heißt, liegen die Söhne so, daß sie den Vater zu weit nach links ziehen, so müssen sie (und damit natürlich auch alle ihre Nachfolger) entsprechend nach rechts verschoben werden. Dazu wird in diesem Durchlauf zusätzlich ein Feld `modifier` angelegt, in dem für jeden Knoten gespeichert wird, wie weit seine Söhne verschoben werden müssen. Dabei muß man dafür sorgen, daß der Modifier in einer Schicht von rechts nach links nie kleiner werden darf. Deshalb merkt man sich für jede Schicht auch den gerade aktuellen Modifier. Im zweiten Durchlauf werden (in einer Pre-Order Durchlaufstrategie) die Knoten entsprechend der Summe der Modifier aller ihrer Vorfahren nach rechts verschoben.

Abb. 5.6 zeigt den ersten Durchlauf durch den Baum, bei dem den Knoten vorläufige Positionen zugewiesen werden:

7–12: Den Söhnen wird vor dem Vater eine Position zugewiesen, damit der Vater später zwischen seine Söhne plaziert werden kann.

15: Die Variable `place` ist abhängig von der Position der Söhne. Ist `current` ein Blatt, so ist `place` die nächste freie Position in Schicht current.height. Hat `current` zwei Söhne, so ist `place` der Durchschnitt ihrer beiden Positionen. Hat `current` nur einen Sohn, so ist `place` um eins größer bzw.

```
1.  procedure first_pass()
2.  begin
3.    current = root;
4.    current.status = first_visit;
5.    while current ≠ nil do
6.      case current.status of
7.        first_visit:
8.          Setze current.status = left_visit und
9.          besuche linken Sohn, falls vorhanden;
10.       left_visit:
11.         Setze current.status = right_visit und
12.         besuche rechten Sohn, falls vorhanden;
13.       right_visit:
14.         h = current.height;
15.         Finde place;
16.         modifier[h] = max(modifier[h],next_pos[h]-place);
17.         if current ist Blatt then
18.           current.x = place;
19.         else
20.           currrent.x = place + modifier[h];
21.         next_pos[h] = current.x + 2;
22.         current.modifier = modifier[h];
23.         current = Vater von current;
24.       end;
25.     endcase;
26. end
```

Abbildung 5.6: Erster Durchlauf

kleiner als die Position seines Sohnes, je nachdem, ob es ein linker oder ein rechter Sohn ist.

16: Da der Modifier innerhalb einer Schicht nicht kleiner werden darf, wird `modifier[h]` als das Maximum vom alten Modifier und dem Abstand von `place` zu `next_pos[h]` gesetzt.

18: Ist `current` ein Blatt, so bekommt er einfach die nächste freie Position in der Schicht (falls `current` Blatt ist, wird `place` nämlich gerade so gewählt).

20: `current` wird soweit nach rechts verschoben, daß seine Nachfolger nicht mit Nachfolgern von Knoten, die links von ihm liegen, kollidieren.

21–23: `next_pos[h]` wird aktualisiert, `current.modifier` wird der aktuelle Modifier der Schicht zugewiesen und danach geht es beim Vater von `current`

```
1.  procedure modifying_pass()
2.  begin
3.    current = root;
4.    current.status = first_visit;
5.    modifier_sum = 0;
6.    while current ≠ nil do
7.      case current.status of
8.        first_visit:
9.          current.x = current.x + modifier_sum;
10.         modifier_sum = modifier_sum + current.modifier;
11.         Setze current.status = left_visit und besuche linken Sohn;
12.       left_visit:
13.         Setze current.status = right_visit und besuche rechten Sohn;
14.       right_visit:
15.         modifier_sum = modifier_sum - current.modifier;
16.         current = Vater von current;
17.     endcase;
18. end
```

Abbildung 5.7: Zweiter Durchlauf

weiter.

Der zweite Durchlauf ist in Abb. 5.7 dargestellt:

5: `modifier_sum` ist die Summe der Modifier aller Vorfahren des gerade besuchten Knotens, und wird zu Beginn auf Null gesetzt.

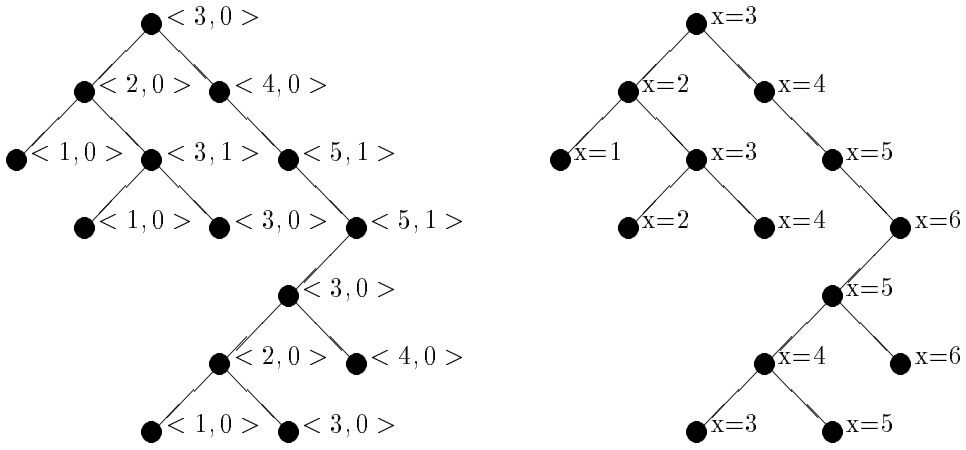
9–11: Die X-Position von `current` wird entsprechend der Summe der Modifier seiner Vorfahren heraufgesetzt, `modifier_sum` wird aktualisiert und der linke Sohn besucht.

15–16: Beide Söhne wurden besucht, und es geht beim Vater weiter. `modifier_sum` wird entsprechend aktualisiert.

In Abb. 5.8 wird anhand eines Beispielbaumes verdeutlicht, wie die einzelnen Phasen des Algorithmus funktionieren. Dieser Beispielbaum ist gemäß der Forderungen A1, A2 und A3 gut gezeichnet, es kann jedoch bei einigen Bäumen vorkommen, daß Algorithmus WS Bilder liefert, die mehr Platz brauchen als nötig wäre, um sie A1, A2 und A3 entsprechend zu zeichnen (siehe Abb. 5.9). Das heißt, Algorithmus WS verletzt Forderung A2. Dies ist darauf zurückzuführen, daß er das folgende Kriterium erfüllt, das eine Verschärfung von A3 ist.

A4: Ein Vater soll über seinen Söhnen zentriert sein.

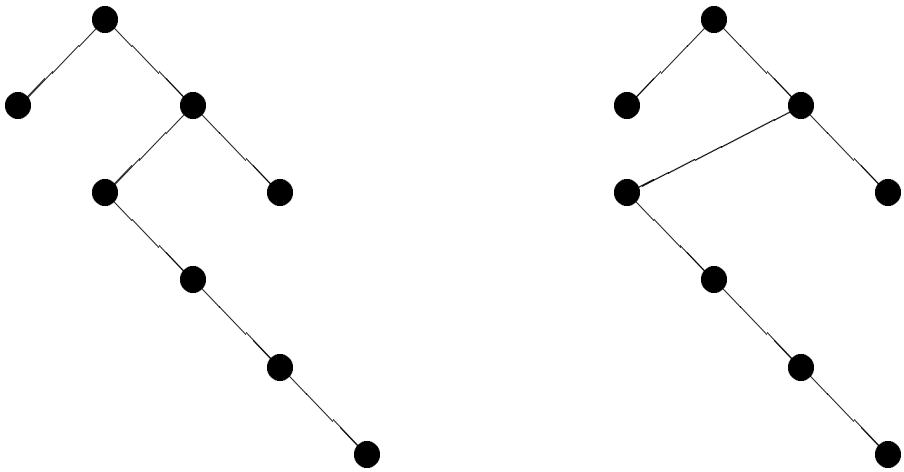
Fordert man Kriterium A4, so sind die Bilder von Algorithmus WS bezüglich der Breite minimal (beachte, daß die schmalere Version in Abb. 5.9 Kriterium A4



X-Koord. und Modifier nach 1. Durchlauf Endgültige X-Koordinaten

Abbildung 5.8: Beispiel für Algorithmus WS

verletzt). Wetherell und Shannon geben eine Möglichkeit an, den Algorithmus so zu modifizieren, daß er Bilder liefert, die Kriterium A2 erfüllen (Kriterium A4 dann natürlich nicht mehr). Diese Modifikation wollen wir hier aber nicht weiter verfolgen. Statt dessen betrachten wir die von ihnen nur skizzierte Möglichkeit, den Algorithmus auf beliebige Bäume (im Gegensatz zu den bisher betrachteten Binärbäumen) zu verallgemeinern. Dazu muß im ersten Durchlauf die **case**-Anweisung so verändert werden, daß nicht nur zwei Söhne eines Knotens besucht werden, bevor man zum Vater zurückkehrt, sondern alle vorhandenen. Dies erreicht man dadurch, daß man mitzählt, wieviele Söhne schon besucht wurden, und diese Zahl mit der Anzahl der Söhne vergleicht. Außerdem muß die Position `place` anders gewählt werden, und zwar wird hier statt des Mittelwertes der Positionen der beiden Söhne bzw. der Position des einen Sohnes um eins erhöht oder vermindert, stets der Mittelwert der Positionen aller Söhne genommen (Dieser muß dann natürlich nicht unbedingt ein ganzzahliger Wert sein, man wählt in dem Fall die nächstkleinere ganze Zahl). Im zweiten Durchlauf ist das einzige, was geändert werden muß, das **case**-Statement. Genau wie im ersten Durchlauf wird es so modifiziert, daß nicht nur zwei sondern mehrere Söhne eines Knotens besucht werden können. Dieser Algorithmus ist eine Kompromißlösung, die den Kriterien A2 (minimaler Platzverbrauch) und A4 (Vater zentriert über Söhnen) recht nahekommt. A4 wird nicht ganz erfüllt, da es jetzt sein kann, daß der Durchschnitt der Positionen zweier Söhne nicht ganzzahlig ist. Dies ist aber insofern nicht so tragisch, als daß man es nun ja mit Bäumen zu tun hat, deren Knoten auch mehr als zwei Söhne haben können, und es ohnehin nicht klar ist, wann man einen Vater von mehreren Söhnen als zentriert ansehen will. Auch der Platzbedarf ist nicht minimal, jedoch geringer als beim Algorithmus WS, da jetzt linke und rechte Söhne nicht mehr unterschieden werden, und somit ein Sohn direkt unter seinem Vater gezeichnet werden kann, wenn er der Einzige ist (siehe Abb. 5.10).



Baum mit Alg. WS Schmalere Version

Abbildung 5.9: Beispiel für einen Baum, der von Algorithmus WS breiter als nötig gezeichnet wird

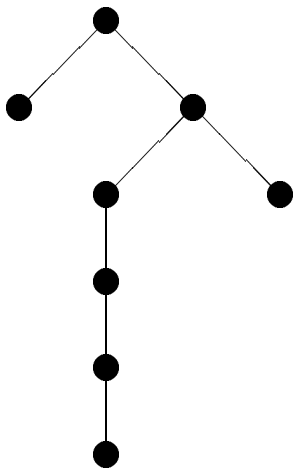


Abbildung 5.10: Der Baum aus Abb. 5.9, gezeichnet mit dem modifizierten Algorithmus WS

5.5 Beispiele

In diesem Abschnitt werden zwei Beispielbilder präsentiert, die direkt mit REL-VIEW gezeichnet wurden. Diese Beispiele zeigen, daß der Algorithmus zwar gute Bilder erzeugen kann, er jedoch in manchen Fällen auch seine Nachteile hat (ähnlich wie beim Algorithmus aus Kapitel 4).

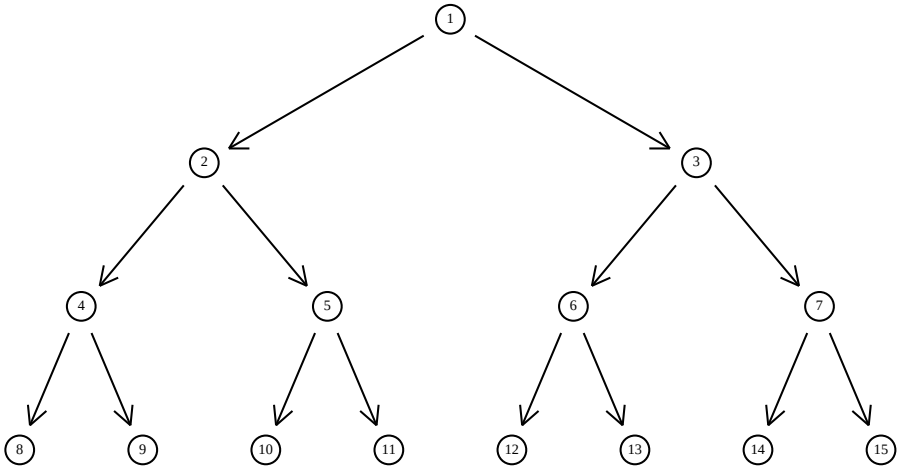


Abbildung 5.11: Beispiel 1

In Abb. 5.11 ist ein vollsändiger binärer Baum dargestellt, der so gezeichnet wird, wie man es erwartet.

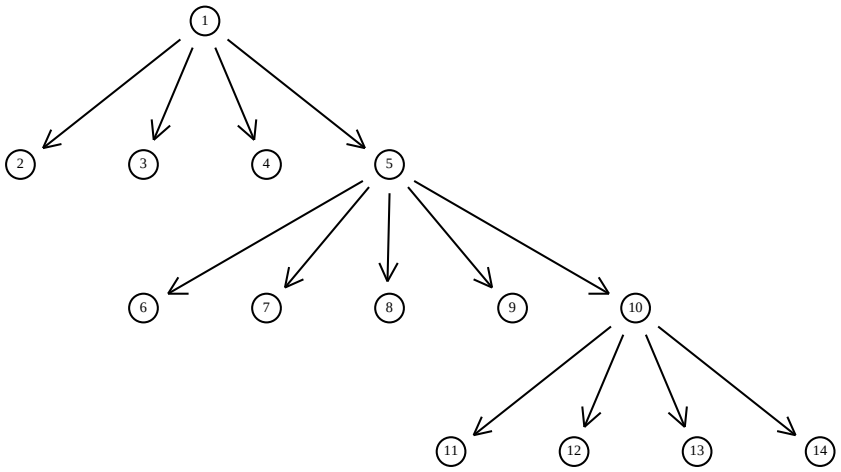


Abbildung 5.12: Beispiel 2

In Abb. 5.12 sehen wir einen Baum, der einen Nachteil des Algorithmus aufzeigt. Man könnte den Baum nämlich schmaler zeichnen, wenn man die Reihenfolge der Knoten in den einzelnen Schichten ändern würde. Dies wäre möglich, da die Reihenfolge der Knoten keinerlei Informationsgehalt hat, denn es kommt bei den Graphen im RELVIEW-System nur darauf an, welche Knoten durch eine Kante verbunden sind. Daß die Knoten in den einzelnen Schichten gerade so angeordnet sind, liegt natürlich daran, daß der Algorithmus so implementiert ist, daß er die Knoten in der Reihenfolge ihrer Numerierung bearbeitet. Im Großen und Ganzen kann man aber sagen, daß der Algorithmus Bäume doch sehr schön zeichnet.

Kapitel 6

Der Algorithmus KK

6.1 Einführung

Dieser Algorithmus von T. Kamada und S. Kawai ist ursprünglich für ungerichtete Graphen entwickelt worden [16]. Er wird hier auf gerichtete Graphen angewandt, indem man einfach die Richtung der Kanten vergißt, den Algorithmus auf den entstehenden ungerichteten Graphen anwendet, und dann die Kanten in ihrer ursprünglichen Orientierung zeichnet. Außerdem ist der Algorithmus nur auf zusammenhängende Graphen anwendbar. Will man nicht zusammenhängende Graphen zeichnen, so müssen die einzelnen Komponenten getrennt behandelt werden.

Die grundlegende Idee des Algorithmus ist, daß man die minimale Länge eines Pfades zwischen zwei Knoten als den gewünschten Abstand zwischen ihnen im Bild des Graphen ansieht. Dazu stellt man sich alle Paare von Knoten durch eine Feder der entsprechenden Länge verbunden vor, und sieht dann das Bild des Graphen als optimal an, in dem die Gesamtenergie dieses dynamischen Systems von Federn am geringsten ist. Die Idee, Graphen zu zeichnen, indem man Kanten durch Federn ersetzt, wurde zuerst von Eades in [12] vorgestellt.

Kanten werden hier als gerade Linien gezeichnet, es geht bei diesem Algorithmus also nur darum, die Positionen der Knoten festzulegen. Das ästhetische Kriterium, aufgrund dessen dieser Algorithmus entwickelt wurde, ist die globale Ausgewogenheit des Bildes, das heißt, die Knoten und Kanten sollen möglichst gleichmäßig verteilt sein. Es gilt nun also eine Möglichkeit zu finden, wie die globale Ausgewogenheit beurteilt werden soll. Das geschieht hier durch die Berechnung der Summe der Quadrate der Differenz von gewünschtem Abstand und tatsächlichem Abstand zwischen allen Knotenpaaren.

6.2 Das Federmodell

Wir führen ein dynamisches System ein, in dem n ($= |V|$) Punkte durch Federn verbunden sind. Seien $p_1, p_2, \ldots, p_n$ die Punkte in der Ebene, die den Knoten $v_1, v_2, \ldots, v_n \in V$ entsprechen. Wir betrachten nun ein Bild des Graphen als ausgeglichen, wenn das entsprechende System von Federn ausgeglichen ist. Das Maß an Unausgewogenheit beschreiben wir als die gesamte Energie der Federn,

das heißt,

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} (|p_i - p_j| - l_{ij})^2. \tag{6.1}$$

Ausgewogene Bilder erhält man also, wenn man die Energie E möglichst weit vermindert, und das Beste ist dasjenige, das einer Minimierung der Energie in unserem Modell entspricht. Die Originallänge l_{ij} der Feder zwischen v_i und v_j entspricht dem gewünschten Abstand zwischen ihnen im Bild, und wird folgendermaßen bestimmt. Der Abstand d_{ij} zwischen zwei Knoten v_i und v_j ist definiert als die Länge eines minimalen Pfades zwischen v_i und v_j. Die Länge l_{ij} ist dann definiert als

$$l_{ij} = L \cdot d_{ij}, \tag{6.2}$$

wobei L die gewünschte Länge einer Kante in der Darstellung des Graphen ist. Ist der Platz für das Bild des Graphen beschränkt, so kann man L abhängig von dem größten Abstand zwischen zwei Knoten wählen, nämlich

$$L = L_0 / \max_{i<j} d_{ij}, \tag{6.3}$$

wobei L_0 die Länge einer Seite der vorhandenen Zeichenfläche ist.
Der Parameter k_{ij} ist die Stärke der Feder zwischen v_i und v_j, und wird wie folgt bestimmt. Ohne die Federkonstante k_{ij} ist der Ausdruck (6.1) einfach die Aufsummierung der Quadrate der Differenzen von gewünschtem und tatsächlichem Abstand zwischen p_i und p_j. Es erscheint nun sinnvoller, hier nur die Differenz auf einer Einheitslänge zu betrachten, um so den Federn, die länger sind auch einen entsprechend größeren Spielraum zu gewähren. Wenn zum Beispiel eine Feder zwischen zwei Knoten liegt, die den Abstand $d_{ij} = 3$ haben, betrachten wir nur ein Drittel der Differenz zwischen gewünschtem und tatsächlichem Abstand der Punkte p_i und p_j. Also definieren wir k_{ij} als

$$k_{ij} = K / d_{ij}^2, \tag{6.4}$$

wobei K eine Konstante ist. Die Parameter l_{ij} und k_{ij} sind symmetrisch, das heißt, $l_{ij} = l_{ji}$ und $k_{ij} = k_{ji}$ $(i \neq j)$. Zwei wichtige Eigenschaften dieses Modells sind, daß die Knoten nicht zu dicht zusammenliegen, weil sie durch die Spannung der Federn auseinandergehalten werden, und daß symmetrische Graphen symmetrische Federsystemen liefern, was bei Minimierung der Energie zu symmetrischen Bildern führt.

6.3 Lokale Minimierung der Gesamtenergie

Um den Algorithmus zu beschreiben, benötigen wir einige Definitionen und Feststellungen. Die Position eines Punktes wird durch X- und Y-Koordinaten ausgedrückt. Seien $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ die Koordinaten der entsprechenden Punkte $p_1, p_2, \ldots, p_n$. Dann kann die in (6.1) definierte Energie E mithilfe dieser $2n$ Variablen folgendermaßen ausgedrückt werden:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} \Big\{ (x_i - x_j)^2 + (y_i - y_j)^2 + l_{ij}^2 - 2 l_{ij} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \Big\} \tag{6.5}$$

Wir wollen nun Werte für diese Variablen berechnen, die

$$E(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$$

minimieren (Die Parameter der Funktion E werden ab jetzt wegelassen). Da es sehr schwer ist, das Minimum zu berechnen, werden wir statt dessen ein lokales Minimum berechnen. Dazu stellen wir hier eine Methode zur Berechnung eines lokalen Minimums [17] vor, die auf der Newton-Raphson Methode beruht. Die notwendige Bedingung für ein lokales Minimum ist

$$\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0 \quad \text{für } 1 \le m \le n. \tag{6.6}$$

Eine Positionierung der Knoten, die (6.6) erfüllt, entspricht einem dynamischen Status des Federsystems, in dem sich alle Kräfte gegenseitig ausgleichen. Die partiellen Ableitungen von (6.5) nach x_m bzw. y_m sind

$$\frac{\partial E}{\partial x_m} = \sum_{i \neq m} k_{mi} \left\{ (x_m - x_i) - \frac{l_{mi}(x_m - x_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{1/2}} \right\}, \tag{6.7}$$

$$\frac{\partial E}{\partial y_m} = \sum_{i \neq m} k_{mi} \left\{ (y_m - y_i) - \frac{l_{mi}(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{1/2}} \right\}. \tag{6.8}$$

Wir müssen nun dieses Gleichungssystem aus $2n$ nichtlinearen Gleichungen lösen. Dieses kann aber nicht durch ein $2n$-dimensionales Newton-Raphson-Verfahren erledigt werden, weil die Gleichungen nicht voneinander unabhängig sind. Deshalb beschreiten wir einen anderen Weg. Wir bewegen immer nur einen Knoten zu einem stabilen Punkt, und lassen die Anderen fest. Wir betrachten also E als Funktion nur in den Variablen x_m und y_m, und berechnen dann ein lokales Minimum von E mit einem zwei-dimensionalen Newton-Raphson Verfahren. Wir erhalten ein lokales Minimum von E, indem wir diesen Schritt iterieren. In jedem Schritt wählen wir den Knoten, der den größten Δ_m-Wert hat, der folgendermaßen definiert ist.

$$\Delta_m = \sqrt{\left\{ \frac{\partial E}{\partial x_m} \right\}^2 + \left\{ \frac{\partial E}{\partial y_m} \right\}^2} \tag{6.9}$$

Ausgehend von der aktuellen Position $(x_m^{(0)}, y_m^{(0)}) = (x_m, y_m)$ wird der folgende Schritt wiederholt ausgeführt:

$$x_m^{(t+1)} = x_m^{(t)} + \delta x, \qquad y_m^{(t+1)} = y_m^{(t)} + \delta y \quad \text{für } t = 0, 1, 2, \ldots \tag{6.10}$$

Die Unbekannten δx und δy erfüllen die folgenden linearen Gleichungen:

$$\frac{\partial^2 E}{\partial x_m^2}\left(x_m^{(t)}, y_m^{(t)}\right)\delta x + \frac{\partial^2 E}{\partial x_m \partial y_m}\left(x_m^{(t)}, y_m^{(t)}\right)\delta y = -\frac{\partial E}{\partial x_m}\left(x_m^{(t)}, y_m^{(t)}\right) \quad (6.11)$$

$$\frac{\partial^2 E}{\partial y_m \partial x_m}\left(x_m^{(t)}, y_m^{(t)}\right)\delta x + \frac{\partial^2 E}{\partial y_m^2}\left(x_m^{(t)}, y_m^{(t)}\right)\delta y = -\frac{\partial E}{\partial y_m}\left(x_m^{(t)}, y_m^{(t)}\right) \quad (6.12)$$

Die Koeffizienten der obigen Gleichungen (6.11) und (6.12), die gerade die Elemente der Jacobi-Matrix sind, werden durch partielles Ableiten von (6.7) und (6.8) nach x_m bzw. y_m wie folgt berechnet.

$$\frac{\partial^2 E}{\partial x_m^2} = \sum_{i \neq m} k_{mi}\left\{1 - \frac{l_{mi}(y_m - y_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}}\right\} \quad (6.13)$$

$$\frac{\partial^2 E}{\partial x_m \partial y_m} = \sum_{i \neq m} k_{mi}\frac{l_{mi}(x_m - x_i)(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}} \quad (6.14)$$

$$\frac{\partial^2 E}{\partial y_m \partial x_m} = \sum_{i \neq m} k_{mi}\frac{l_{mi}(x_m - x_i)(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}} \quad (6.15)$$

$$\frac{\partial^2 E}{\partial y_m^2} = \sum_{i \neq m} k_{mi}\left\{1 - \frac{l_{mi}(x_m - x_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}}\right\} \quad (6.16)$$

Aus Gleichung (6.13) - (6.16) können δx und δy berechnet werden. Die Iteration (6.10) terminiert, wenn der Wert Δ_m bei $(x_m^{(t)}, y_m^{(t)})$ klein genug wird.

6.4 Der Algorithmus

Für den Zeichenalgorithmus muß zuerst die Entfernung d_{ij} für alle Knoten v_i, v_j berechnet werden. Einen einfachen Algorithmus hierfür findet man bei Floyd [13]. Danach werden l_{ij} und k_{ij} mittels (6.2), (6.3) und (6.4) aus d_{ij} berechnet. Bevor die Iteration gestartet wird, werden für die Punkte p_i initiale Werte vergeben. Versuche haben gezeigt, daß die initialen Positionen keinen großen Einfluß auf das Ergebnis haben. Wir verteilen die Punkte einfach gleichmäßig über den Bildschirm. Nach der Initialisierung wird die Energie E Schritt für Schritt reduziert, indem jeweils ein Knoten auf eine stabile Position gerückt wird. Der Algorithmus ist kurz dargestellt in Abb. 6.1.

Wir wollen noch kurz die Komplexität des Algorithmus betrachten. Diese ist mindestens $O(n^3)$, da soviel Zeit schon für die Berechnung von d_{ij} benötigt wird (für sehr große Graphen könnte man auch schnellere Algorithmen [14] [15] hierfür benutzen), der Rechenaufwand wird aber hauptsächlich durch die beiden verschachtelten **while** Schleifen bestimmt. In der inneren Schleife, die die Newton-Raphson Methode realisiert, wird Zeit $O(n)$ benötigt, um Δ_m, δx und δy in jedem Schritt zu berechnen. In der äußeren Schleife wird Zeit $O(n)$ benötigt, um das Maximum der Δ_i zu berechnen, da die neuen Δ_i (nach Verschieben von p_m) jeweils in $O(1)$ berechnet werden können, wenn man sich die alte Position von p_m merkt. Also ist die gesamte Zeit, die für die **while**-Schleifen gebraucht wird, $O(Tn)$, wobei T die gesamte Anzahl der Ausführungen der inneren Schleife ist. Es ist schwer, T abzuschätzen, weil T erstens von dem gegebenen Graphen und hier besonders von der Zahl der Knoten und zweitens von

1. berechne d_{ij} für $1 \leq i \neq j \leq n$;
2. berechne l_{ij} für $1 \leq i \neq j \leq n$;
3. berechne k_{ij} für $1 \leq i \neq j \leq n$;
4. initialisiere $p_1, p_2 \ldots, p_n$;
5. **while** $\max_i \Delta_i > \epsilon$
6. **begin**
7. Sei p_m der Punkt mit $\Delta_m = \max_i \Delta_i$;
8. **while** $\Delta_m > \epsilon$
9. **begin**
10. berechne δx und δy durch Lösen von (11) und (12);
11. $x_m = x_m + \delta x$;
12. $y_m = y_m + \delta y$;
13. **end**
14. **end**

Abbildung 6.1: Algorithmus KK

den Ausgangspositionen abhängt. Ein effektives Mittel um die Laufzeit zu reduzieren, ist auf jeden Fall, die Schranke ϵ in der Abbruchbedingung der beiden **while**-Schleifen zu vergrößern, womit man natürlich eine Verschlechterung der Qualität der Bilder in Kauf nehmen muß.

6.5 Beispiele

Wir wollen hier einige Beipielgraphen betrachten, für die der Algorithmus KK gute Bilder liefert, andere, für die er nur sehr unbefriedigende Bilder liefert, und auch solche, bei denen das Bild kleine aber auffällige Mängel aufweist. Die folgenden Bilder wurden wiederum von RELVIEW erzeugt.

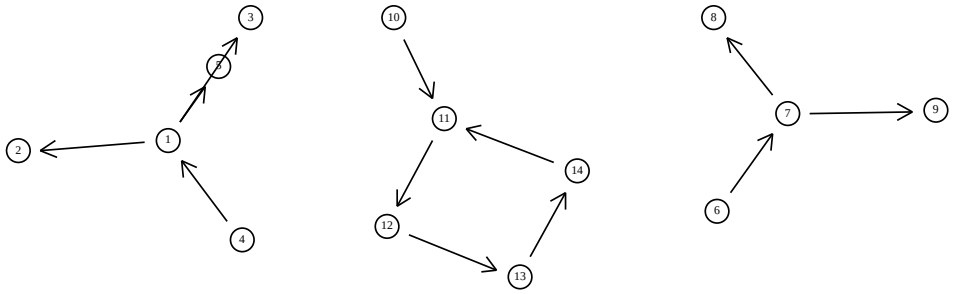


Abbildung 6.2: Beispiel 1

Abb. 6.2 zeigt, daß der Algorithmus nicht immer optimale Bilder liefert. Knoten Nr. 5 ist hier sicherlich schlecht plaziert. Außerdem würde man sich vielleicht wünschen, daß die Zusammenhangskomponenten nicht nebeneinander gezeichnet sondern besser auf das gesamte Bild verteilt werden (vergleiche hierzu Abb.

7.4). Welche Möglichkeit man vorzieht, ist natürlich eine Frage des Geschmacks.

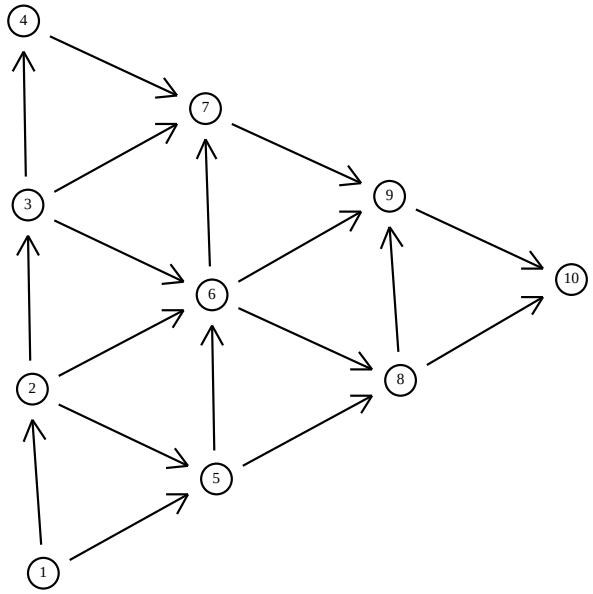


Abbildung 6.2: Beispiel 2

Der Graph aus Abb. 6.3 ist sicherlich so gezeichnet, wie man sich es wünschen würde.

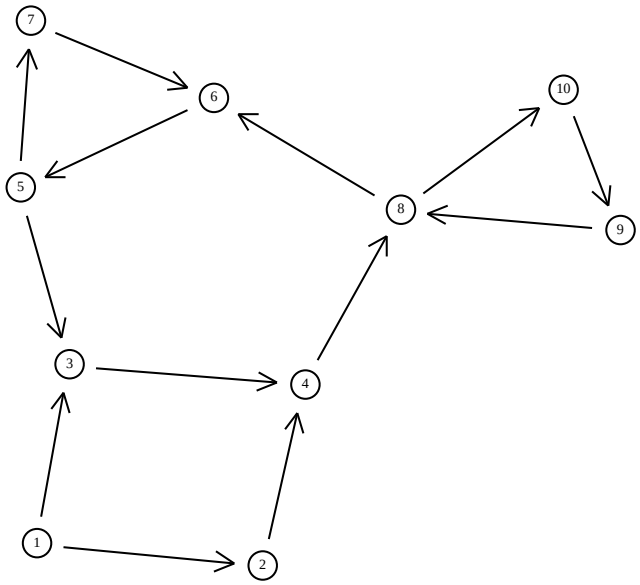


Abbildung 6.4: Beispiel 3

Auch Abb. 6.4 zeigt wieder ein vernünftiges Bild.

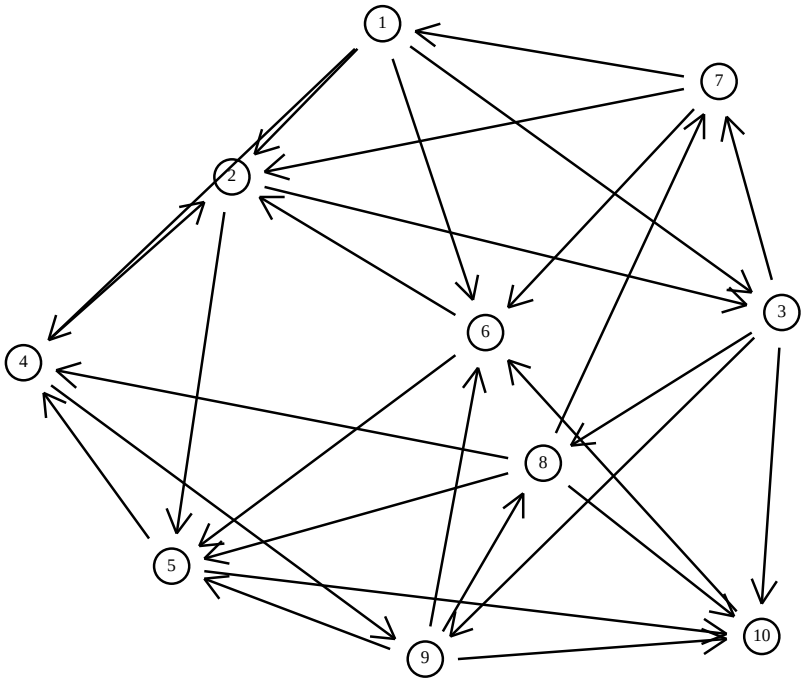


Abbildung 6.5: Beispiel 4

In Abb. 6.5 sehen wir nun einen Graphen, der relativ viele Kanten hat. Dieser Graph wird schon ziemlich unübersichtlich gezeichnet. Daß es besser geht, können wir in Abb. 4.13 sehen, wo derselbe Graph dargestellt ist.

Zum Algorithmus KK läßt sich allgemein sagen, daß er für Graphen, die nicht sehr komplex sind, das heißt, die wenig Kanten haben, ganz brauchbare Bilder macht. Der Vorteil von Algorithmus KK ist, daß er sehr kompakte Bilder macht, was aber bei etwas komplexeren Graphen sehr schnell dazu führt, daß sie unübersichtlich dargestellt werden.

Kapitel 7

Der Algorithmus FR

7.1 Einführung

Auch dieser Algorithmus, der von Thomas M. J. Fruchterman und Edward M. Reingold entwickelt wurde [18], ist, wie auch der Algorithmus von Kamada und Kawai, ursprünglich für ungerichtete Graphen konzipiert worden, wird bei uns aber auf gerichtete Graphen angewandt, indem man vorübergehend die Orientierung der Kanten unbeachtet läßt.

Dieser Algorithmus ist eine Modifikation von Eades' Algorithmus [12]. Also wird auch hier wieder ein physikalisches Modell herangezogen, um eine Positionierung der Knoten zu bekommen. Und auch bei diesem Algorithmus ist es wieder das Ziel, ein möglichst ausgeglichenes Bild zu erhalten, das heißt die Kanten sollen gleich lang und die Knoten gleichmäßig verteilt sein. Genau verfolgen wir hier folgende Ziele:

1. Knoten, die durch eine Kante verbunden sind, sollen dicht beieinander liegen.

2. Knoten sollen nicht zu dicht beieinander liegen.

Wie dicht Knoten nun genau zusammen liegen sollen, hängt davon ab, wieviel Zeichenfläche vorhanden ist, und wieviele Knoten der Graph hat. Wie wir bereits gesehen haben, lassen sich manche Graphen aufgrund ihrer Struktur überhaupt nicht schön darstellen. Die groben Richtlinien, nach denen der Algorithmus laufen soll, sind an Erkenntnisse der Atomphysik angelehnt:

> Bei einer Entfernung von etwa 1 fm (fm = femto-meter) wirkt die starke Kernkraft anziehend, und ist ungefähr zehnmal so stark wie die elektrische Kraft zwischen zwei Protonen. Die Kraft nimmt mit zunehmender Entfernung sehr schnell ab und ist bei etwa dem 15-fachen dieser Entfernung völlig vernachlässigbar. Bei einer Entfernung von ungefähr 0.4 fm wirken die starken Kernkräfte abstoßend, so daß die Elementarteilchen nicht kollabieren.

Betrachten wir nun die folgende Analogie: wir sehen die Knoten als Elementarteilchen an, die abstoßende und anziehende Kräfte aufeinander ausüben, und

so Bewegungen der Knoten hervorrufen. Der Algorithmus ähnelt so einer Simulation von Molekülen oder Planetensystemen. Natürlich ist es nicht notwendig, daß wir eine exakt der Natur entsprechende Simulation durchführen. Wir können uns die Freiheit nehmen, unrealistische Kräfte auf unrealistische Art und Weise anzuwenden, wie es bereits Eades in seiner Arbeit getan hat. Wie auch Eades, betrachten wir Anziehungskräfte nur zwischen Knoten, die durch eine Kante verbunden sind, Abstoßungskräfte jedoch zwischen allen Knoten. Dies ist auch konsistent mit der Asymmetrie der obigen Richtlinien.

Erwähnenswert ist noch die folgende Diskrepanz zwischen den natürlichen Systemen wie die durch Kernkräfte zwischen Elementarteilchen entstehen, die die Inspiration zu diesem Algorithmus geliefert haben, und der Art, wie diese Systeme hier simuliert werden. Hier werden Kräfte benutzt, um Geschwindigkeiten und damit ein Displacement zu berechnen. In der Natur dagegen induzieren Kräfte Beschleunigung, was dazu führt, daß sich dynamische Gleichgewichte herausbilden, wir suchen jedoch nach statischen Gleichgewichten.

7.2 Der Algorithmus

Der Algorithmus läuft über eine vorgegebene Anzahl von Iterationen, die man experimentell ermittelt (in der aktuellen Implementierung im RELVIEW-System sind es 50). Jede Iteration besteht aus drei Schritten.

- Die Berechnung der Wirkung der Anziehungskräfte auf jeden Knoten,
- die Berechnung der Wirkung der Abstoßungskräfte auf jeden Knoten und
- die Begrenzung des Displacements bezüglich der Temperatur und der Zeichenfläche.

Ein spezieller Fall tritt auf, wenn zwei Knoten die gleiche Position haben. Dann werden die Knoten so behandelt, als ob sie ein klein wenig voneinander entfernt wären, was zu einem sehr starken Abstoßungseffekt führt, wodurch die Knoten voneinander getrennt werden. Ein zentraler Begriff des Algorithmus ist die Temperatur. Die Idee hierbei ist, daß die Knoten zu Beginn eine große Bewegungsfreiheit haben, die dann im Verlauf des Algorithmus abnimmt. Genauer wird das Displacement der Knoten in jeder Iteration auf einen bestimmten maximalen Wert beschränkt, und diesen Wert verringert man dann im Laufe der Zeit. So wird das Ausmaß der Korrekturen immer feiner, je besser das Bild wird. Als Beispiel könnte man die Temperatur mit einem beliebigen Wert initialisieren, und sie dann linear bis auf Null sinken lassen. Wir werden später noch bessere Methoden hierzu kennenlernen. Die Begrenzung des Displacements bezüglich der Zeichenfläche bedeutet, daß man es nicht zuläßt, daß Knoten Positionen zugewiesen werden, die über den Rand der Zeichenfläche hinaus gehen. Dieser Punkt kann entfallen, wenn entweder die Zeichenfläche nicht beschränkt ist oder wenn man nach Durchführung des Algorithmus das Bild so verkleinert, daß es auf die Zeichenfläche paßt. Meiner Meinung nach erhält man mit letzterer Methode bessere Bilder.

```
 1. area = width * length;
 2. assign initial positions;
 3. k = sqrt(area/|V|)
 4. function f_a(x) = begin return  x^2/k end;
 5. function f_r(x) = begin return  k^2/x end;
 6. for i = 1 to iterations do begin
 7.    for v in V do begin
 8.       v.disp = 0;
 9.       for u in V do
10.          if u ≠ v then begin
11.             Δ = v.pos − u.pos;
12.             v.disp = v.disp + (Δ/|Δ|) * f_r(|Δ|)
13.          end
14.    end
15.    for e in E do begin
16.       Δ = e.v.pos − e.u.pos;
17.       e.v.disp = e.v.disp −(Δ/|Δ|) * f_a(|Δ|);
18.       e.u.disp = e.u.disp +(Δ/|Δ|) * f_a(|Δ|);
19.    end
20.    for v in V do begin
21.       v.pos = v.pos + (v.disp/|v.disp|) * min(v.disp,t);
22.       v.pos.x = min(width/2,max(−width/2,v.pos.x));
23.       v.pos.y = min(length/2,max(−length/2,v.pos.y));
24.    end
25.    t = cool(t);
26. end
```

Abbildung 7.1: Der Algorithmus von Fruchterman und Reingold

Der Algorithmus ist in Abb. 7.1 dargestellt:

1–3: Die Variablen `area` und `k` werden initialisiert. Dabei sind `width` und `length` die Länge bzw. die Breite der Zeichenfläche. `k` ist dann der Radius der leeren Fläche um jeden Knoten, wenn die Knoten gleichförmig auf der Zeichenfläche verteilt sind. Weiter werden den Knoten hier initiale Positionen zugewiesen. Dies kann auf vielfältige Art und Weise geschehen. Wir verteilen die Knoten einfach möglichst gleichmäßig. Denkbar wäre zum Beispiel auch, die Knoten auf die Ecken eines regelmäßigen n-Ecks zu plazieren, oder sie zufällig auf der Zeichenfläche zu verteilen.

4–5: Die Funktionen, die die Abstoßungskräfte bestimmen, werden hier definiert.

6–26: In jeder Iteration werden die drei Schritte Berechnung der Abstoßungskräfte, Berechnung der Anziehungskräfte und Begrenzung bezüglich Temperatur ausgeführt.

8–13: Hier werden die Abstoßungskräfte, die von allen anderen Knoten auf `v` wirken, bzw. die daraus resultierenden Displacements aufsummiert.

15–19: Die Anziehungskräfte zwischen benachbarten Knoten bzw. die daraus resultierenden Displacements werden berechnet und auf das `disp`-Feld der entsprechenden Knoten aufaddiert (eine Kante ist ein geordnetes Paar von Knoten `.u` und `.v`).

21: Das Displacement wird auf die aktuelle Temperatur t begrenzt, und die Knoten werden auf ihre neuen Positionen gesetzt.

22–23: Es kann sein, daß die Positionen der Knoten nun außerhalb der Zeichenfläche liegen. Dies wird hier wieder rückgängig gemacht, das heißt, die Knoten werden auf den Punkt auf dem Rand der Zeichenfläche gesetzt, der ihrer jetzigen Position am nächsten ist. Diese Begrenzung auf die Zeichenfläche kann unter gewissen Vorraussetzungen auch weggelassen werden.

25: Die Temperatur `t` wird durch die Funktion `cool` für die nächste Iteration neu bestimmt. Wie man die Funktion `cool` wählen kann, wird noch gesondert beschrieben.

Wir wollen noch kurz die Komplexität des Algorithmus betrachten. Jede Iteration braucht Zeit $O(|V|^2 + |E|)$. Die Frage ist, wie viele Iterationen nötig sind, um einen Graphen schön darzustellen. Die Situation ist hier ähnlich wie bei dem Algorithmus von Kamada und Kawai, bei dem man schlecht abschätzen konnte, wie lange er läuft. Deshalb läßt sich die hier benutzte feste Anzahl von 50 Iterationen auch kaum theoretisch rechtfertigen. Wir können nur feststellen, daß sich in der Praxis gezeigt hat, daß diese Anzahl auch für komplexere Graphen vernünftige Ergebnisse liefert. Für kleinere Graphen ist die Zahl 50 vielleicht etwas überdimensioniert, aber da die Iterationen bei kleinen Graphen ohnehin sehr wenig Zeit in Anspruch nehmen, ist es nicht schlimm, wenn man ein paar Iterationen mehr als nötig durchführt.

7.3 Wahl der Funktionen f_a, f_r und *cool*

Wir wollen zunächst noch einmal die Wahl der Funktionen, die die Abstoßungs- und Anziehungskräfte modellieren, betrachten. Wir berechnen k, die optimale Entfernung zweier benachbarter Knoten als

$$k = sqrt\left(\frac{\text{area}}{\text{number of vertices}}\right)$$

Die Idee ist, daß die Knoten gleichmäßig verteilt werden und k der Radius der leeren Fläche um einen Knoten ist. Intuitiv wird man sagen, je weiter zwei

Knoten auseinander liegen bzw. je dichter sie zusammen liegen, desto weniger kann man das aktuelle Bild akzeptieren und desto stärker muß die Korrektur ausfallen. Bezeichnen wir mit f_a und f_r die Anziehungs- bzw. die Abstoßungskräfte (r wie repulsive) und mit d die Entfernung zwischen zwei Knoten dann definieren wir wie folgt:

$$f_a(d) = d^2/k$$

$$f_r(d) = -k^2/d$$

Dann haben die Funktionen die folgenden Eigenschaften:

- Der Betrag von $f_a(d)$ wächst sehr schnell, wenn d größer als k ist.
- Der Betrag von $f_r(d)$ wächst sehr schnell, wenn d kleiner als k ist.
- Bei der idealen Entfernung $d = k$, heben sich die Kräfte f_a und f_r gerade gegenseitig auf.

Natürlich gibt es auch andere Möglichkeiten, die Funktionen f_a und f_r zu definieren. Fruchterman und Reingold [18] haben zum Beispiel auch mit folgenden Funktionen experimentiert:

$$f_a(d) = d/k$$

$$f_r(d) = -k/d$$

Diese Funktionen funktionierten jedoch für etwas komplexere Graphen sehr schlecht, da es nicht möglich schien, lokale Minima zu überwinden. Dies hatte zur Folge, daß ein Knoten nicht an einem anderen Knoten vorbeibewegt werden konnte, der ihm im Weg war, etwa durch eine schlechte initiale Plazierung. Funktionen mit höheren Potenzen bringen gegenüber den quadratischen Funktionen auch keine Verbesserungen, sie kosten bloß mehr Zeit bei der Berechnung. Die Funktionen, die Eades benutzt hat, waren

$$f_a(d) = k_a \log d$$

$$f_r(d) = -k_r/d^2$$

. Die Ergebnisse von Eades waren in etwa die gleichen wie die von Fruchterman und Reingold, jedoch ist die Funktion f_a von Eades nicht so effizient zu berechnen.

Die Funktion *cool* verbindet hier zwei Ideen miteinander. Einerseits soll sie so definiert werden, daß es möglich ist, lokale Minima zu überwinden, das heißt, es muß möglich sein, in einem Schritt große Veränderungen vorzunehmen. Andererseits soll sie es ermöglichen, durch Feinabstimmung sehr dicht an einen Status mit minimaler Energie heranzukommen. Man verwirklicht dies, indem man den Algorithmus in zwei Phasen aufteilt. In der Ersten beginnt man mit einer hohen Temperatur, die man dann schnell sinken läßt, um so eine Plazierung der Knoten zu finden, die schon relativ gut ist. In der zweiten Phase läßt man die Temperatur auf einem konstant niedrigen Wert, um so eine Feinabstimmung zu erreichen. Experimente haben gezeigt, daß diese Methode deutlich bessere Bilder produziert, als es bei der Benutzung einer Funktion der Fall wäre, die die Temperatur über den gesamten Verlauf des Algorithmus stetig senkt.

7.4 Zeichnen nicht zusammenhängender Graphen

Der Algorithmus in der jetzigen Form ist nur für zusammenhängende Graphen geeignet. Er läßt sich zwar auf beliebige Graphen anwenden, jedoch treten bei nicht zusammenhängenden Graphen folgende Probleme auf. Verwendet man den Algorithmus in der Form, die die Zeichenfläche von vornherein beschränkt, so werden die einzelnen Komponenten an den Rand gedrückt und in der Mitte bleibt der meiste Platz ungenutzt. Dies kommt daher, daß die Knoten aus den verschiedenen Komponenten sich nur gegenseitig abstoßen, da es ja keine Kanten zwischen den Komponenten gibt, die Anziehungskräfte hervorrufen würde. Aus demselben Grund tritt bei der Version, die die Zeichenfläche nicht begrenzt, das Problem auf, daß die verschiedenen Zusammenhangskomponenten mit jeder Iteration weiter auseinander streben würden. Diese Schwierigkeiten werden nun folgendermaßen überwunden. Bisher betrachteten wir in der Phase des Algorithmus, in der die Abstoßungskräfte berechnet werden, die Wirkung sämtlicher anderen Knoten auf einen Knoten v. Nun ist es aber so, daß die Kraft $f_r(d)$ mit zunehmender Entfernung d sehr schnell abnimmt, also Knoten, die weit entfernt sind, keinen großen Einfluß auf den Knoten v haben. Aufgrund dieser Erkenntnis modifizieren Fruchterman und Reingold den Algorithmus so: Die Abstoßungskräfte werden nur noch zwischen Knoten berechnet, die nicht weiter als $2k$ voneinander entfernt sind. Experimente haben gezeigt, daß dies auf die Qualität der Bilder keinen Einfluß hat. Der Vorteil ist jedoch, daß Komponenten sich nicht mehr gegenseitig abstoßen, wenn sie erst einmal einen gewissen Abstand voneinander haben, und der Algorithmus so auch für nicht zusammenhängende Graphen vernünftige Bilder liefert.

7.5 Beispiele

Auch zu diesem Algorithmus wollen wir einige durch RELVIEW erzeugte Beispielgraphen betrachten.

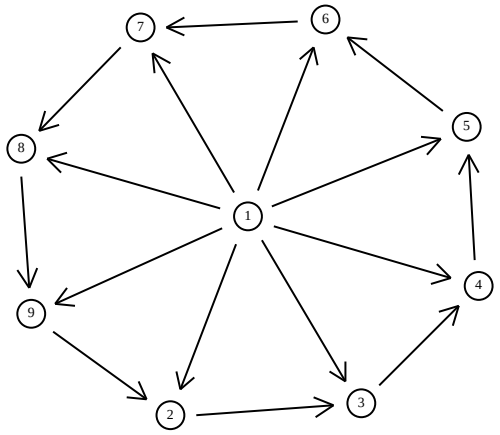


Abbildung 7.2: Beispiel 1

Dieser Graph wird hier besser gezeichnet, als in Abb. 4.14, da die Kanten hier

nicht so unnötig lang werden.

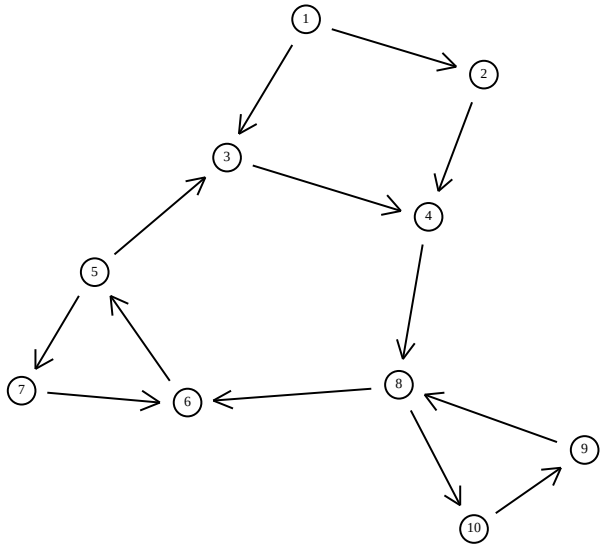


Abbildung 7.3: Beispiel 2

Der Graph in Abb. 7.3 ist schön dargestellt, ähnlich wie der in Abb. 6.4 vom Algorithmus KK erzeugte.

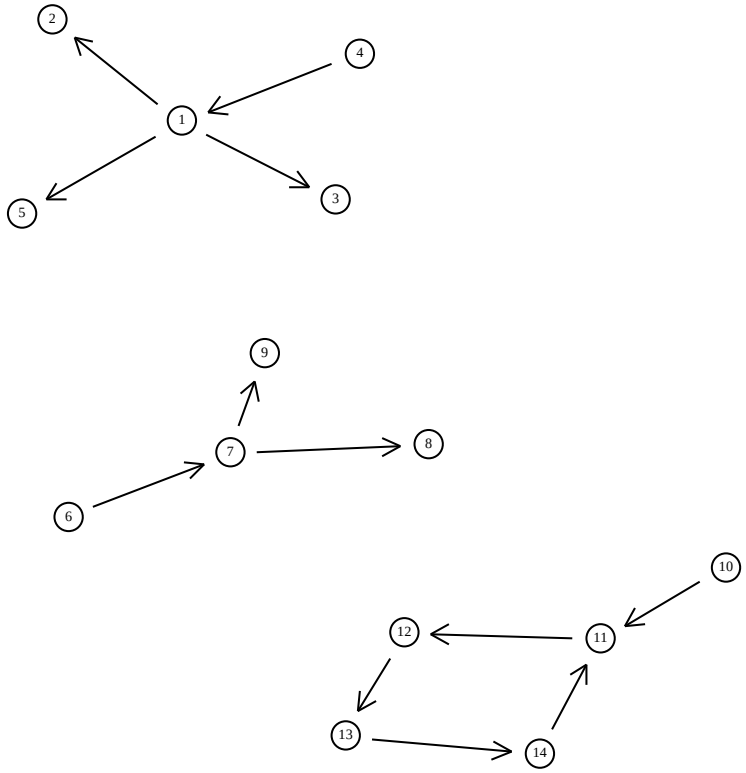


Abbildung 7.4: Beispiel 3

In Abb. 7.4 sieht man einen Vorteil des Algorithmus FR gegenüber dem Algorithmus KK, nämlich daß nicht zusammenhängende Graphen nicht erst in ihre Zusammenhangskomponenten zerlegt werden müssen, um sie dann nebeneinander zu zeichnen, sondern daß sie in einem Stück bearbeitet werden können (vergleiche hierzu Abb. 6.2).

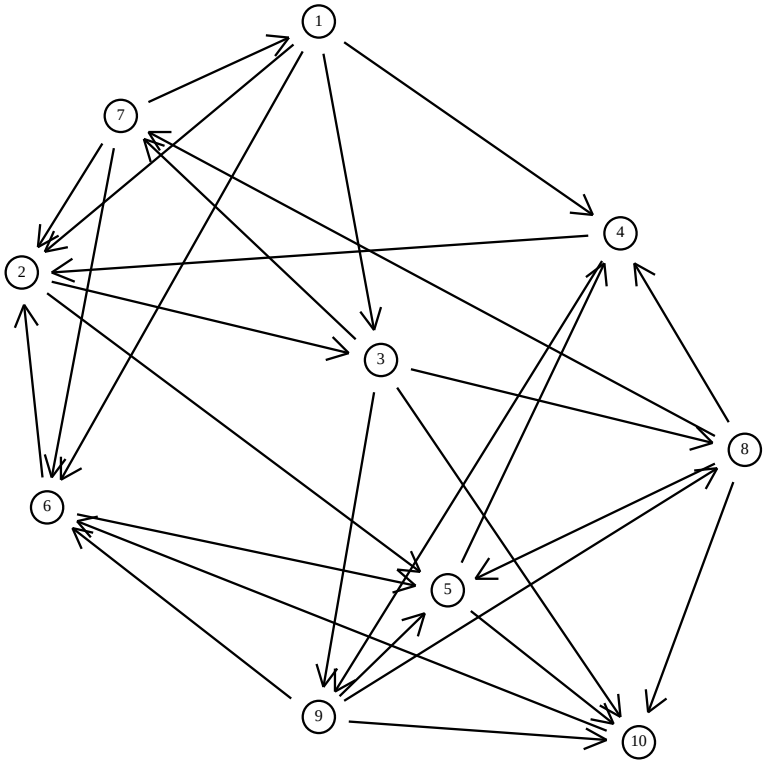


Abbildung 7.5: Beispiel 4

Abb. 7.5 zeigt nun den Graphen, der schon in Abb. 6.5 von KK gezeichnet und in Abb. 4.13 vom Algorithmus von Gansner et al. gezeichnet zu sehen ist. Auch hier ist er wie im Fall von KK sehr unübersichtlich gezeichnet, im Gegensatz zu dem Bild das der Algorithmus von Gansner et al. liefert.

Im Allgemeinen gilt für den Algorithmus FR das Gleiche wie für den Algorithmus KK. Für Graphen mit wenig Kanten ist er brauchbar, für komplexere Graphen nicht unbedingt. Beim Vergleich dieser beiden Algorithmen stellt man fest, daß Algorithmus KK meistens die etwas schöneren Bilder macht, jedoch ist der Algorithmus FR um einiges schneller.

Kapitel 8

Relationen und RELVIEW

8.1 Relationenalgebra

In diesem Abschnitt werden grundlegende Begriffe, die beim Arbeiten mit Relationen benötigt werden, eingeführt. Zuerst werden die Basisoperationen auf Relationen komponentenweise eingeführt. Dann betrachten wir Relationen mit speziellen Eigenschaften und beschäftigen uns mit einigen höheren Operationen auf Relationen. Hier benutzen wir statt der komponentenweisen Darstellung dann die zuvor eingeführten Basisoperationen. Dadurch wird die Lesbarkeit der Definitionen deutlich verbessert. Weiterführende Informationen findet man in dem Buch von Schmidt und Ströhlein [19].

8.1.1 Basisoperationen auf Relationen

Seien X und Y zwei Mengen. Eine Teilmenge R des Kartesischen Produkts $X \times Y$ heißt dann (*heterogene*) Relation zwischen X und Y mit Argumentbereich X und Wertebereich Y. Statt $R \subset X \times Y$ schreiben wir auch $R : X \leftrightarrow Y$ und $(X \leftrightarrow Y)$ bezeichnet die Mengen der Relationen zwischen X und Y. Sei $x \in X$ und $y \in Y$. Statt $(x, y) \in R$ schreiben wir kurz R_{xy}. Ist $X = Y$, so heißt die Relation R *homogen*. Sind X und Y endliche Mengen der Mächtigkeit m bzw. n, so können wir R als Boolesche Matrix mit m Zeilen und n Spalten darstellen. Diese Art der Repräsentation wird neben der Graphdarstellung auch im RELVIEW-System benutzt.

Durchschnitt, Vereinigung und Komplement und die dazugehörigen Konstanten können wie folgt eingeführt werden: Der *Durchschnitt* $R \cap S : X \leftrightarrow Y$ und die *Vereinigung* $R \cup S : X \leftrightarrow Y$ zweier Relationen $R, S : X \leftrightarrow Y$ werden komponentenweise definiert durch die Gleichungen

- $(R \cap S)_{xy} \quad \Leftrightarrow \quad R_{xy} \wedge S_{xy}$
- $(R \cup S)_{xu} \quad \Leftrightarrow \quad R_{xy} \vee S_{xy}$.

Die Symbole auf der rechten Seite $\wedge$ und $\vee$ bezeichnen die übliche Konjunktion bzw. Disjunktion auf den Wahrheitswerten. Das *Komplement* (oft auch als Negation bezeichnet) $\overline{R} : X \leftrightarrow Y$ ist komponentenweise definiert durch

- $\overline{R}_{xy} \quad \Leftrightarrow \quad \neg R_{xy}$,

wobei $\neg$ die Negation auf den Wahrheitswerten bezeichnet. Für die *Nullrelation* $O : X \leftrightarrow Y$ und die *Universalrelation* $L : X \leftrightarrow Y$ gilt:

- $0_{xy} \quad \Leftrightarrow \quad$ false
- $L_{xy} \quad \Leftrightarrow \quad$ true

Die Menge $(X \leftrightarrow Y)$ bildet mit den Operationen $\cup$, $\cap$ und $\overline{\ }$ eine vollständige Boolesche Algebra mit L als größtem Element, O als kleinstem Element und mit der Ordnung $\subset$ (*Inklusion*), die gegeben ist durch

- $R \subset S \quad \Leftrightarrow \quad \forall x \in X, y \in Y : R_{xy} \rightarrow S_{xy}$.

Als nächstes definieren wir die Komposition von Relationen und eine zugehörige Konstante. Die *Komposition* (oft auch als Produkt oder Multiplikation bezeichnet) $RS : X \leftrightarrow Z$ zweier Relationen $R : X \leftrightarrow Y$ und $S : Y \leftrightarrow Z$ ist komponentenweise definiert als

- $(RS)_{xz} \quad \Leftrightarrow \quad \exists y \in Y : R_{xy} \wedge S_{xy}$

und die *identische Relation* $I : X \leftrightarrow X$ ist der Gleichheitstest auf X, das heißt, die komponentenweise Definition lautet

- $I_{xy} \quad \Leftrightarrow \quad x = y$

Die Menge $(X \leftrightarrow X)$ bildet mit Komposition und Identischer Relation ein Monoid (das heißt, eine Halbgruppe mit neutralem Element). Ferner erhalten wir aus der Existenz des Produkts RS (mit R und S wie oben), daß QS für alle Relationen $Q : X \leftrightarrow Y$ definiert ist.

Die letzte der Basisoperationen auf Relationen ist die Transposition. Für eine Relation $R : X \leftrightarrow Y$ ist die *transponierte* Relation $R^T : Y \leftrightarrow X$ komponentenweise definiert durch

- $R^T_{xy} \quad \Leftrightarrow \quad R_{xy}$

Es ist klar, daß die Transposition involutorisch ist, und mit der Komplementbildung kommutiert. Der Zusammenhang zwischen Transposition und den Operationen der Booleschen Algebra und Komposition zeigt sich in der Dedekind Regel

$$(QR \cap S) \subset (Q \cap SR^T)(R \cap Q^T S),$$

die gilt, wenn alle drei geklammerten Ausdrücke definiert sind.

8.1.2 Spezielle homogene Relationen

Sei R eine homogene Relation. R ist *reflexiv* wenn $I \subset R$, *transitiv* wenn $R^2 \subset R$ und *antisymmetrisch* wenn $R \cap R^T \subset I$. Eine *partielle Ordnung* ist eine reflexive, antisymmetrische und transitive Relation.

Eine andere wichtige Klasse von homogenen Relationen sind *Äquivalenzrelationen*. Äquivalenzrelationen sind reflexiv, transitiv und symmetrisch. Eine Relation R ist symmetrisch, wenn gilt $R \subset R^T$.

8.1.3 Spezielle heterogene Relationen

Eine Relation R heißt (*partielle*) *Funktion*, wenn $R^T R \subset I$. Falls gilt $RL = L$ heißt R *totale* Funktion. Eine totale Funktion kann auch charakterisiert werden durch $R\overline{I} = \overline{R}$. Eine Relation R ist *injektiv*, wenn R^T eine partielle Funktion ist (das heißt, $RR^T \subset I$), und surjektiv, wenn R^T total ist (das heißt, $R^T L = L$). Eine surjektive und injektive Relation heißt bijektiv. Die Menge aller Funktionen von einer Menge X in eine Menge Y bezeichnen wir mit $(X \to Y)$.

8.1.4 Relationale Beschreibung von Teilmengen

Eine Relation $v : X \leftrightarrow Y$ mit $v = vL$ heißt zeilenkonstant. Solche Relationen können als Teilmenge von X, als Prädikate von X oder *Vektoren* betrachtet werden. Da für einen Vektor $v : X \leftrightarrow Y$ die Menge Y nicht relevant ist, bezeichnen wir mit V(X) die *Menge aller Vektoren* mit Argumentbereich X. Ist $v : X \leftrightarrow Y$ ein Vektor, so schreiben wir v_x statt v_{xy} für alle $y \in Y$, da $v_{xy_1} \Leftrightarrow v_{xy_2}$ für alle $y_1, y_2 \in Y$.

Ein *Punkt* ist eine Teilmenge, die genau ein Element enthält. Also ist ein Vektor genau dann ein Punkt, wenn er bijektiv ist: Surjektivität bedeutet, daß der Vektor eine Teilmenge beschreibt, die mindestens ein Element enthält, und Injektivität bedeutet, daß er eine Teilmenge beschreibt, die höchstens ein Element enthält. Die Menge aller Punkte wird mit P(X) bezeichnet.

Eine Boolesche Matrix repräsentiert genau dann einen Vektor, wenn jede Zeile nur Einsen oder nur Nullen enthält. Gibt es von den Zeilen, die nur Einsen enthalten genau eine, so stellt die Matrix einen Punkt dar.

Neben Vektoren gibt es noch eine zweite Möglichkeit zur Beschreibung von Teilmengen, nämlich injektive Abbildungen. Haben wir eine injektive Abbildung $i : Y \to X$, so nennen wir Y die durch i gegebene Teilmenge. Ist Y eine durch i gegebene Teilmenge von X, so beschreibt der Vektor $i^T L : X \leftrightarrow 1$ die Teilmenge Y im obigen Sinne (1 steht hier für eine einelementige Menge). Umgekehrt kann man auch eine injektive Abbildung aus einem gegebenen Vektor konstruieren.

8.1.5 Hüllen

Sei R eine homogene Relation. Die kleinste reflexive (symmetrische bzw. transitive) Relation, die R enthält, heißt *reflexive* (*symmetrische* bzw. *transitive*) *Hülle* von R. Mit den Basisoperationen aus 8.1.1 lassen sich die Hüllenbildungen folgendermaßen beschreiben:

- $I \cup R$ ist die reflexive Hülle von R.
- $R \cup R^T$ ist die symmetrische Hülle von R.
- $\bigcup_{n \geq 1} R^n$ ist die transitive Hülle von R.

wobei die Potenzen R^n für alle $n \geq 0$ rekursiv definiert sind durch $R^0 = I$ und $R^{n+1} = RR^n$.

Üblicherweise schreibt man R^+ für die transitive Hülle von R. Die reflexive Hülle hiervon $I \cup R^+$ heißt *reflexiv transitive Hülle* von R und man schreibt hierfür R^*.

8.1.6 Residuen und Quotienten

Residuen sind die größten Lösungen bestimmter Inklusionen. Das *Linksresiduum* von S über R (in Zeichen S/R) ist die größte Relation X mit $XR \subset S$; das Rechtsresiduum von S über R (in Zeichen R\S) ist die größte Relation X mit $RX \subset S$. Beide Residuen können auch mithilfe der Basisoperationen aus 8.1.1 beschrieben werden:

- $S/R = \overline{\overline{S}R^T}$
- $R\backslash S = \overline{R^T\overline{S}}$

Jedes Residuum kann auch durch das andere definiert werden, denn es gilt $(S/R)^T = R^T\backslash S^T$ und $(R\backslash S)^T = S^T/R^T$.

Relationen, die sowohl Rechts- als auch Linksresiduum sind, sind die symmetrischen Quotienten. Der *symmetrische Quotient* $\mathrm{syq}(R,S)$ zweier Relationen R und S ist definiert als die größte Relation X mit $RX \subset S$ und $XS^T \subset R^T$. Mit den Basisoperationen läßt sich der symmetrische Quotient ausdrücken durch

- $\mathrm{syq}(R,S) = \overline{R^T\overline{S}} \cap \overline{\overline{R^T}S}$.

In komponentenweiser Darstellung sind die Residuen allquantifizierte Implikationen und der symmetrische Quotient eine allquantifizierte Äquivalenz. Ist zum Beispiel $S : X \leftrightarrow Z$ und $R : Y \leftrightarrow Z$, so gilt für das Linksresiduum $S/R : X \leftrightarrow Y$

- $(S/R)_{xy} \quad \Leftrightarrow \quad \forall z(R_{yz} \rightarrow S_{xz})$

und ist $S : X \leftrightarrow Z$ und $R : X \leftrightarrow Y$, so gilt für den symmetrischen Quotienten $\mathrm{syq}(R,S) : Y \leftrightarrow Z$

- $\mathrm{syq}(R,S)_{yz} \quad \Leftrightarrow \quad \forall x(R_{xy} \leftrightarrow S_{xz})$.

8.2 Das RELVIEW-System

8.2.1 Allgemeines

Wenn man sich mit der Theorie der Relationen (Booleschen Matrizen) oder mit Graphentheorie beschäftigt, wird man häufig mit mehr oder weniger kleinen Beispielen konkreter Relationen arbeiten, um zu zeigen, daß diese Relationen gewisse Eigenschaften haben bzw. nicht haben. Das RELVIEW-System bietet nun die Möglichkeit, Relationen maschinenunterstützt zu manipulieren, statt dies mit Papier und Bleistift zu tun.

RELVIEW ist ein interaktives bildschirmorientiertes System zur Manipulation konkreter Relationen. Es wurde entwickelt an der Universität der Bundeswehr München, teilweise unterstützt durch die Deutsche Forschungsgemeinschaft. Das System wurde ständig erweitert, ist nun aber ausgehend von der Ursprungsversion an der Christian-Albrechts-Universität zu Kiel vollkommen neu entwickelt worden. RELVIEW betrachtet Relationen als Boolesche Matrizen. Diese können mithilfe der Maus auf dem Bildschirm erzeugt und manipuliert

werden. Die Ausführung von Kommandos erfolgt durch das Anklicken der entsprechenden Knöpfe. Neben den Basisoperationen wie Vereinigung oder Komplementbildung gibt es auch Kommandos zum Ausführen höherer Operationen wie Hüllenbildung oder Residuenbildung und bestimmte Test-Kommandos (zum Beispiel Test auf Transitivität). Außerdem gibt es noch Kommandos mit administrativem Charakter, um zum Beispiel Relationen in einer Datei zu speichern.

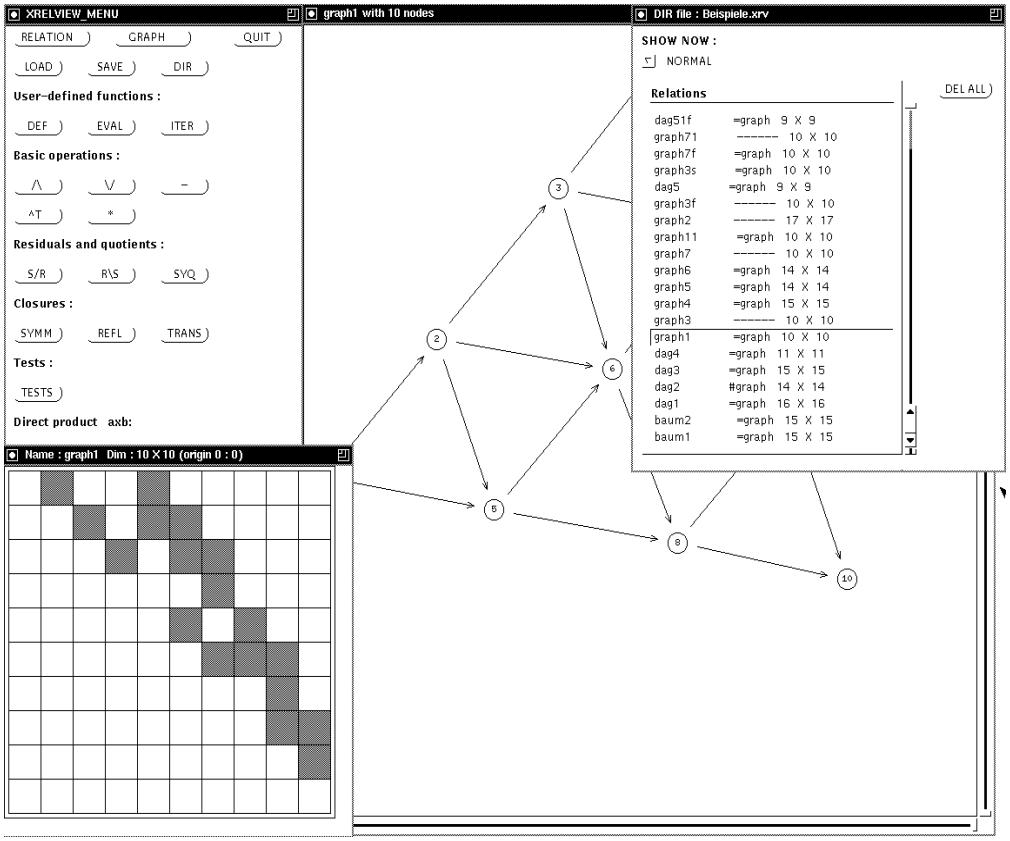


Abbildung 8.1: Der RELVIEW-Bildschirm

Der Bildschirm von RELVIEW ist in Abb. 40 zu sehen. Das RELVIEW-Menü-Fenster erscheint beim Aufruf von RELVIEW zuerst. Von diesem Fenster aus kann man durch Anklicken der entsprechenden Knöpfe weitere Fenster öffnen wie zum Beispiel das Relationen-, das Graph- und das Directory- Fenster. Außerdem befinden sich in diesem Fenster die Knöpfe zur Ausführung von Operationen auf Relationen und administrativen Funktionen, wie Laden und Speichern von Relationen und Graphen.

Im Directory-Fenster (im Bild rechts oben) kann man unter den Relationen, die man in den Arbeitsspeicher geladen hat, eine auswählen, die dann im Relationenfenster (im Bild links unten) als Boolesche Matrix dargestellt wird (In diesem Beispiel ist das die Relation graph1). Im Graphfenster (auf der rechten Seite des Bildschirms, hinter dem Directory-Fenster) kann man sich die Relation als Graph anzeigen lassen, falls sie quadratisch ist. Andersherum kann man auch im Graphfenster mithilfe der Maus einen Graph zeichnen, und sich diesen

dann im Relationenfenster als Boolesche-Matrix anzeigen lassen. Wir wollen in den folgenden Abschnitten die wichtigsten Grundbegriffe zum Arbeiten mit RELVIEW, die wir in 8.3 benötigen, betrachten. Dabei werden die einzelnen Fenster, die man beim Arbeiten mit RELVIEW öffnen kann, jeweils in einem eigenen Abschnitt erklärt.

8.2.2 Das RELVIEW-Menü-Fenster

In diesem Fenster befinden sich mehrere Knöpfe, die mit der linken Maustaste angeklickt werden können. Ganz oben befinden sich folgende Knöpfe:

- Der Knopf RELATION öffnet das Relationenfenster, in dem dann die im Directory-Fenster ausgewählte Relation angezeigt wird.
- Mit dem Knopf GRAPH öffnet man das Graphfenster, in dem dann die aktuelle Relation als Graph angezeigt wird, falls die Relation schon einmal als Graph gezeichnet wurde.
- Mit SAVE kann man den aktuellen Inhalt des Arbeitsspeichers in einer Datei ablegen. Dazu kann man nach Anklicken von SAVE einen Namen eingeben, der dann die Extension ".xrv" erhält, oder man kann in dem dann erscheinenden Menü den Namen einer bereits existierenden Datei auswählen.
- LOAD dient zum Laden einer durch SAVE abgespeicherten Datei in den Arbeitsspeicher. Den Namen der Datei kann man auch hier wieder aus dem erscheinenden Menü auswählen.
- DIR öffnet das Directory-Fenster, in dem der gesamte Inhalt des Arbeitsspeichers aufgelistet ist.
- Mit QUIT kann man die RELVIEW-Sitzung beenden.

Darunter befinden sich die Köpfe zum Arbeiten mit benutzerdefinierten Funktionen:

- Mit DEF kann man sich benutzereigene Funktionen definieren. Dazu gibt man in dem sich öffnenden Fenster die Funktionsdefinition in der Form

  $$< \text{fname} > (< \text{varlist} >) = \ < \text{relterm} >,$$

  wobei <fname> der Name der Funktion ist, <varlist> eine Liste von Bezeichnern, die durch Kommata getrennt sind und <relterm> ein relationenalgebraischer Term in der Notation von RELVIEW. Wir wollen hier auf eine formale Definition eines RELVIEW-Terms verzichten und nur kurz aufführen, woraus solche Terme aufgebaut sind. Der Term kann aufgebaut sein aus den in der Variablenliste aufgeführten Bezeichnern, aus Namen von Relationen, die im Arbeitsspeicher vorhanden sind und aus Funktionssymbolen. Dabei können die Funktionssymbole sowohl in RELVIEW vordefiniert als auch benutzerdefiniert sein. Ein Beispiel zur Definition von Funktionen findet man in 8.3.

- Mit dem Knopf EVAL kann man benutzerdefinierte Funktionen aufrufen. Klickt man EVAL an, so öffnet sich ein Fenster, in dem man einen RELVIEW-Term eingibt, in dem jetzt keine Variablen vorkommen dürfen. Weiter muß man einen Namen eingeben, unter dem die Ergebnisrelation abgelegt werden soll. Dann wird der Term ausgewertet, und das Ergebnis erscheint unter dem angegebenen Namen im Relationenfenster.

- Auch mit dem Knopf ITER kann man benutzerdefinierte Funktionen aufrufen. Klickt man ITER an, so muß man wieder einen Namen für das Ergebnis eingeben, den Namen einer einstelligen Funktion, den Namen einer Relation und eine Zahl zwischen 0 und 5000. Dann wird die Funktion zunächst auf die angegebene Relation und dann immer wieder auf das erhaltene Ergebnis angewandt, bis die Iteration entweder stationär wird, oder so oft wie angegeben durchgeführt wurde.

Als nächstes kommen die Knöpfe zum Ausführen der Grundoperationen. Klickt man diese an, so muß man stets einen Namen für die Ergebnisrelation angeben, und ein oder zwei Argumente, je nachdem , ob die Operation ein- oder zweistellig ist. Dabei sind die Argumente wieder RELVIEW-Terme, ohne Variablen. Es werden dann zunächst die Argumentterme ausgewertet und danach wird die gewählte Operation auf die Resultate angewandt. Dabei haben die Knöpfe folgende Bedeutung:

- $\vee$ bedeutet Vereinigung (zweistellig),

- $\wedge$ bedeutet Durchschnitt (zweistellig),

- – bedeutet Komplement (einstellig),

- ^T bedeutet Transposition (einstellig) und

- $*$ bedeutet Produkt (zweistellig).

Diese Symbole werden auch beim Termaufbau verwendet (außer für die Transposition; dafür verwende statt ^T nur ^).

Die Knöpfe zur Berechnung der Residuen und und des symmetrischen Quotienten tragen die folgenden Symbole:

- S/R für das Linksresiduum,

- R\S für das Rechtsresiduum und

- SYQ für den symmetrischen Quotienten.

Wie bei den zweistelligen Operationen oben, muß man auch hier einen Namen für das Ergebnis und zwei Argumente angeben.

In der nächsten Reihe befinden sich die Knöpfe zur Berechnung von Hüllen.

- Mit SYMM berechnet man die symmetrische Hülle,

- mit REFL die reflexive Hülle

- und mit TRANS die transitive Hülle einer Relation.

Dies sind einstellige Funktionen, bei denen jeweils eine Argument- und eine Resultatrelation angegeben werden muß.

Ganz unten im Fenster finden wir folgende Knöpfe:

- EPSI berechnet zu einer quadratischen Relation der Dimension n $\times$ n die Relation, die alle Teilmengen einer Menge der Mächtigkeit n beschreibt. Dazu muß man wieder als Argument einen Term eingeben, dessen Auswertung eine quadratische Relation liefert.

- PARTF erwartet als Argumente zwei quadratische Relationen. Haben diese die Dimension n $\times$ n bzw. m $\times$ m, so wird als Ergebnis eine Relation geliefert, deren Spalten alle partiellen Funktionen von einer Menge der Mächtigkeit n in eine Menge der Mächtigkeit m beschreiben.

- TOT tut das gleiche wie PARTF, bloß liefert TOT nur die totalen Funktionen.

- Mit INJ kann man zu einem Vektor eine injektive Abbildung erzeugen, die die gleiche Teilmenge beschreibt, wie der Vektor (siehe 8.1.4).

Alle Funktionen, die durch die hier angeführten Knöpfe aufgerufen werden können, können mit den gleichen Bezeichnungen auch zum Aufbau von Termen verwendet werden (siehe 8.3 für Beispiele).

Es gibt in diesem Fenster noch weitere Knöpfe, auf die wir hier nicht weiter eingehen wollen. Diese dienen zum Testen von Relationen auf bestimmte Eigenschaften wie zum Beispiel Symmetrie oder Reflexivität und zur Bereichsdefinition, wir werden sie aber im Rahmen dieser Arbeit nicht benötigen.

8.2.3 Das Directory-Fenster

Das Directory-Fenster zeigt den Inhalt des Arbeitsspeichers von RELVIEW und zwar in drei verschiedenen Feldern.

Im ersten Feld werden die im Arbeitsspeicher enthaltenen Relationen angezeigt. In einer Zeile stehen jeweils der Name der Relation, ihre Dimension und die Angabe `=graph` falls die Relation schon als Graph gezeichnet wurde, und diese Graphdarstellung abgespeichert ist. Durch Anklicken der entsprechenden Zeile kann man nun eine Relation auswählen, die dann im Relationenfenster als Boolesche Matrix dargestellt wird. Ist die Relation auch als Graph abgespeichert, so wird der Graph im Graphfenster angezeigt.

Das zweite Feld zeigt die benutzerdefinierten Funktionen in der Form, wie man sie eingegeben hat. Genauso verhält es sich mit den Bereichsdefinitionen, die im dritten Feld angezeigt werden.

In allen drei Feldern kann man die Liste editieren. Zum Beispiel kann man eine der benutzerdefinierten Funktionen aus dem Arbeitsspeicher löschen oder auch ihre Definition ändern. Für uns ist hier aber nur wichtig, daß wir Relationen auswählen können.

8.2.4 Das Graphfenster

Die Grapheingabe

Um einen Graph einzugeben, klickt man im Graphfenster die rechte Maustaste. Es erscheint dann ein Menü, aus dem man den Punkt NEW auswählt. In dem sich nun öffnenden Fenster gibt man einen Namen für den zu zeichnenden Graphen ein. Dann kann man im Graph-Fenster mithilfe der Maus einen Graph zeichen.

- Um Knoten zu zeichnen, klickt man die linke Maustaste im Graphfenster. Dann wird ein Knoten an der Stelle erzeugt, an der sich der Mauszeiger befindet. Die Knoten werden automatisch durchnumeriert.
- Um eine Kante von Knoten a nach Knoten b zu zeichnen, klickt man zuerst Knoten a mit der linken Maustaste an. Knoten a wird dadurch markiert. Klickt man nun Knoten b an, so wird die Kante gezeichnet.
- Will man einen Knoten löschen, so markiert man ihn zunächst mit der linken Maustaste und löscht ihn dann durch Klicken der mittleren.
- Zum Löschen einer bereits gezeichneten Kante von Knoten a nach Knoten b markiert man diese, indem man zunächst Knoten a und dann Knoten b mit der linken Maustaste anklickt. Dann löscht man die Kante durch Drücken der mittleren Maustaste.
- Will man einen Knoten verschieben, so markiert man ihn mit der linken Maustaste und klickt dann nochmal mit der linken Maustaste die Stelle im Graphfenster an, an die man den Knoten verschieben möchte.

Das Graph-Menü

Den im Graphfenster dargestellten Graphen kann man mithilfe des Graph-Menüs, das erscheint, wenn man die rechte Maustaste klickt, bearbeiten. Folgende Menüpunkte stehen zur Auswahl.

- NEW wurde bereits oben erklärt.
- Mit RENAME kann man den angezeigten Graphen umbenennen.
- Mit GRAPH$->$RELATION wird der Graph in eine Relation umgewandelt, und als Boolesche Matrix im Relationenfenster angezeigt.
- Klickt man PRINT an, so wird der Graph als Postscript-Datei unter dem Namen <graphname>.gr.eps abgespeichert. Dabei ist <graphname> der Name des Graphen.
- Beim Anklicken von MARK_NODES öffnet sich ein Fenster, in dem man einen relationalen Term eingeben kann. Die Auswertung dieses Terms muß einen Vektor ergeben, der genauso viele Zeilen hat wie der dargestelte Graph Knoten. Dann werden im Graph die Knoten markiert, die in der durch den Vektor beschriebenen Teilmenge enthalten sind.

- Der Punkt MARK_EDGES dient zum Markieren von Kanten. Dazu muß in dem sich öffnenden Fenster ein Term eingegeben werden, dessen Auswertung eine Relation liefert, die in der durch den Graph dargestellten Relation enthalten ist. Dann werden die entsprechenden Kanten markiert.

- Mit UNMARK_GRAPH kann man die Markierungen wieder löschen.

- FIT_IN_WINDOW dient dazu, die Größe des Graphen der aktuellen Fenstergröße anzupassen. Der Graph wird dann so vergrössert bzw. verkleinert, daß er genau ins Fenster paßt.

- Beim Anklicken von GRAPH_DRAWING erscheint ein weiteres Menü, aus dem man sich dann einen Graphzeichenalgorithmus auswählen kann. Der Graph wird dann mit dem gewählten Algorithmus im Graphfenster dargestellt. Bisher stehen zum Zeichnen von Graphen fünf Algorithmen zur Verfügung. Dazu gehören die vier hier beschriebenen Algorithmen, die unter folgenden Namen aufrufbar sind:

  Mit LAYER ruft man den Algorithmus aus Kapitel 4 auf.

  FOREST bezeichnet den Algorithmus zum Zeichnen von Bäumen bzw. Wäldern. Diesen Algorithmus kann man nur aufrufen, wenn der Graph auch wirklich ein Wald ist.

  Mit SPRING(slow) kann man den Algorithmus KK aus Kapitel 6 aufrufen.

  Mit SPRING(fast) ruft man Algorithmus FR aus Kapitel 7 auf.

  Schließlich kann man noch CORRESPONDENCE anklicken. Dann wird der Graph folgendermaßen dargestellt. Hat der Graph im Fenster n = |V| Knoten, so hat der dann erzeugte Graph 2n Knoten. Diese werden in zwei Reihen nebeneinander gezeichnet, wobei die Knoten in jeder Reihe von 1 bis n durchnumeriert sind. Gab es im Ausgangsgraphen eine Kante von Knoten a zu Knoten b, so gibt es nun eine Kante von Knoten a aus der linken Reihe zu Knoten b aus der rechten Reihe.

8.2.5 Das Relationenfenster

Die Relationeneingabe

Um eine Relation als Boolesche Matrix einzugeben, klickt man im Relationenfenster die rechte Maustaste. Wie auch bei der Grapheingabe erscheint dann ein Menü, aus dem man zunächst den Punkt NEW auswählt. Dann gibt man einen Namen für die Relation und ihre Dimension an (d.h. die Anzahl der Zeilen und Spalten). Nun erscheint im Relationenfenster eine leere Matrix der gewünschten Dimension, die man mithilfe der Maus editieren kann.

- Um ein Feld in der Matrix schwarz zu färben, klickt man dieses Feld mit der linken Maustaste an.

- Um ein bereits schwarzes Feld wieder zu löschen, klickt man es nochmals mit der linken Maustaste an.
- Klickt man ein weißes Feld mit der mittleren Maustaste an, so werden je nach gewähltem Zeichenmodus (siehe nächsten Abschnitt) entweder alle Felder in der entsprechenden Spalte, der entsprechenden Zeile, der entsprechenden Diagonale von links unten nach rechts oben oder der entsprechenden Diagonale von links oben nach rechts unten schwarz gefärbt.
- Klickt man ein schwarzes Feld mit der mittleren Maustaste an, so wird entsprechend die gesamte Zeile, Spalte oder Diagonale gelöscht.

Das Relation-Menü

Die Relation, die im Relationenfenster dargestellt ist, kann man mithilfe des Relation-Menüs, das nach Klicken der rechten Maustaste im Relationenfenster erscheint, bearbeiten. Dazu stehen folgende Menüpunkte zur Verfügung.

- NEW wurde bereits oben erklärt.
- Mit DELETE kann man die angezeigte Relation aus dem Arbeitsspeicher löschen.
- RENAME dient zum Umbenennen der angezeigten Relation.
- Mit CLEAR kann man alle Felder der Relaton löschen.
- Klickt man RANDOM-FILL an, so kann man eine Zahl zwischen 0 und 100 auswählen, und die Matrix wird dann zu dem angegebenen Prozentsatz zufällig gefärbt.
- Mit EXTRACT_COLUMN(S) kann man sich eine beliebige Anzahl zusammenhängender Spalten aus der angezeigten Matrix herausziehen. Für die so entstandene neue Matrix kann man dann einen neuen Namen eingeben.
- Beim Anklicken von RELATION−>GRAPH wird die Relation im Graphfenster als Graph dargestellt, und zwar in der Form, daß die Knoten alle auf einem Kreis angeordnet werden. Dieser Graph kann dann im Graphfenster, wie in 8.2.4 beschrieben, weiter bearbeitet werden.
- Mit PRINT wird die Matrix unter dem Namen <relname>.rel.eps als Postscript-Datei gespeichert. Dabei ist <relname> der Name der Relation.
- Klickt man DRAW_MODE an, so öffnet sich ein Menü mit den folgenden Punkten: DRAW_MODE \, DRAW_MODE /, DRAW_MODE | und DRAW_MODE −. Damit wählt man den Zeichenmodus aus, der angibt, ob man beim Anklicken eines Feldes der Matrix die Diagonale von links oben nach rechts unten, die Diagonale von links unten nach rechts oben, die Spalte bzw. die Zeile schwarz färbt bzw. löscht.

8.3 Eine Anwendung von RELVIEW

In diesem Abschnitt setzen wir Grundkenntnisse der Verbandstheorie voraus. Die Grundbegriffe der Verbandstheorie findet man zum Beispiel in [22]. Wir betrachten die Schnittvervollständigung einer geordneten Menge, wie sie in [21] vorgestellt wird. Die Methode der Dedekind Schnitte ist eine Art, die reellen Zahlen $\mathbb{R}$ aus den rationalen Zahlen $\mathbb{Q}$ zu konstruieren. Sie wurde verallgemeinert zu einer Methode zur Vervollständigung geordneter Mengen. Wir werden sehen, wie diese Schnittvervollständigung im Kalkül der Relationen behandelt wird und wie wir sie mit RELVIEW berechnen können. Um dies tun zu können, müssen die Symbole der Metasprache $\in$ und $\subseteq$ und der ist-Element-von-Relation und der Mengeninklusionsrelation der Objektsprache unterschieden werden. Im Folgenden verwenden wir in der Objektsprache die Relationen $\varepsilon : \mathrm{X} \leftrightarrow 2^{\mathrm{X}}$ und $\sqsubseteq\colon 2^{\mathrm{X}} \leftrightarrow 2^{\mathrm{X}}$.

8.3.1 Schnittvervollständigung einer geordneten Menge

Sei $\mathrm{R} : \mathrm{X} \leftrightarrow \mathrm{X}$ eine (partielle) Ordnung. Dann heißt das Paar (X,R) eine geordnete Menge. Wenn außerdem jede Teilmenge $\mathrm{Y} \subseteq \mathrm{X}$ ein Supremum und ein Infimum in X besitzt, so heißt (X,R) *vollständiger Verband*.

Die Konstruktion von Vervollständigungen geordneter Mengen mit Dedekind Schnitten sieht nun folgendermaßen aus. Sei (X,R) eine geordnete Menge. Man definiert sich zwei Funktionen auf der Potenzmenge 2^{X} von X, nämlich

$$\mathrm{Ma} : 2^{\mathrm{X}} \to 2^{\mathrm{X}} \qquad \mathrm{Ma(Y)} := \{\mathrm{x} \in \mathrm{X} | \mathrm{x \text{ ist untere Schranke von Y}}\}$$

$$\mathrm{Mi} : 2^{\mathrm{X}} \to 2^{\mathrm{X}} \qquad \mathrm{Mi(Y)} := \{\mathrm{x} \in \mathrm{X} | \mathrm{x \text{ ist obere Schranke von Y}}\}.$$

Ein Element $\mathrm{M} \in 2^{\mathrm{X}}$ heißt *Schnitt* gdw. $\mathrm{Mi(Ma(M))} = \mathrm{M}$. Offenbar ist für jedes $\mathrm{p} \in \mathrm{X}$ die Menge

$$(\mathrm{p}) := \{\mathrm{x} \in \mathrm{X} | \mathrm{R}_{\mathrm{xp}}\}$$

ein Schnitt und heißt der *von p erzeugte Hauptschnitt*. Sei $\mathcal{C}_{\mathrm{X}}$ die Menge der Schnitte von X und $\mathcal{P}_{\mathrm{X}}$ die Menge der Hauptschnitte von X. Seien weiter $\sqsubseteq_{\mathcal{C}}\colon \mathcal{C}_{\mathrm{X}} \leftrightarrow \mathcal{C}_{\mathrm{X}}$ und $\sqsubseteq_{\mathcal{P}}\colon \mathcal{P}_{\mathrm{X}} \leftrightarrow \mathcal{P}_{\mathrm{X}}$ die Restriktionen der Mengeninklusion $\sqsubseteq\colon 2^{\mathrm{X}} \leftrightarrow 2^{\mathrm{X}}$ auf die Mengen der Schnitte bzw. der Hauptschnitte. Dann ist $(\mathcal{C}_{\mathrm{X}}, \sqsubseteq_{\mathcal{C}})$ ein vollständiger Verband mit $(\mathcal{P}_{\mathrm{X}}, \sqsubseteq_{\mathcal{P}})$ als Teilordnung. Außerdem ist die Abbildung

$$\mathrm{e} : \mathrm{X} \to \mathcal{C}_{\mathrm{X}} \quad \mathrm{e(x)} := (\mathrm{x})$$

ein injektiver Ordnungshomomorphismus von R nach $\sqsubseteq_{\mathcal{C}}$. Der Verband $(\mathcal{C}_{\mathrm{X}}, \sqsubseteq_{\mathcal{C}})$ heißt *Schnittvervollständigung* der geordneten Menge (X,R). Ist (X,R) selbst ein Verband, so ist $(\mathcal{P}_{\mathrm{X}}, \sqsubseteq_{\mathcal{P}})$ ein Teilverband von $(\mathcal{C}_{\mathrm{X}}, \sqsubseteq_{\mathcal{C}})$ und e ist ein injektiver Verbandshomomorphismus.

8.3.2 Relationaler Zugang zur Schnittvervollständigung

Wir setzen für den gesamten Abschnitt eine feste geordnete Menge (X,R) voraus. Unser Ziel ist die Konstruktion der Schnittvervollständigung von (X,R)

mit relationenalgebraischen Mitteln. Diese Konstruktion besteht aus mehreren Stufen, die jede für sich mit RELVIEW durchgeführt werden kann.

Eine relationenalgebraische Konstruktion der Vervollständigung $(\mathcal{C}_X, \sqsubseteq_\mathcal{C})$ der geordneten Menge (X,R) und des injektiven Ordnungshomomorphismus e geschieht folgendermaßen.

1. Beschreiben wir eine Teilmenge von X durch einen Vektor $v : X \leftrightarrow 1$ im Sinne von 8.1.4, dann ist $\overline{\overline{R}^T}v$ der Vektor der oberen Schranken von v und $\overline{\overline{R}v}$ der Vektor der unteren Schranken von v. Zum Beispiel kann die erste Behauptung durch folgende Ableitung gezeigt werden. Für alle $x \in X$ gilt

$$\begin{aligned}(\overline{\overline{R}^T}v)_x &\iff \neg(\exists y \in X : \overline{R}^T_{xy} \wedge v_y)\\ &\iff \forall y \in X : \neg(\overline{R}_{yx} \wedge v_y)\\ &\iff \forall y \in X : R_{yx} \vee \neg v_y\\ &\iff \forall y \in X : v_y \rightarrow R_{yx}.\end{aligned}$$

   Gemäß der oben eingeführten Beschreibungen der oberen und unteren Schranken führen wir nun zwei Funktionen auf Mengen von Relationen ein.

$$\begin{aligned}\mathrm{Ma} &: (X \leftrightarrow X) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \qquad \mathrm{Ma}(R,S) := \overline{\overline{R}^T S}\\ \mathrm{Mi} &: (X \leftrightarrow X) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \qquad \mathrm{Mi}(R,S) := \overline{\overline{RS}}\end{aligned}$$

   Sei $S : X \leftrightarrow Y$ eine Relation. In der Terminologie der Booleschen Matrizen kann man sagen, daß für ein Element $y \in Y$ die y-Spalte von Ma(R,S) der Vektor der oberen Schranken bezgl. R der y-Spalte von S ist und daß die y-Spalte von Ma(R,S) der Vektor der unteren Schranken bzgl. R der y-Spalte von S ist.

   Für die Anwendung von RELVIEW kann man die Funktionen Mi und Ma mit dem DEF-Kommando definieren und zwar folgendermaßen:

$$\begin{aligned}\texttt{Ma(R,S)} &= \texttt{-(-R}^\wedge \texttt{ * S)}\\ \texttt{Mi(R,S)} &= \texttt{-(-R * S)}\end{aligned}$$

2. Als nächstes brauchen wir die ist-Element-von-Relation $\varepsilon : X \leftrightarrow 2^X$ und vergleichen sie mit der Relation $\mathrm{Mi}(R, \mathrm{Ma}(R, \varepsilon)) : X \leftrightarrow 2^X$. Durch

$$s : 2^X \leftrightarrow 1 \qquad s := (\mathrm{syq}(\mathrm{Mi}(R, \mathrm{Ma}(R, \varepsilon)), \varepsilon) \cap I)L,$$

   wobei $I : 2^X \leftrightarrow 2^X$ und $L : 2^X \leftrightarrow 1$ die identische bzw. die Universalrelation sind, erhalten wir den Vektor der die Teilmengen von X beschreibt, die Schnitte sind. Zum Beweis der Korrektheit sei $m \in 2^X$. Benutzen wir die Tatsache, daß die identische Relation dem Gleichheitssymbol in der Metasprache entspricht, so erhalten wir

$$s_m \iff ((\mathrm{syq}(\mathrm{Mi}(R, \mathrm{Ma}(R, \varepsilon)), \varepsilon) \cap I)L)_m$$

$$\begin{aligned}
&\iff \exists n \in 2^X : \mathrm{syq}(\mathrm{Mi}(R, \mathrm{Ma}(R, \varepsilon)), \varepsilon)_{mn} \wedge I_{mn} \wedge L_n \\
&\iff \exists n \in 2^X : \mathrm{syq}(\mathrm{Mi}(R, \mathrm{Ma}(R, \varepsilon)), \varepsilon)_{mn} \wedge (m = n) \\
&\iff \mathrm{syq}(\mathrm{Mi}(R, \mathrm{Ma}(R, \varepsilon)), \varepsilon)_{mm} \\
&\iff \forall x \in X : \mathrm{Mi}(R, \mathrm{Ma}(R, \varepsilon))_{xm} \leftrightarrow \varepsilon_{xm}.
\end{aligned}$$

Dies impliziert, daß s_m genau dann gilt, wenn die Teilmenge von X, die durch die m-Spalte der ist-Element-von-Relation $\varepsilon : X \leftrightarrow 2^X$ beschrieben wird, ein Schnitt ist.

Im RELVIEW-System berechnet das Kommando `EPSI` für jede homogene Relation $Q : X \leftrightarrow X$ (die die Menge X repräsentiert) die ist-Element-von-Relation ε. Nun kann man mit diesem Knopf, dem Knopf `SYQ`, den Knöpfen für die Grundoperationen und den oben definierten Funktionen `Ma` und `Mi` den Vektor der Schnitte s leicht berechnen.

3. Da die Schnitte durch Mengeninklusion geordnet sind, betrachten wir nun die Relation $\sqsubseteq : 2^X \leftrightarrow 2^X$. Benutzen wir, daß die Relationen $\sqsubseteq$ und ε den Symbolen $\subseteq$ und $\in$ der Metasprache entsprechen, erhalten wir für beliebige Mengen $m, n \in 2^X$ die Äquivalenz

$$\begin{aligned}
\sqsubseteq_{mn} &\iff m \subseteq n \\
&\iff \forall x \in X : x \in m \to x \in n \\
&\iff \forall x \in X : \varepsilon_{xm} \to \varepsilon_{xn} \\
&\iff (\varepsilon \backslash \varepsilon)_{mn}.
\end{aligned}$$

Dies zeigt, daß die Mengeninklusion als Relation der Objektsprache das Rechtsresiduum $\varepsilon \backslash \varepsilon$ ist. Also kann man sie mit den RELVIEW-Kommandos `EPSI` und `R \ S` berechnen.

4. Wir brauchen nun zunächst eine injektive Abbildung $i : \mathcal{C}_X \to 2^X$, so daß $\mathcal{C}_X$ die durch i gegebene Teilmenge von 2^X ist (im Sinne von 8.1.4). RELVIEW berechnet so eine injektive Abbildung mit der Funktion `INJ`, wobei der oben berechnete Vektor s als Argument eingegeben wird. Benutzen wir die Funktion i, so können wir die Restriktion $\sqsubseteq_{\mathcal{C}} : \mathcal{C}_X \leftrightarrow \mathcal{C}_X$ der Mengeninklusion auf die Mengen der Schnitte wie folgt berechnen:

$$\sqsubseteq_{\mathcal{C}} = i \sqsubseteq i^T$$

Benutzen wir für die Abbildung i die übliche Funktionennotation, so erhalten wir die Gleichung aus der Tatsache, daß für alle Schnitte $m, n \in \mathcal{C}_X$ gilt

$$\begin{aligned}
\sqsubseteq_{\mathcal{C}\,mn} &\iff \exists m_1, n_1 \in 2^X : m_1 = i(m) \wedge n_1 = i(n) \wedge m_1 \sqsubseteq n_1 \\
&\iff \exists m_1 \in 2^X : m_1 = i(m) \wedge (\exists n_1 \in 2^X : m_1 \sqsubseteq n_1 \wedge n_1 = i(n)) \\
&\iff \exists m_1 \in 2^X : m_1 = i(m) \wedge (\sqsubseteq i^T)_{m_1 n} \\
&\iff (i \sqsubseteq i^T)_{mn}.
\end{aligned}$$

5. Als letztes brauchen wir nun noch die relationenalgebraische Beschreibung des injektiven Ordnungshomomorphismus e von X nach $\mathcal{C}_X$. Mit der ist-Element-von-Relation ε aus Schritt (2) und der injektiven Abbildung i aus Schritt (4) erhalten wir

$$\mathrm{e} : \mathrm{X} \to \mathcal{C}_\mathrm{X} \qquad \mathrm{e} = \mathrm{syq}(\mathrm{R}, \varepsilon \mathrm{i}^\mathrm{T}).$$

Zum Nachweis der Korrektheit sei $\mathrm{x} \in \mathrm{X}$ und $\mathrm{m} \in \mathcal{C}$ ein Schitt. Wir benutzen für i wieder Funktionennotation und die Symbole der Metasprache $\in$ und $=$ und erhalten die Äquivalenz

$$\begin{aligned}
\mathrm{syq}(\mathrm{R}, \varepsilon \mathrm{i}^\mathrm{T})_{\mathrm{xm}} &\iff \forall \mathrm{y} \in \mathrm{X} : \mathrm{R}_{\mathrm{yx}} \leftrightarrow (\varepsilon \mathrm{i}^\mathrm{T})_{\mathrm{ym}} \\
&\iff \forall \mathrm{y} \in \mathrm{X} : \mathrm{R}_{\mathrm{yx}} \leftrightarrow (\exists \mathrm{z} \in 2^\mathrm{X} : \mathrm{y} \in \mathrm{z} \wedge \mathrm{i}(\mathrm{m}) = \mathrm{z}) \\
&\iff \forall \mathrm{y} \in \mathrm{X} : \mathrm{R}_{\mathrm{yx}} \leftrightarrow \mathrm{y} \in \mathrm{i}(\mathrm{m}) \\
&\iff \forall \mathrm{y} \in \mathrm{X} : \mathrm{y} \in (\mathrm{x}) \leftrightarrow \mathrm{y} \in \mathrm{i}(\mathrm{m}) \\
&\iff (\mathrm{x}) = \mathrm{i}(\mathrm{m}).
\end{aligned}$$

Also ist die Relation $\mathrm{syq}(\mathrm{R}, \varepsilon \mathrm{i}^\mathrm{T})$ eine Abbildung und bildet ein Element $\mathrm{x} \in \mathrm{X}$ gerade auf den von ihm erzeugten Hauptschnitt $(\mathrm{x}) \in \mathcal{P}_\mathrm{X}$ ab.

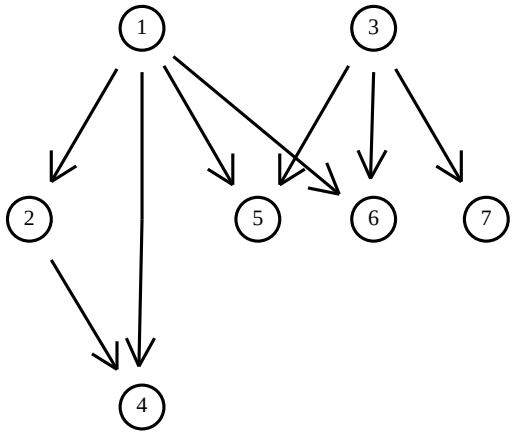
Wir können nun noch die Relation $\mathrm{C} : \mathrm{X} \leftrightarrow \mathcal{C}_\mathrm{X}$ berechnen, deren Spalten genau die Schnitte beschreiben, und zwar gilt für alle $\mathrm{x} \in \mathrm{X}$ und $\mathrm{m} \in \mathcal{C}_\mathrm{X}$

$$\begin{aligned}
(\varepsilon \mathrm{i}^\mathrm{T})_{\mathrm{xm}} &\iff \exists \mathrm{z} \in 2^\mathrm{X} : \varepsilon_{\mathrm{xz}} \wedge (\mathrm{i}(\mathrm{m}) = \mathrm{z}) \\
&\iff \exists \mathrm{z} \in 2^\mathrm{X} : \mathrm{x} \in \mathrm{z} \wedge (\mathrm{i}(\mathrm{m}) = \mathrm{z}) \\
&\iff \mathrm{x} \in \mathrm{i}(\mathrm{m}).
\end{aligned}$$

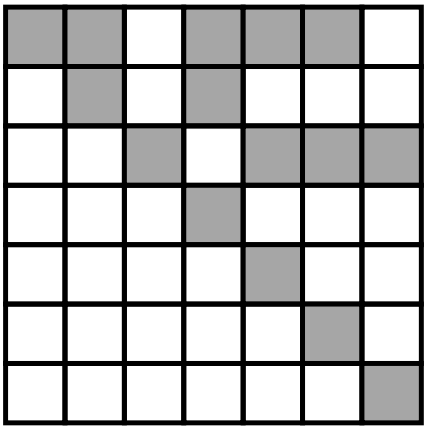
Also ist die Relation, die die Schnitte beschreibt gegeben durch $\mathrm{C} = \varepsilon \mathrm{i}^\mathrm{T}$.

8.3.3 Beispiel

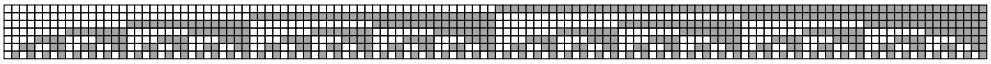
Nachdem wir nun im letzten Abschnitt einen Algorithmus zur Berechnung der Schnittvervollständigung einer geordneten Menge entwickelt haben, wollen wir diesen nun an einem konkreten Beispiel mit RELVIEW ausprobieren. Wir betrachten die Menge $\mathrm{X} = \{1, 2, 3, 4, 5, 6, 7\}$ mit 7 Elementen und der partiellen Ordnung R, dessen irreflexiver Teil durch folgenden Graph gegeben ist (Dieses Bild wurde mit dem Zeichenalgorithmus aus Kapitel 4 gezeichnet, der Pfeile von oben nach unten zeichnet. Es ist eigentlich üblich, die größeren Elemente über den kleineren zu zeichnen, aber hier stehen die kleineren Elemente über den größeren):



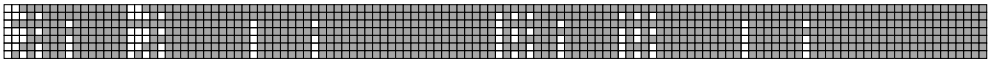
Die Darstellung von R als Boolesche Matrix sieht folgendermaßen aus:



Die nächsten drei Bilder zeigen die Relationen, die wir in Schritt (2) berechnen. Zuerst haben wir die ist-Element-von-Relation ε. Diese hat 7 Zeilen und 128 Spalten. Jede Spalte repräsentiert eine Teilmenge von X. Die erste Zeile steht zum Beispiel für die leere Menge und die achte Zeile für die Menge $\{5, 6, 7\}$.



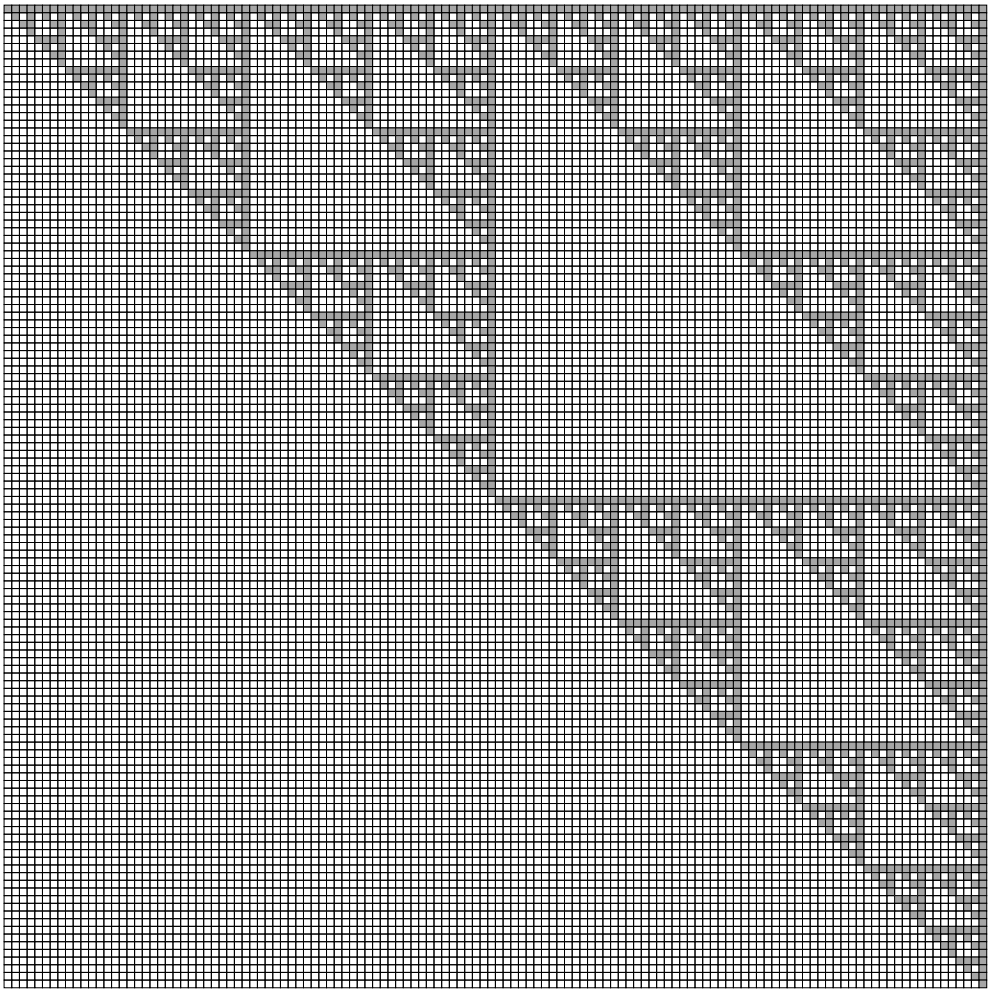
Nun wird zunächst die Funktion `Ma` und dann die Fuktion `Mi` auf diese Relation angewandt, das heißt, man wertet den Term `Mi(R, Ma(R, epsi))` aus. Dabei steht `epsi` hier für die zuvor erhaltene Relation ε. Das Resultat sieht dann folgendermaßen aus:



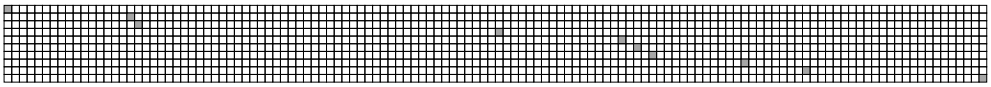
Wie in Schritt (2) beschrieben, vergleichen wir diese Relation nun mit der Relation ε indem wir den symmetrischen Quotienten berechnen, danach mit der identischen Relation schneiden und dann mit der Universalrelation multiplizie-

ren. Wir erhalten den Vektor der Schnitte s. Seine Transponierte sehen wir im folgenden Bild. Wir können feststellen, daß genau 10 der Teilmengen von X Schnitte sind.

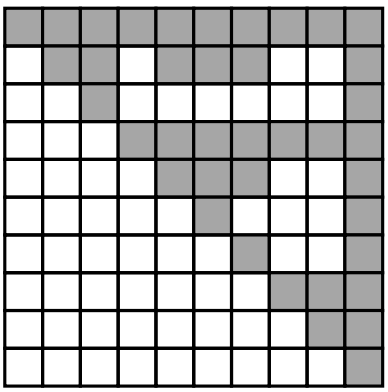
Das nächste Bild zeigt die Mengeninklusion (vgl. Schritt (3)) $\sqsubseteq\colon 2^{\mathrm{X}} \leftrightarrow 2^{\mathrm{X}}$. Zum Beispiel ist das Feld in Zeile 17 und Spalte 18 schwarz, weil $\{3\}$ eine Teilmenge von $\{3, 7\}$ ist.



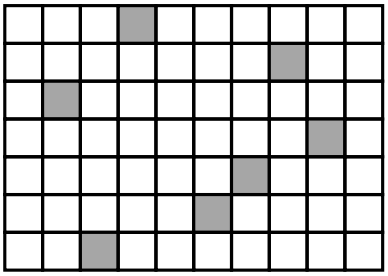
Die Relationen aus Schritt (4) sehen folgendermaßen aus. Das Kommando `INJ` liefert mit dem Argument s die folgende injektive Abbildung $\mathrm{i} : \mathcal{C}_{\mathrm{X}} \to 2^{\mathrm{X}}$:



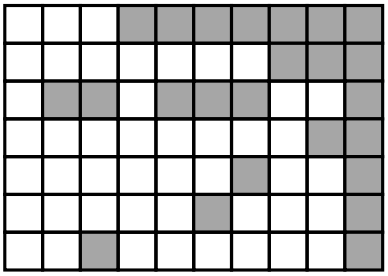
Aus dieser Relation erhalten wir zusammen mit der Mengeninklusion $\sqsubseteq$ die partielle Ordnung $\sqsubseteq_{\mathcal{C}}\colon \mathcal{C}_{\mathrm{X}} \leftrightarrow \mathcal{C}_{\mathrm{X}}$ der Schnittvervollständigung $(\mathcal{C}_{\mathrm{X}}, \sqsubseteq_{\mathcal{C}})$. Die Matrix dazu sehen wir in folgendem Bild:



In Schritt (5) berechnen wir die injektive Abbildung e : X $\rightarrow \mathcal{C}_X$, die wir im nächsten Bild sehen.



Außerdem berechnen wir die Relation C : X $\leftrightarrow \mathcal{C}_X$, deren Spalten genau die Schnitte beschreiben,



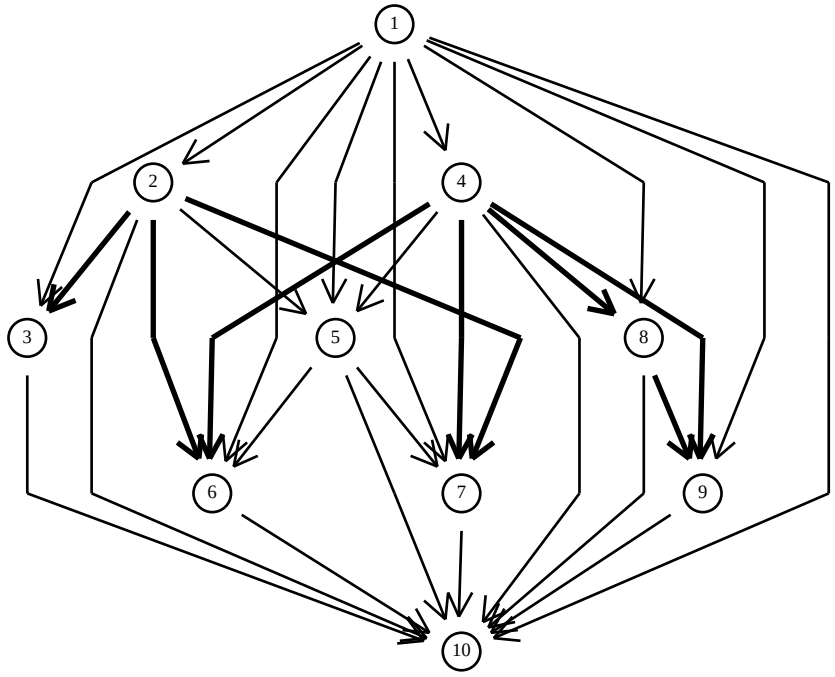
das heißt, $\mathcal{C}_X$ besteht aus den Mengen $\emptyset$, {3}, {3, 7}, {1}, {1, 3}, {1, 3, 6}, {1, 3, 5}, {1, 2}, {1, 2, 4} und X.

Wir wollen nun sehen, wie die ursprüngliche geordnete Menge (X,R) in die Schnittvervollständigung eigebettet ist. Dazu stellen wir $\sqsubset_{\mathcal{C}}$ als Graph dar, und markieren die Kanten der zu (X,R) isomorphen Teilrelation von $\sqsubset_{\mathcal{C}}$. Dazu berechnen wir zunächst aus der injektiven Abb. e : X $\rightarrow \mathcal{C}_X$ den Vektor w, der genau die gleiche Teilmenge wie e beschreibt. Multiplizieren wir diesen Vektor mit seiner Transponierten und schneiden das Ergebnis dann mit $\sqsubset_{\mathcal{C}}$, so erhalten wir genau die Teilrelation, mit der wir dann den Graph markieren müssen (zum Markieren von Graphen siehe 8.2.4). Zur Berechnung der Teilrelation definieren wir uns eine Funktion `mark` durch

$$\texttt{mark}(\texttt{X},\texttt{Y}) = \texttt{X} \;\&\; \texttt{dom}(\texttt{Y}^\wedge) * \texttt{dom}(\texttt{Y}^\wedge)^\wedge .$$

Dabei ist `dom` eine fest eingebaute Funktion, die zu jeder Relation den Vek-

tor liefert, der ihren Definitionsbereich beschreibt (dies kann man auch durch Multiplikation mit der entsprechenden Universalrelation erreichen). Ruft man nun `mark` mit den Argumenten $\sqsubset_{\mathcal{C}}$ und e auf, so erhält man die gewünschte Markierungsrelation. Der so markierte Graph (wieder nur der irreflexive Teil der Relation) ist im folgenden Bild zu sehen (gezeichnet mit dem Algorithmus aus Kapitel 4).



Will man diese Berechnung der Schnittvervollständigung öfter durchführen, das heißt mit verschiedenen Relationen, so ist es sinnvoll, sich Funktionen zu definieren, die sofort die interessanten Ergebnisrelationen wie zum Beispiel $\sqsubset_{\mathcal{C}}$ liefern. Man muß dann nicht mehr alle Zwischenresultate einzeln berechnen. Definiert man sich etwa zusätzlich zu den bereits angegebenen noch die folgenden Funktionen

$$
\begin{aligned}
\texttt{I(X)} &= \texttt{refl(X \& -X)},\\
\texttt{decut(X)} &= \texttt{dom(syq(epsi(X),mi(X,ma(X,epsi(X))))}\\
&\quad \texttt{\& I(epsi(X)\^{} * epsi(X)))},\\
\texttt{cutord(X)} &= \texttt{inj(decut(X)) * (epsi(X)\backslash epsi(X)) * inj(decut(X))\^{}},\\
\texttt{emb(X)} &= \texttt{syq(X,epsi(X) * inj(decut(X))\^{})} \quad \text{und}\\
\texttt{markrel(X)} &= \texttt{mark(cutord(X),emb(X))},
\end{aligned}
$$

so erhält man die Relation $\sqsubseteq_{\mathcal{C}}$ durch die Auswertung des Terms `cutord(R)` und die Markierungsrelation durch Auswertung von `markrel(R)`.

Kapitel 9

Abschließende Bemerkungen

Mit dieser Diplomarbeit wurden nun vier Algorithmen zum Zeichnen von Graphen in das RELVIEW-System eingebaut.

Ein Algorithmus der die Knoten in Schichten auf der Zeichenfläche anordnet und der für beliebige Graphen benutzt werden kann, obwohl er eigentlich für azyklische Graphen entwickelt wurde,

ein Algorithmus, der speziell für das Zeichnen von Bäumen bzw. Wäldern entwickelt wurde und auch nur auf solche anwendbar ist,

zwei Algorithmen, die die Knoten gleichmäßig auf der Zeichenfläche verteilen und wiederum auf beliebige Graphen anwendbar sind.

Diese Algorithmen sind für viele Fälle brauchbar, für viele andere liefern sie jedoch nur unbefriedigende Ergebnisse. Daher soll die Graphausgabe weiterentwickelt werden. Das heißt, es sollen noch mehr Algorithmen eingebaut werden, um auch die Graphen schön zeichnen zu können, für die die bisher vorhandenen Algorithmen keine zufriedenstellenden Bilder liefern. Als mögliche Erweiterungen wären zum Beispiel denkbar

ein Algorithmus zum Zeichnen von planaren Graphen oder

ein Algorithmus zum Zeichnen von Graphen in Orthogonaldarstellung.

Natürlich wird es immer Graphen geben, die so komplex sind, daß man sie überhaupt nicht schön darstellen kann, aber je mehr verschiedene Algorithmen zur Verfügung stehen, desto größer ist die Menge der Graphen, für die sich eine schöne Ausgabe erzeugen läßt.

Literaturverzeichnis

[1] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis, "Algorithms for Drawing Graphs: an Annotated Bibliography", Brown University, Department of Computer Science, 1994.

[2] K. S. Sugiyama, S. Tagawa und M. Toda, "Methods for visual understanding of hierarchical system structures", *IEEE Trans. Syst. Man Cybern.,* Band SMC-11, Nr.2, S. 109–125, Feb. 1981.

[3] V.Chvatal, *Linear Programming*. New York: W. H. Freeman, 1983.

[4] W. H. Cunningham, "A network simplex method", *Mathematical Programming*, Band 11, S. 105–116, 1976.

[5] P. Eades, B. McKay und N.Wormald, "On an edge crossing problem", in *Proc. 9th Australian Computer Science Conf.*, 1986, S. 327–334.

[6] J.Warfield, "Crossing theory and hierarchy mapping", *IEEE Trans. Syst. Man Cybern.,* Band SMC-7, Nr. 7, S. 505–523, 1977.

[7] P.Eades und N. Wormald, "The median heuristic for drawing 2-layers networks", Tech. Rep. 69, Dept. of Computer Science, University of Queensland, 1986.

[8] E. R. Gansner, S. C. North und K.-P. Vo, "DAG—A Program that draws directed graphs", *Software–Practice and Experience*, Band 17, Nr. 1, S. 1047–1062, 1988.

[9] D. E. Knuth, "Optimum binary search trees", Acta Informatika, Band 1, S. 14–25, 1971.

[10] E. R. Gansner, E. Koutsofios, S. C. North und Kiem-Phong Vo, "A technique for drawing directed graphs", *IEEE Trans. Software Eng.*, Band 19, Nr. 3, S. 214–229, 1993.

[11] C. Wetherell und A. Shannon, "Tidy Drawings of Trees", *IEEE Trans. Software Eng.*, Band SE-5, Nr. 5, S. 514–520, 1979.

[12] P. Eades, "A heuristic for graph drawing", *Congr. Numer.* 42 (1984) 149–160.

[13] R. W. Floyd, Algorithm 97: shortest path, *Comm. ACM* 5, (1962), S. 345.

[14] A. Moffat und T. Takaoka, "An all pairs shortest path algorithm with expected running time $O(n^2 \log n)$", *Proc. Conf. Found. Comp. Sci.*, 1985, S. 101–105.

[15] P. M. Spira, "A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log n)$", *SIAM J. Comp.* 2 (1973) S. 28–32.

[16] Tomihisa Kamada und Satoru Kawai, "An algorithm for drawing general undirected graphs", *Information Processing Letters*, Band 31, S. 7–15, 1989.

[17] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis und A. Tuan, "A browser for directed graphs", *Software–Practice and Experience*, Band 11, S. 61–76, 1987.

[18] T. Fruchterman und E. Reingold, "Graph drawing by force-directed placement", *Software–Practice and Experience*, Band 21, Nr. 11, S. 1129–1164, 1991.

[19] G. Schmidt und T. Ströhlein, *Relationen und Graphen*, Springer, Berlin-Heidelberg-New York, 1989.

[20] H. Abold-Thalman, R. Berghammer und G. Schmidt, "Manipulation of Concrete Relations: The RELVIEW-System", Bericht Nr. 8905, Universität der Bundeswehr München, 1989.

[21] R. Berghammer, "Computing the Cut Completion of a Partially Ordered Set - An Example for the Use of the RELVIEW-System", Bericht Nr. 9205, Universität der Bundeswehr München, 1992.

[22] H. Hermes, "Eine Einführung in die Verbandstheorie", 2. Auflage, Springer, Berlin-Heidelberg-New York, 1989.