Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Informatik I
Effiziente Algorithmen und wissensbasierte Systeme

Masterarbeit

Zeichnen von Rechengraphen

Julian Schuhmann

Eingereicht am 23. September 2013

Betreuer:
Prof. Dr. Alexander Wolff
Dipl.-Inf. Martin Fink

Inhaltsverzeichnis

1. Einleitung

Die Verbesserung der Unterrichtsqualität ist nicht erst seit der PISA-Studie ein großes Ziel von Lehrern und Bildungsbeauftragten. Im Bereich der Mathematik können dazu seit einiger Zeit sogenannte Rechengraphen eingesetzt werden. Sie wurden erstmals im Jahr 1999 von Martin Hennecke vorgestellt [Hen99] und dienen dazu, häufige Fehlermuster von Schülern beim Bearbeiten von Mathematikaufgaben zu finden. Dadurch können Lehrer leichter erkennen, wo für ihre Schüler die Probleme liegen, und ihren Unterricht dann entsprechend anpassen.

Um einen solchen Rechengraph zu erstellen, werden zunächst Schüler ausgewählt, die als Versuchsgruppe dienen. Die Größe dieser Gruppe kann je nach Ziel der Untersuchung stark variieren. Zum Anpassen der Unterrichtsplanung kann ein Lehrer als Versuchsgruppe eine Schulklasse benutzen, für allgemeinere Untersuchungen kann die Gruppe aber auch aus mehreren Tausend Schülern bestehen. Diese Gruppe bekommt dann eine Aufgabe gestellt und versucht sie zu lösen. Anschließend werden die Ergebnisse analysiert, wozu die Zwischenergebnisse aller Schüler gesammelt werden. Sie werden später beim Zeichnen des Rechengraphen als Rechtecke dargestellt und als Knoten bezeichnet. Sie erhalten als Beschriftung das entsprechende Zwischenergebnis, weshalb die Knoten unterschiedlich groß sein können. Die von den Schülern angewendeten Rechenschritte werden als Pfeile zwischen zwei Knoten dargestellt. Diese Pfeile bezeichnet man als Kanten. Sie werden dann noch mit einer Zahl versehen, die angibt, wie viele Schüler diesen Rechenschritt gewählt haben. Diese Zahl bezeichnet man als das Gewicht der Kante.

Ein einfaches Beispiel für einen Rechengraph ist in Abbildung 1.1 zu sehen. Der Startknoten, welcher der gestellten Aufgabe entspricht, wird ganz links gezeichnet (hier gelb eingefärbt). Die anderen Knoten entsprechen den Zwischenergebnissen. Wie man sieht, sind die Knoten unterschiedlich groß, je nachdem wie viel Platz die Beschriftung benötigt. Die Kanten zeigen an, dass ein Schüler den Rechenschritt von einem Zwischenergebnis zum Nächsten angewendet hat. Die Zahlen an den Kanten geben ihr Gewicht an.

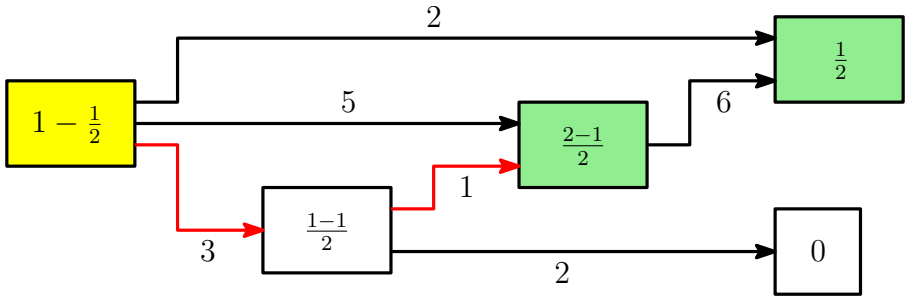


Abb. 1.1.: Ein einfacher Rechengraph

Damit enthält der Rechengraph alle wichtigen Informationen, die für den Betrachter von Nutzen sein können. Um die Lesbarkeit zu verbessern, können jetzt noch verschiedene Anpassungen vorgenommen werden. Dazu gehört zum Beispiel das Einfärben von Knoten und Kanten. Knoten, die ein korrektes Zwischenergebnis repräsentieren, können grün eingefärbt werden. Ähnlich können Kanten, die einen fehlerhaften Rechenschritt repräsentieren, rot gezeichnet werden. Eine weitere, im Beispielgraph nicht enthaltene Möglichkeit wäre, dass die Dicke einer Kante von ihrem Gewicht abhängt. So sind wichtige Kanten auffälliger und leichter zu erkennen.

Für einfache Aufgaben mit wenigen Zwischenergebnissen und einer kleinen Gruppe von Schülern ist das Erstellen von Rechengraphen per Hand möglich, weil es nur wenige Knoten gibt. Bei komplizierteren Aufgaben und einer großen Gruppe von Schülern wird dies jedoch zunehmend schwerer, da die Anzahl der Knoten schnell sehr groß wird. Um auch aus großen Graphen noch die wichtigsten Informationen und die häufigsten Fehlerquellen herauslesen zu können, sind Methoden nötig, die automatisch einen Rechengraph erstellen. Dabei ist eine Fläche mit festgelegter Größe gegeben, auf die der Rechengraph gezeichnet werden soll, wobei diese meist deutlich kleiner ist als der Platz, der für den ganzen Rechengraph benötigt würde. Deshalb spielt die Auswahl der Knoten, welche gezeichnet werden sollen, eine wichtige Rolle.

Im Allgemeinen ist das Zeichnen eines Teilgraphen mit einer vorgegebenen Platzbeschränkung ein recht neues Problem, das bisher kaum untersucht wurde. Hierzu leistet diese Arbeit einen Beitrag, indem ein Verfahren vorgestellt wird, das einen Teilgraph auswählt und zeichnet, je nachdem wie wichtig die unterschiedlichen Knoten sind. Das entwickelte Verfahren lässt sich grob in zwei Schritte einteilen:

- **Welche Knoten werden gezeichnet?** Zuerst müssen die Knoten ausgewählt werden, die dem Betrachter die meisten Informationen über häufige Fehlerquellen liefern.
- **Wo werden die Knoten gezeichnet?** Danach müssen die Zeichenpositionen für die Knoten festgelegt werden. Dabei muss darauf geachtet werden, dass der Graph übersichtlich ist und der Betrachter somit schnell die wichtigsten Informationen herauslesen kann. Hier sollen vor allem Kantenkreuzungen verhindert werden.

Das automatische Zeichnen von Rechengraphen ist NP-schwer (Kapitel 2.2). Es ist also unwahrscheinlich, dass es einen schnellen Algorithmus gibt, der optimale Rechengraphen erstellen kann. Für andere NP-schwere Probleme können jedoch mit Hilfe von gemischt ganzzahligen Programmen sehr gute Lösungen gefunden werden. Dieser Ansatz wurde auch für das Zeichnen von Rechengraphen verwendet, es stellte sich jedoch heraus, dass das Verfahren für große Graphen deutlich zu langsam und somit nicht praktikabel ist.

In dieser Arbeit wird deshalb ein heuristischer Ansatz untersucht, der auf dem hierarchischen Zeichnen von Graphen basiert. Dieser Zeichenstil, der in Kapitel 3 beschrieben wird, wurde bereits 1981 von Sugiyama et al. [STT81] vorgestellt. Zuvor wird in Kapitel 2 noch einmal genauer auf die Problemstellung und die Anforderungen der Lösung eingegangen. In Kapitel 4 werden dann die Anpassungen erläutert, welche gemacht wurden, um das Verfahren von Sugiyama für das Zeichnen von Rechengraphen anzupassen

und zu optimieren. Anschließend werden in Kapitel 5 die Ergebnisse der Heuristik mit unterschiedlichen Einstellungen diskutiert, bevor in Kapitel 6 noch auf zukünftige Verbesserungen eingegangen wird, die im Rahmen dieser Arbeit nicht untersucht wurden.

2. Problemstellung und Anforderungen an die Lösung

2.1. Grundlegende Definitionen

Da in dieser Arbeit ein Graph gezeichnet werden soll, muss zunächst definiert werden, was ein Graph ist. Eine gute Einführung in die Graphentheorie bietet ein Buch von Noltemeier und Krumke [KN09], die in dieser Arbeit benötigten Definitionen und Notationen werden zudem jetzt erläutert. Ein *Graph* ist ein Paar $G = (V, E)$, wobei V eine Menge von Objekten ist, die man, wie schon in der Einleitung erwähnt, als *Knoten* bezeichnet. Bei Rechengraphen sind das die Zwischenergebnisse der Schüler. Die Menge E besteht aus Paaren von Knoten, welche man als *Kanten* bezeichnet. Jede Kante verbindet immer genau zwei Knoten. Eine Kante, die die Knoten u und v verbindet, wird mit (u, v) bezeichnet. In Rechengraphen entsprechen die Kanten den Rechenschritten, die die Schüler angewendet haben. Ein Graph $G = (V, E)$ heißt *bipartit*, wenn V so in zwei Mengen V_1 und V_2 aufgeteilt werden kann, dass jede Kante immer einen Knoten aus V_1 mit einem Knoten aus V_2 verbindet. Der Graph wird dann auch als $G = (V_1 \cup V_2, E)$ bezeichnet.

Ein *gerichteter* Graph enthält in der Kantenmenge geordnete Paare von Knoten. Das heißt eine Kante verknüpft nicht nur zwei Knoten, sondern sie legt auch eine Richtung der Verknüpfung fest. Das ist bei der Arbeit mit Rechengraphen wichtig, da es von Bedeutung ist, in welcher Reihenfolge die Schüler die Zwischenergebnisse berechnen. Bei einer gerichteten Kante $e = (u, v)$ wird u als *Startknoten* und v als *Zielknoten* der Kante bezeichnet. Ein Knoten v und eine Kante e heißen *inzident*, wenn v einer der beiden Knoten ist, die e miteinander verbindet. Jeder Knoten v hat eine Menge von *ausgehenden Kanten*, welche alle Kanten enthält, die v als Startknoten besitzen. Ebenso hat v ein Menge von *eingehenden Kanten*, die v als Zielknoten besitzen. Ein Knoten in einem gerichteten Graph heißt *Quelle*, wenn er nur ausgehende Kanten hat. In Rechengraphen gibt es immer genau eine Quelle, welche der gestellten Aufgabe entspricht. Diese Quelle wird auch als Startknoten des Graphen bezeichnet.

In einem gerichteten Graph hat jeder Knoten eine Menge von Vorgängern und Nachfolgern. Vorgänger eines Knotens v sind alle Knoten u, für die eine Kante (u, v) existiert, und Nachfolger von v sind alle Knoten w, für die eine Kante (v, w) existiert. Die Menge der Vorgänger eines Knotens v wird mit $N^-(v)$ bezeichnet, die Menge seiner Nachfolger wird mit $N^+(v)$ bezeichnet. Als *Nachbarn* eines Knoten v bezeichnet man alle Vorgänger und Nachfolger von v, die Menge aller Nachbarn wird mit $N(v)$ abgekürzt. Ein *Weg* ist eine Liste von Knoten $(v_1, v_2, \ldots, v_k)$, die durch die Kanten $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ verbunden sind. Ein *Kreis* ist ein Weg $(v_1, \ldots, v_n)$ mit $v_1 = v_n$. Ein Graph heißt *azyklisch*,

wenn er keinen Kreis enthält. Ein Knoten v ist von einem Knoten u *erreichbar*, wenn es einen Weg von u nach v gibt. Ein ungerichteter Graph G heißt *zusammenhängend* wenn jeder Knoten von jedem anderen Knoten erreichbar ist. Ist G nicht zusammenhängend, so zerfällt er in seine maximal zusammenhängende Teile, die man als *Zusammenhangskomponenten* bezeichnet. Ein gerichteter Graph G heißt *schwach zusammenhängend*, wenn der zugrunde liegende ungerichtete Graph zusammenhängend ist. Ist G nicht schwach zusammenhängend, zerfällt er in seine *schwachen Zusammenhangskomponenten*, die mit den Zusammenhangskomponenten im entsprechenden ungerichteten Graph übereinstimmen.

2.2. Problemstellung

Die Eingabe des Rechengraph-Problems ist ein gerichteter Graph, in dem für jeden Knoten die Höhe und Breite sowie eine Beschriftung angegeben ist, welche dem zugehörigen Zwischenergebnis entspricht. Die Kanten haben jeweils ein Gewicht, das die Anzahl von Schülern angibt, die den entsprechenden Rechenschritt ausgeführt haben. Beim Einlesen der Eingabe wird auch den Knoten ein Gewicht zugeordnet, das gleich der Summe des Gewichts aller eingehenden Kanten ist. Das Gewicht eines Knotens entspricht also der Anzahl von Schülern, die mit diesem Zwischenergebnis gerechnet haben. Dieses Gewicht bleibt konstant, auch wenn im Verlauf des Algorithmus eingehende Kanten gelöscht werden. Ein weiterer Teil der Eingabe ist die Höhe und Breite der Fläche, die zum Zeichnen zur Verfügung steht.

Das Hauptproblem beim Zeichnen von Rechengraphen ist, dass die Graphen meist sehr groß sind, während der verfügbare Platz auf der Zeichenfläche begrenzt ist. Die Schwierigkeit besteht dann darin, auszuwählen, welche Knoten gezeichnet werden sollen und welche vernachlässigt werden können, weil sie für den Betrachter keine entscheidenden Informationen liefern. Die Wichtigkeit eines Knotens hängt dabei vor allem von seinem Gewicht, aber auch von seiner Größe ab. Das Ziel ist nun, einen Rechengraph zu zeichnen, der auf die Zeichenfläche passt und möglichst hohes Knotengewicht hat. Das ist ein Optimierungsproblem. Die formale Definition des entsprechenden Entscheidungsproblems lautet folgendermaßen:

Definition 1 (Rechengraph$(V, E, v_s, b, h, w, d_{\min}, B, H, W)$)**.** Sei Folgendes gegeben:

- V : Menge von Knoten
- $E \subseteq V \times V$: Menge von gerichteten Kanten
- $v_s \in V$: Startknoten, so dass für alle $v \in V$ gilt: v ist von v_s aus erreichbar
- $b\colon V \to \mathbb{N}$: Breite eines Knotens
- $h\colon V \to \mathbb{N}$: Höhe eines Knotens
- $w\colon V \to \mathbb{R}^+$: Gewicht eines Knotens
- $d_{\min} \in \mathbb{N}$: Mindestabstand zwischen zwei Knoten
- $B \in \mathbb{N}$: Breite der Zeichenfläche
- $H \in \mathbb{N}$: Höhe der Zeichenfläche
- $W \in \mathbb{R}^+$: Das minimale Gewicht des Graphen ohne den Startknoten

Dann gilt: Die Rechengraph-Instanz $(V, E, v_s, b, h, w, d_{\min}, B, H, W)$ ist erfüllbar, genau dann wenn es einen Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$ sowie eine Zeichnung Z von G' gibt, die folgende Eigenschaften erfüllen:

- $v_s \in V'$
- In G' sind alle Knoten von v_s aus erreichbar
- alle Knoten liegen in Z innerhalb der Zeichenfläche
- alle Knoten sind in Z mindestens $d_{\min}$ voneinander entfernt
- es gilt: $\sum_{v' \in (V' \setminus \{v_s\})} w(v') \geq W$

In Satz 3 wird gezeigt, dass dieses Problem NP-schwer ist, indem das NP-schwere Teilsummen-Problem [GJ79] auf eine vereinfachte Version des Rechengraph-Problems, in dem alle Knoten die gleiche Höhe haben, reduziert wird. Das Teilsummen-Problem ist folgendermaßen definiert:

Definition 2 (Teilsumme(S, T))**.** Sei $S = \{s_1, s_2, \ldots, s_n\} \subset \mathbb{N}$ (Mehrfachelemente sind möglich) und sei $T \in \mathbb{N}$. Dann gilt: Die Teilsummen-Instanz (S, T) ist erfüllbar, genau dann wenn es eine Menge $S' \subseteq S$ gibt mit $\sum_{s' \in S'} s' = T$.

Satz 3. *Rechengraph ist NP-schwer.*

Beweis. Sei eine Teilsummen-Instanz $(S = \{s_1, s_2, \ldots, s_n\}, T)$ gegeben. Dann lässt sich daraus eine Rechengraph-Instanz $(V, E, v_s, b, h, w, d_{\min}, B, H, W)$ wie folgt erzeugen. Sei $V = \{v_1, v_2, \ldots, v_n, v_s\}$ die Knotenmenge und seien $b(v_i) = w(v_i) = s_i \cdot n$ für $1 \leq i \leq n$ sowie $b(v_s) = w(v_s) = n$. Sei $d_{\min} = 1$, sei $B = (T \cdot n) + 2 \cdot n$ und sei $W = T \cdot n$. Sei zudem $H = 3/2$ und $h(v_s) = h(v_i) = 1$ für $1 \leq i \leq n$. Die Festlegungen der Höhen haben zur Folge, dass über und unter den Knoten noch Platz ist, um Kanten zu zeichnen. Trotzdem müssen alle Knoten nebeneinander angeordnet sein, weil in einer zulässigen Zeichnung zwei Knoten nicht übereinander auf die Zeichenfläche passen. Damit ist für die Auswahl nur die Breite der Knoten relevant. Sei außerdem noch $E = \{(v_s, v_i)\}$ für $1 \leq i \leq n$, das heißt es existiert eine Kante vom Startknoten zu jedem anderen Knoten im Graph. Damit sind alle Knoten vom Startknoten aus erreichbar und der Graph ist azyklisch. Eine so erstellte Rechengraph-Instanz ist dann äquivalent zu der gegebenen Teilsummen-Instanz:

„⇒“: Angenommen die Teilsummen-Instanz (S, T) ist erfüllbar. Dann existiert eine Menge $S' \subseteq S$ mit $(\sum_{s' \in S'} s') = T$. Sei nun $V' = \{v_i \mid s_i \in S'\} \cup \{v_s\}$. Dann gilt:

$$\begin{aligned}
\left(\sum_{s' \in S'} s'\right) = T &\Rightarrow \left(\sum_{v' \in V' \setminus \{v_s\}} \frac{b(v')}{n}\right) = T = \frac{B}{n} - 2 \\
&\Rightarrow \left(\sum_{v' \in V' \setminus \{v_s\}} b(v')\right) = B - 2 \cdot n \\
&\Rightarrow \left(\sum_{v' \in V'} b(v')\right) = B - n
\end{aligned}$$

$$\Rightarrow \left(\sum_{v' \in V'} b(v') \right) + n \leq B \tag{2.1}$$

Außerdem gilt:

$$\left(\sum_{s' \in S'} s' \right) = T \Rightarrow \left(\sum_{v' \in V' \setminus \{v_s\}} \frac{w(v')}{n} \right) = \frac{W}{n}$$
$$\Rightarrow \left(\sum_{v' \in V' \setminus \{v_s\}} w(v') \right) \geq W \tag{2.2}$$

Aus (2.1) folgt, dass alle Knoten aus $|V'|$ so in der Zeichenfläche platziert werden können, dass sie mindestens $d_{\min} = 1$ voneinander entfernt sind, da $|V'| \leq n + 1$ gilt und somit höchstens n Zwischenräume benötigt werden. Aus (2.2) wiederum folgt, dass das Mindestgewicht für den Rechengraph erfüllt ist. Sei nun noch $E' = \{(v_s, v_i) \mid v_i \in V'\}$. Dann sind alle Knoten in V' von v_s aus erreichbar und somit sind alle Bedingungen erfüllt, damit die Rechengraph-Instanz $(V, E, v_s, b, h, w, d_{\min}, B, H, W)$ erfüllbar ist.

„⇐“: Angenommen die Rechengraph-Instanz $(V, E, v_s, b, h, w, d_{\min}, B, H, W)$ ist erfüllbar. Dann existiert ein $V' \subseteq V$ mit $v_s \in V'$ und $(\sum_{v' \in V'} b(v')) + (|V'| - 1) \leq B$. Sei $S' = \{s_i \mid v_i \in V' \setminus \{v_s\}\}$. Somit gilt:

$$\left(\sum_{v' \in V'} b(v') \right) + (|V'| - 1) \leq B \Rightarrow \left(\sum_{v' \in V'} b(v') \right) \leq B - |V'| + 1$$
$$\Rightarrow \left(\sum_{v' \in V'} b(v') \right) \leq (T \cdot n) + 2 \cdot n - |V'| + 1$$
$$\Rightarrow \left(\sum_{s' \in S'} s' \cdot n \right) + b(v_s) \leq (T \cdot n) + 2 \cdot n - |S'|$$
$$\Rightarrow \left(\sum_{s' \in S'} s' \cdot n \right) \leq (T \cdot n) + n - |S'|$$
$$\Rightarrow \left(\sum_{s' \in S'} s' \right) \leq T + 1 - \frac{|S'|}{n} \tag{2.3}$$

Außerdem gilt $(\sum_{v' \in V' \setminus \{v_s\}} w(v')) \geq W$. Dann folgt:

$$\left(\sum_{v' \in V' \setminus \{v_s\}} w(v') \right) \geq W \Rightarrow \left(\sum_{v' \in V' \setminus \{v_s\}} w(v') \right) \geq T \cdot n$$
$$\Rightarrow \left(\sum_{s' \in S'} s' \cdot n \right) \geq T \cdot n$$

$$\Rightarrow \left(\sum_{s' \in S'} s' \right) \geq T \tag{2.4}$$

Aus (2.3) und (2.4) folgt:

$$T \leq \left(\sum_{s' \in S'} s' \right) \leq T + 1 - \frac{|S'|}{n}$$

$$\Rightarrow \left(\sum_{s' \in S'} s' \right) = T \qquad \text{(für } |S'| \neq 0\text{, da } s' \text{ und } T \text{ ganzzahlig sind)}$$

Daraus folgt, dass die Teilsummen-Instanz (S, T) erfüllbar ist. Folglich sind die die beiden Probleme äquivalent und somit ist Rechengraph NP-schwer. □

Aufgrund dieses Ergebnisses macht es Sinn, für das Zeichnen von Rechengraphen Heuristiken einzusetzen, um eine möglichst gute Lösung zu finden.

2.3. Anforderungen an die Lösung

Die Ausgabe des Algorithmus soll eine Reihe von Anforderungen erfüllen, um dem Betrachter einen möglichst guten Überblick über den Rechengraph zu geben. Das Wichtigste ist, wie bereits erwähnt, dass das Gewicht der gezeichneten Knoten möglichst groß ist. Je schwerer die dargestellten Knoten sind, desto mehr Information steckt in der Zeichnung. Dabei dürfen sich die Knoten aber nicht gegenseitig überlappen und es soll einen Mindestabstand zwischen zwei Knoten geben, damit die Knoten klar unterscheidbar sind.

Außerdem soll es dem Betrachter möglichst leicht gemacht werden, den Rechenwegen der Schüler zu folgen. Das wird dadurch erreicht, dass die Kanten immer möglichst von links nach rechts gezeichnet werden (siehe auch Abb. 1.1), wobei das nicht möglich ist, wenn der Rechengraph Kreise enthält. In diesem Fall muss entschieden werden, welche Kanten vorwärts, also von links nach rechts, verlaufen sollen. Hier muss auf zwei Kriterien geachtet werden, denn einerseits sollen möglichst viele Kanten vorwärts verlaufen, andererseits sollen die wichtigen Kanten, also Kanten mit hohem Gewicht, vorwärts gezeichnet werden. Ursprünglich sollten auch die rückwärts verlaufenden Kanten in der Zeichnung enthalten sein. Da sich jedoch herausstellte, dass es meist nur wenige solcher Kanten gibt, wurde die Zeit in wichtigere Dinge investiert und es werden keine Kanten gezeichnet, die von rechts nach links verlaufen würden.

Der nächste wichtige Punkt ist das Verhindern von Kantenkreuzungen. Je mehr es davon gibt, desto schwieriger ist es für den Betrachter, den Kanten zu folgen. In kleinen Graphen können sie zum Teil komplett verhindert werden, in großen Rechengraphen lassen sie sich aber wahrscheinlich nicht vermeiden. Hier soll darauf geachtet werden, dass es möglichst wenige Kreuzungen gibt. Es können aber auch mehr Kreuzungen in Kauf genommen werden, wenn dafür die wichtigen Kanten ohne Kreuzungen auskommen. Hier muss abgewogen werden, welches Ziel wichtiger ist.

Ursprünglich sollten die Kanten im orthogonalen Stil gezeichnet werden, das heißt sie sind nur aus horizontalen und vertikalen Segmenten zusammengesetzt. Das hat den Vorteil, dass sich Kanten immer im 90 Grad Winkel schneiden. Dadurch entstehen jedoch viele Knicke, was es schwerer macht, einer Kante zu folgen. Im Verlauf der Arbeit wurde deshalb noch ein anderer Kantenstil interessant, in dem die Kanten als Kurven dargestellt werden und somit keine Knicke mehr in der Zeichnung enthalten sind.

Ein weiterer Punkt ist das Beschriften der Kanten mit ihren Gewichten. Diese sollen so gesetzt werden, dass sie sich eindeutig ihren Kanten zuordnen lassen. Im orthogonalen Fall macht es Sinn, die Beschriftungen an den horizontalen Teilen der Kanten anzubringen. Das Beschriften der Kanten wurde jedoch in dieser Arbeit nicht näher untersucht, weshalb die Beispielausgaben keine Kantengewichte aufweisen. Trotzdem sind wichtige Kanten erkennbar, weil die Breite einer Kante von ihrem Gewicht abhängt.

3. Sugiyama-Framework

Der Algorithmus, der im Rahmen dieser Arbeit entwickelt wurde, basiert auf dem Sugiyama-Framework, welches 1981 von Sugiyama et al. [STT81] vorgestellt wurde. Dieses Verfahren hat das Ziel, einen gegebenen Graph so zu zeichnen, dass die Hierarchie möglichst gut wiedergegeben wird, das heißt es sollen so viele Kanten wie möglich in die gleiche Richtung zeigen. Dazu werden die Knoten horizontalen Linien, so genannten *Lagen*, zugeordnet. Diese Zuordnung wird so vorgenommen, dass alle Kanten immer Knoten aus verschiedenen Lagen verbinden und alle Kanten in die gleiche Richtung zeigen (üblicherweise nach oben). Die Anzahl der Lagen soll dabei klein gehalten werden, um den Graph kompakt zeichnen zu können. Außerdem sollen die Kanten möglichst kurz sein und die Anzahl der Kantenkreuzungen soll gering gehalten werden. Diese Kriterien ähneln bereits den in Kapitel 2.3 formulierten Anforderungen an das Zeichnen von Rechengraphen. Damit liegt die Vermutung nahe, dass diese Heuristik auch beim Zeichnen von Rechengraphen gute Ergebnisse liefert.

Das Sugiyama-Framework besteht aus vier Teilschritten. Zuerst müssen aus dem zu zeichnenden Graph eventuell vorhandene Kreise entfernt werden. Dazu wird eine möglichst kleine Menge von Kanten gesucht, so dass der Graph kreisfrei ist, wenn die Richtung dieser Kanten geändert wird. Wie man eine solche Kantenmenge findet, wird in Kapitel 3.1 erläutert. Das Entfernen von Kreisen ist nötig, damit im zweiten Schritt die Lagenzuordnung vorgenommen werden kann. Die Lagen werden hier so zugeordnet, dass alle Kanten in die gleiche Richtung zeigen, was bei vorhandenen Kreisen nicht möglich wäre. Wie die Lagenzuordnung genau funktioniert, wird in Kapitel 3.2 anhand von drei verschiedenen Algorithmen gezeigt.

Zu diesem Zeitpunkt kann man den Graph bereits hierarchisch zeichnen, indem man die Positionen der Knoten innerhalb der Lagen beliebig festlegt. Diese Zeichnung ist aber wahrscheinlich nur schlecht lesbar und wird deshalb in den nächsten Schritten noch verbessert. Dazu wird die Anzahl von Kantenkreuzungen minimiert (Kap. 3.3) und die konkreten Positionen der Knoten werden so festgelegt, dass der Graph möglichst übersichtlich ist (Kap. 3.4). In diesem Schritt werden dann auch die anfangs umgedrehten Kanten wieder so gezeichnet, dass sie in ihre ursprüngliche Richtung zeigen.

3.1. Entfernen von Kreisen

Der erste Schritt beim hierarchischen Zeichnen von Graphen besteht darin, dass eventuell im Graph vorhandene Kreise entfernt werden müssen. Das ist nötig, da die im zweiten Schritt eingesetzten Algorithmen darauf ausgelegt sind, eine Lagenzuordnung für die Knoten zu finden, so dass für alle Kanten der Startknoten einer niedrigeren Lage

zugeordnet ist als der Zielknoten. Wenn der Graph hier Kreise enthält, ist das nicht möglich. Deshalb wird der Graph nun kreisfrei gemacht, indem ein sogenanntes *Feedback-Set* gesucht wird. Das ist eine Menge F von Kanten, so dass der Graph durch das Umdrehen der Kanten in F kreisfrei wird.

Definition 4 (Feedback-Set)**.** Sei ein gerichteter Graph $G = (V, E)$ gegeben. Eine Menge $F \subseteq E$ heißt Feedback-Set, genau dann wenn

$$G' = (V, (E \backslash \{F\}) \cup \{(u, v) \mid (v, u) \in F\}) \quad \text{azyklisch ist.}$$

Hat man ein Feedback-Set gefunden, betrachtet man im Folgenden nur noch den Graph, der die umgedrehten Kanten enthält. Erst am Ende, wenn der Graph tatsächlich gezeichnet wird, erhalten die umgedrehten Kanten wieder ihre ursprüngliche Richtung. Das heißt die Größe des Feedback-Set entspricht der Anzahl von Kanten, die in der finalen Zeichnung in die „falsche“ Richtung zeigen. Da die Zahl dieser Kanten möglichst gering gehalten werden soll, ist es nötig, ein möglichst kleines Feedback-Set des Graphen zu finden. Dieses Problem ist für gerichtete Graphen NP-vollständig [Kar72].

Ein ähnliches Problem ist das Finden eines *Feedback-Arc-Set*, bei dem die enthaltenen Kanten nicht umgedreht, sondern entfernt werden, um den Graph kreisfrei zu machen. Auch dieses Problem ist für gerichtete Graphen NP-vollständig [Kar72].

Definition 5 (Feedback-Arc-Set)**.** Sei ein gerichteter Graph $G = (V, E)$ gegeben. Eine Menge $F \subseteq E$ heißt Feedback-Arc-Set, genau dann wenn

$$G' = (V, E \backslash \{F\}) \quad \text{azyklisch ist.}$$

Im optimalen Fall sind die beiden Probleme zudem äquivalent, das heißt ein Feedback-Set, aus dem man keine Kante entfernen kann, ist auch ein minimales Feedback-Arc-Set und umgekehrt [BETT98]. Es genügt also, für den Eingabegraph ein Feedback-Arc-Set zu finden. Zur Lösung dieses Problems existiert eine Heuristik, die für einen Graph $G = (V, E)$, in dem es keine Kreise mit Länge zwei gibt, in $O(V + E)$ Zeit ein Feedback-Arc-Set F findet, für das $|F| \leq |E|/2 - |V|/6$ gilt [BETT98].

Eine andere Möglichkeit ist es, ein sogenanntes *ganzzahliges lineares Programm* zur Lösung des Problems zu verwenden. Diese Programme sind definiert durch eine Menge von Variablen und dazugehörigen Nebenbedingungen, die das Problem modellieren. Dazu ist dann eine Zielfunktion gegeben, die angibt, wie gut eine gefundene Lösung des Problems ist. Zum Lösen des Programms wird ein Solver verwendet, der den Wert der Zielfunktion minimiert oder maximiert. Für diese Arbeit wurde Gurobi [Inc13] in der Version 5.1 verwendet. Ob ein ganzzahliges lineares Programm effektiv eingesetzt werden kann, hängt dabei immer von der Problemstellung ab. Für manche Probleme sind sie ungeeignet, weil die benötigte Rechenzeit zu groß ist. Zum Finden eines Feedback-Arc-Set ist ein ganzzahliges lineares Programm jedoch gut geeignet, weil damit auch für große

Graphen sehr schnell eine optimale Lösung des Problems gefunden werden kann. Selbst für einen Graph mit über 1000 Knoten benötigt das Programm nur ca. 0,2 Sekunden auf einem durchschnittlichen Rechner (siehe auch Kapitel 5.11).

Wenn das ganzzahlige lineare Programm verwendet wird, ist es außerdem möglich, eine Variante des Problems zu betrachten, in der nicht die Anzahl der Kanten im Feedback-Arc-Set minimiert werden soll, sondern das Gewicht der Kanten (siehe Kapitel 4.3), was beim Zeichnen von Rechengraphen sinnvoll ist.

Das ganzzahlige lineare Programm zum Finden eines möglichst kleinen Feedback-Arc-Sets ist folgendermaßen definiert:

Definition 6 (Feedback-Arc-ILP)**.** Sei ein gerichteter Graph $G = (V, E)$ gegeben und sei $n = |V|$. Dann findet folgendes ganzzahliges lineares Programm ein minimales Feedback-Arc-Set F.
Variablen:

- $x_e \in \{0, 1\}$: gibt für die Kante e an, ob sie im Feedback-Arc-Set enthalten ist ($x_e = 1$) oder nicht ($x_e = 0$)
- $1 \leq x_v \leq n$: jeder Knoten erhält eine Nummer zwischen 1 und n, welche benötigt wird, um in den Nebenbedingungen die Kreisfreiheit sicherzustellen

Zielfunktion:

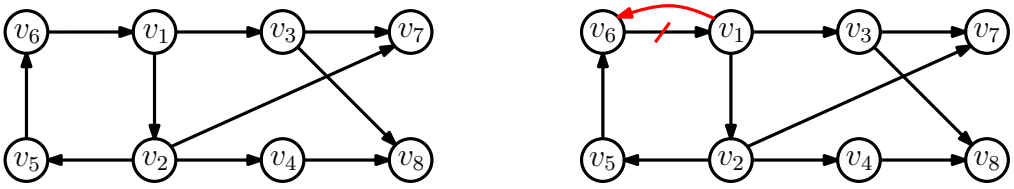
- Minimiere $\sum_{e \in E} x_e$

Nebenbedingungen:

- $\forall e = (u, v) \in E : x_u < x_v + n \cdot x_e$

Die Zielfunktion zählt die Kanten, die im Feedback-Arc-Set enthalten sind und die Nebenbedingungen stellen sicher, dass bei allen Kanten, die im Graph enthalten bleiben, der Startkonten eine kleinere Nummer hat als der Zielknoten und der Graph somit azyklisch ist. Für eine Kante, die entfernt wird, wird die Nebenbedingung „ausgeschaltet“, da hier die Ungleichung immer erfüllt ist.

Abbildung 3.1 zeigt ein einfaches Beispiel für das Entfernen von Kreisen. Der Ursprungsgraph enthält den Kreis (v_1,v_2,v_5,v_6,v_1), der durch das Umdrehen der Kante (v_6, v_1) entfernt wird. Anhand dieses Beispielgraphen werden auch die weiteren Schritte des Sugiyama-Frameworks in den folgenden Kapiteln gezeigt.



(a) Ursprungsgraph **(b)** Graph nach Entfernen von Kreisen

Abb. 3.1.: Entfernen von Kreisen

3.2. Lagenzuordnung

Da der Graph im letzten Schritt azyklisch gemacht wurde, ist es jetzt möglich, jeden Knoten des Graphen einer Lage zuzuordnen. Die Lage eines Knotens v wird dann mit $l(v)$ bezeichnet. Eine Lagenzuordnung heißt *gültig*, wenn für alle Kanten der Startknoten einer niedrigeren Lage zugeordnet ist als der Zielknoten. Die *Höhe der Lagenzuordnung* ist die Anzahl von verwendeten Lagen. Die *Breite einer Lage* ist die Anzahl von Knoten in dieser Lage, die *Breite der Lagenzuordnung* ist das Maximum der Breite aller Lagen. Das Ziel ist nun, eine gültige Lagenzuordnung zu finden, die eine möglichst geringe Höhe und Breite hat.

Das Finden einer Lagenzuordnung mit minimaler Höhe bei beliebiger Breite ist in Linearzeit möglich (Kap. 3.2.1). Deutlich komplizierter wird es allerdings, wenn man zusätzlich noch minimale Breite fordert. Denn das Problem, eine Lagenzuordnung mit minimaler Höhe und Breite zu finden, ist NP-schwer [BETT98]. Es gibt jedoch Verfahren, die bei einer vorgeschriebenen maximalen Breite Approximationsgarantien für die entstehende Höhe geben können. Zwei davon werden in Kapitel 3.2.2 und Kapitel 3.2.3 vorgestellt.

3.2.1. Lagenzuordnung mit minimaler Höhe

Eine Lagenzuordnung mit minimaler Höhe bei beliebiger Breite lässt sich mit einem einfachen Algorithmus finden. Es werden alle Quellen des Graphen auf die erste Lage gesetzt und alle anderen Knoten werden so zugeordnet, dass ihre Lagennummer um eins höher ist als die maximale Lagennummer ihrer Vorgänger (siehe Alg. 1). Mit einer geschickten rekursiven Implementierung lässt sich dieser Algorithmus in Linearzeit ausführen.

Algorithmus 1: Lagenzuordnung mit minimaler Höhe

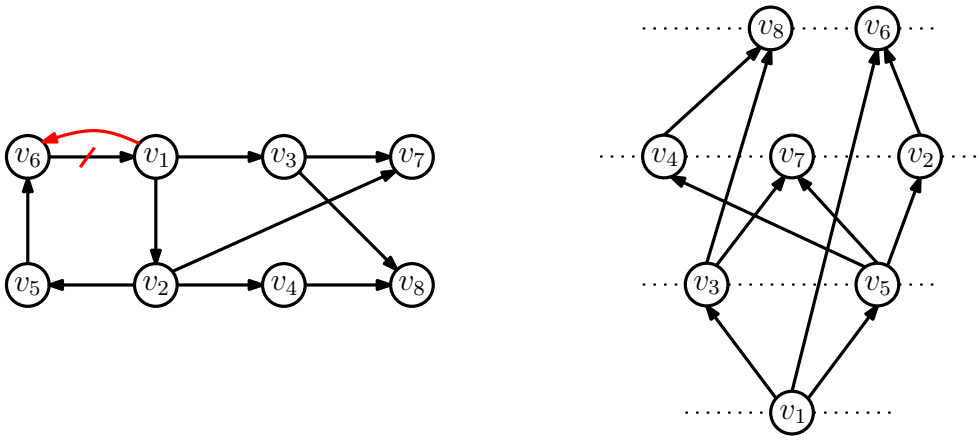
Eingabe : gerichteter Graph $G = (V, E)$
Ausgabe : gültige Lagenzuordnung mit minimaler Höhe

```
1 foreach v ∈ V do
2     if v ist Quelle in G then
3         l(v) = 1
4     else
5         l(v) = max{l(u) | u ∈ N⁻(v)} + 1
```

Abbildung 3.2b zeigt das Ergebnis des Algorithmus für den Beispielgraph aus dem letzten Kapitel. Wie man sieht kann kein Knoten mehr eine Lage nach unten verschoben werden, die Lagenzuordnung hat also minimale Höhe.

3.2.2. Precedence-Constrained Multi-Processor Scheduling

Es ist also in Linearzeit möglich, eine Lagenzuordnung mit minimaler Höhe zu finden. Wie bereits erwähnt, wird das Problem aber NP-schwer, wenn man zusätzlich noch mi-



(a) Beispielgraph **(b)** Graph mit Lagenzuordnung

Abb. 3.2.: Beispiel für eine Lagenzuordnung

nimale Breite fordert. Es gibt jedoch Verfahren, die das Problem approximieren können. Sie erhalten als Eingabe zusätzlich zum Graph noch die Breite, die die Lagenzuordnung maximal haben darf. Damit kann eine Approximationsschranke für die Höhe der Lagenzuordnung garantiert werden. Eines dieser Verfahren basiert auf einem Algorithmus zur Lösung des *Precedence-Constrained Multi-Processor Scheduling*-Problems. Hier geht es darum, für eine Menge von Aufträgen und Maschinen sowie einer partiellen Ordnung der Aufträge, einen Ablaufplan zu finden, der minimale Bearbeitungsdauer hat. Dieses Problem ist äquivalent zum Problem der Lagenzuordnung mit festgelegter Maximalbreite. Die Aufträge entsprechen hier den Knoten und die Maschinen entsprechen den Plätzen in den Lagen.

Das Verfahren (Alg. 2) arbeitet mit einer Liste, die immer die Knoten enthält, die selbst noch keiner Lage zugeordnet wurden, deren Vorgänger aber alle einer Lage zugeordnet wurden. Anfangs besteht diese Liste aus den Quellen des Graphen in beliebiger Reihenfolge. So lange diese Liste nicht leer ist, wird der erste Knoten u der Liste einer Lage zugeordnet. Dazu wird überprüft, ob u in die aktuelle Lage eingefügt werden kann. Dafür müssen zwei Bedingungen erfüllt sein: In der aktuellen Lage muss noch Platz sein und keiner der Vorgänger von u darf in der aktuellen Lage liegen. Sind beide Bedingungen erfüllt, wird u in die aktuelle Lage eingefügt, andernfalls wird u der nächst höheren Lage zugeordnet. Jetzt muss noch die Knotenliste aktualisiert werden. Dazu wird für jeden Nachfolger v von u überprüft, ob v noch nicht in der Liste ist und ob alle Vorgänger von v schon einer Lage zugeordnet wurden. Ist das der Fall, wird v hinten in die Liste eingefügt. Zuletzt wird noch u aus der Liste entfernt.

Man kann leicht zeigen, dass dann die Höhe der Lagenzuordnung eine $(2 - 1/B)$-Approximation der Höhe der optimalen Lösung ist, wobei B die maximal erlaubte Breite ist. Dieses Verfahren wird auch beim Zeichnen von Rechengraphen eingesetzt, wobei die Liste durch eine Prioritätswarteschlange ersetzt wird, die immer den schwersten Knoten zurück gibt. Diese Anpassung wird in Kapitel 4.4.2 näher erläutert.

Algorithmus 2: Lagenzuordnung durch PCMPS

Eingabe : gerichteter Graph $G = (V, E)$, maximale Knoten pro Lage B
Ausgabe : gültige Lagenzuordnung mit maximaler Breite B

```
1  U = leere Liste; k = 1; L_1 = ∅
2  füge alle Quellen von G in U ein
3  while U ist nicht leer do
4      u = erster Knoten in U
5      if |L_k| < B und für jeden Knoten w ∈ N^-(u) gilt: w ∈ L_1 ∪ L_2 ∪ ... ∪ L_{k-1}
       then
6          L_k = L_k ∪ {u}
7      else
8          k = k + 1; L_k = {u}
9      foreach v ∈ N^+(u) do
10         if v ∉ U und alle Vorgänger von v sind einer Lage zugeordnet then
11             füge v hinten in die Liste U ein
12     entferne u aus U
```

3.2.3. Lagenzuordnung nach Coffman-Graham

Ein etwas besseres, aber auch kompliziertes Verfahren wurde von Coffman und Graham [CG72] entwickelt (Alg. 3). Es liefert eine Lösung, dessen Höhe eine $(2 - 2/B)$-Approximation der Höhe der optimalen Lösung ist. Das Verfahren besteht aus zwei Teilen. Im ersten Schritt erhalten alle Knoten v ein eindeutiges Label $\pi(v)$. Im zweiten Teil werden diese Labels dann benutzt, um die Lagenzuordnung vorzunehmen. Beim Verteilen der Labels wird eine Ordnung verwendet, die Mengen von ganzen Zahlen vergleicht und folgendermaßen definiert ist: Für zwei Mengen S, T gilt $S < T$, wenn einer dieser drei Fälle gegeben ist:

- $S = \emptyset$ und $T \neq \emptyset$
- $S \neq \emptyset$, $T \neq \emptyset$ und $\max(S) < \max(T)$
- $S \neq \emptyset$, $T \neq \emptyset$, $\max(S) = \max(T)$ und $S \setminus \{\max(S)\} < T \setminus \{\max(T)\}$

Im Algorithmus wird diese Ordnung benutzt, um den Knoten auszuwählen, der als nächstes ein Label zugeordnet bekommt. Damit ein Knoten v ein Label bekommen kann, darf er selbst noch kein Label haben, aber alle Vorgänger von v müssen ein Label haben. Unter den Knoten, die diese Bedingungen erfüllen, wird nun der ausgewählt, für den die Menge $\{\pi(u) \mid (u, v) \in E\}$ unter der oben definierten Ordnung minimal ist. Dadurch erhält ein Knoten ein höheres Label, je weiter er von einer Quelle entfernt ist. Diese Eigenschaft wird dann im zweiten Teil des Verfahrens ausgenutzt.

Im zweiten Teil wird nun die Lagenzuordnung vorgenommen. Grundsätzlich kann ein Knoten v nur dann einer Lage zugeordnet werden, wenn bereits alle Nachfolger von v

einer anderen als der aktuellen Lage zugeordnet wurden. Beginnend mit der untersten Lage werden einer Lage nun so lange Knoten zugeordnet, bis sie entweder voll ist oder es keine Knoten mehr gibt, die ihr zugeordnet werden können. Ist das der Fall, wird mit der nächsten Lage weitergemacht. Wenn während des Verfahrens mehrere Knoten als nächstes der aktuellen Lage zugeordnet werden können, wird immer der mit dem höchsten Label ausgewählt. Das ist sinnvoll, da an diesem Knoten der längste Weg bis zu einer Quelle anschließt und der Knoten deshalb möglichst früh einer Lage zugeordnet werden sollte. Bei diesem Algorithmus ist zu beachten, dass die Quellen des Graphen sehr hohen Lagen zugeordnet werden. Das heißt verglichen mit den beiden anderen Algorithmen zeigen die Kanten jetzt alle in die entgegengesetzte Richtung.

Algorithmus 3: Lagenzuordnung nach Coffman und Graham

Eingabe : gerichteter Graph $G = (V, E)$, maximale Knoten pro Lage B
Ausgabe : gültige Lagenzuordnung mit maximaler Breite B

anfangs sind alle Knoten ohne Label
for $i = 1$ **to** $|V|$ **do**
    wähle einen Knoten v ohne Label so, dass alle Vorgänger von v ein Label haben und die Menge $\{\pi(u) \mid (u, v) \in E\}$ minimal ist
    $\pi(v) = i$
$k = 1$; $L_1 = \emptyset$; $U = \emptyset$
while $U \neq V$ **do**
    $u =$ Knoten in $V \setminus U$, so dass $N^+(u) \subseteq U$ gilt und $\pi(u)$ maximiert wird
    if $L_k < B$ und für jeden Knoten $w \in N^+(u)$ gilt: $w \in L_1 \cup L_2 \cup \ldots \cup L_{k-1}$ **then**
        $L_k = L_k \cup \{u\}$
    else
        $k = k + 1$; $L_k = \{u\}$
    $U = U \cup \{u\}$

3.3. Minimierung von Kantenkreuzungen

Wenn die Lagenzuordnung durchgeführt wurde, kann man bereits eine Zeichnung anfertigen, die die Hierarchie des Graphen gut wiedergibt. Es ist jedoch noch nicht festgelegt, wo die Knoten innerhalb einer Lage gezeichnet werden sollen. Die Positionen der Knoten haben jedoch großen Einfluss auf die Qualität der Zeichnung, besonders die Anzahl von Kantenkreuzungen ist stark von den Knotenpositionen abhängig. Deshalb folgt jetzt ein Schritt, der die Positionen der Knoten so festlegt, dass möglichst wenige Kantenkreuzungen entstehen.

Wenn man hier bereits mit den tatsächlichen Koordinaten der Knoten arbeitet, ist das jedoch extrem kompliziert und für größere Eingaben praktisch unmöglich. Um das Problem zu vereinfachen, werden deshalb zunächst sogenannte *Dummyknoten* eingefügt. Das geschieht folgendermaßen: Jede Kante, die nicht zwischen benachbarten Lagen

verläuft, wird entfernt und durch neue Kanten und Knoten ersetzt, so dass nur noch Kanten zwischen benachbarten Lagen existieren. Die neu eingefügten Knoten werden als Dummyknoten bezeichnet und am Ende des Algorithmus wieder entfernt. Mit Hilfe der Dummyknoten ist es deutlich einfacher, die Anzahl der Kantenkreuzungen zu minimieren, weil jetzt nicht mehr die exakten Koordinaten der Knoten entscheidend für die Existenz von Kreuzungen sind, sondern nur noch die Reihenfolge der (Dummy-)Knoten innerhalb der Lagen. Abbildung 3.3 zeigt den Graph aus dem letzten Kapitel, nachdem die Dummyknoten eingefügt wurden.

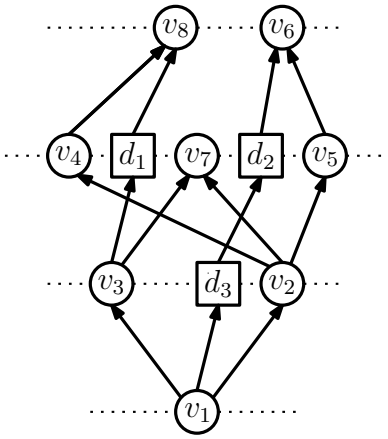


Abb. 3.3.: Beispielgraph nach dem Einfügen von Dummyknoten

Aber selbst mit dieser Vereinfachung ist es noch schwer, das Problem effizient zu lösen. Denn selbst wenn der Graph nur aus zwei Lagen besteht, er also bipartit ist, und die Reihenfolge der Knoten in einer Lage bereits festgelegt ist, bleibt das Problem NP-vollständig [BETT98]. Dieses Problem wird als *Two-Layer Crossing Problem* bezeichnet.

Definition 7 (Two-Layer Crossing Problem)**.** Sei ein bipartiter Graph $G = (L_1 \cup L_2, E)$ und eine Sortierung der Knoten in L_1 gegeben. Finde eine Sortierung der Knoten in L_2, so dass die Anzahl von Kantenkreuzungen in einer hierarchischen Zeichnung von G minimal ist.

Für dieses Problem können jedoch gute Ergebnisse mit Hilfe von Heuristiken gefunden werden. Um diese auch für einen Graph einsetzen zu können, der nicht bipartit ist, kommt ein Verfahren namens *Layer-by-Layer-Sweep* zum Einsatz. Das funktioniert folgendermaßen:

Zunächst wird das Two-Layer Crossing Problem für die ersten beiden Lagen gelöst, das heißt die Reihenfolge der Knoten in der ersten Lage wird fixiert und die Knoten in der zweiten Lage werden umgeordnet. Dann wird die zweite Lage fixiert und die dritte Lage umgeordnet. Das geht so lange weiter, bis die letzte Lage erreicht ist. Dann läuft das Verfahren rückwärts, das heißt es wird die letzte Lage fixiert und die vorletzte Lage umgeordnet, bis man wieder bei der ersten Lage angekommen ist. Diese Schritte werden dann so lange wiederholt, bis die sich die Anzahl von Kreuzungen nicht mehr verringert.

Dabei ist zu beachten, dass das Ergebnis der Heuristik auch von der Startsortierung in den Lagen abhängt. Deshalb sollte das Verfahren immer mit mehreren zufälligen Startsortierungen ausgeführt werden. Es ist außerdem sinnvoll, einen Durchlauf nicht sofort abzubrechen, wenn sich die Anzahl von Kreuzungen nicht mehr verringert, sondern ihn noch etwas weiter laufen zu lassen. Denn es kann durchaus vorkommen, dass sich die Anzahl von Kreuzungen kurzzeitig erhöht, aber nach einigen Schritten wieder sinkt.

Die in den folgenden drei Unterkapiteln vorgestellten Verfahren zur Kreuzungsminimierung verwenden die Layer-by-Layer-Sweep-Heuristik, unterscheiden sich aber dadurch, wie sie das Two-Layer Crossing Problem lösen.

3.3.1. Vertauschen benachbarter Knoten

Eine Möglichkeit ist das Verwenden der sogenannten *Adjacent-Exchange*-Heuristik (Alg. 4). Hier werden in der variablen Lage so lange die Positionen von benachbarten Knoten vertauscht, wie sich dadurch die Anzahl von Kreuzungen verringert. Damit der Algorithmus weiß, ob die sich Kantenkreuzungen reduzieren, werden sogenannte *Kreuzungszahlen* benutzt. Eine Kreuzungszahl $c_{u,v}$ gibt die Anzahl der Kreuzungen zwischen Kanten inzident zu u und Kanten inzident zu v an, wenn u links von v liegt. Es müssen also für jedes Paar von Knoten zwei Kreuzungszahlen berechnet werden. Da die Kreuzungszahl nur von der relativen Position der beiden Knoten abhängt und nicht von den anderen Knoten der Lage, reicht es dabei aus, die Kreuzungszahlen nur einmal zu berechnen.

Algorithmus 4: Adjacent-Exchange-Heuristik

Eingabe : bipartiter Graph $G = (L_1 \cup L_2, E)$, wobei L_1 die fixierte Lage ist; Sortierung der Knoten in L_1 und L_2

Ausgabe : verbesserte Sortierung der Knoten in L_2

1 berechne die Kreuzungszahlen für alle Paare von Knoten in L_2
2 **repeat**
3 | gehe die Knoten in L_2 von links nach rechts durch und vertausche die Positionen eines Knotens u und des rechts daneben liegenden Knotens v, wenn $c_{vu} < c_{uv}$ gilt
4 **until** die Anzahl von Kreuzungen reduziert sich nicht mehr

3.3.2. Mittelwert-Heuristik

Eine andere Heuristik zum Lösen des Two-Layer-Crossing Problems ist die sogenannte Mittelwert-Heuristik. Für diese Arbeit wurde eine Variante von Spönemann et al. [SFHM10] verwendet, da sie auch Knoten in variablen Lage beachtet, die keine Nachbarn in der fixierten Lage haben.

Bei der Mittelwert-Heuristik wird für jeden Knoten v in der variablen Lage ein Wert $s(v)$ berechnet. Dieser Wert ist die durchschnittliche Position aller Nachbarn von v in der fixierten Lage. Die Position eines Knotens v in seiner Lage wird mit $p(v)$ bezeichnet. Mit Hilfe dieser s-Werte werden dann die Knoten in der variablen Lage umsortiert.

Hat ein Knoten keine Nachbarn in der fixierten Lage, berechnet sich sein Wert aus dem Durchschnitt der Werte der links und rechts daneben liegenden Knoten in der variablen Lage. Das genaue Vorgehen der Heuristik zeigt Algorithmus 5. Eine positive Eigenschaft der Heuristik ist, dass sie eine Lösung ohne Kreuzungen findet, wenn eine solche existiert [BETT98].

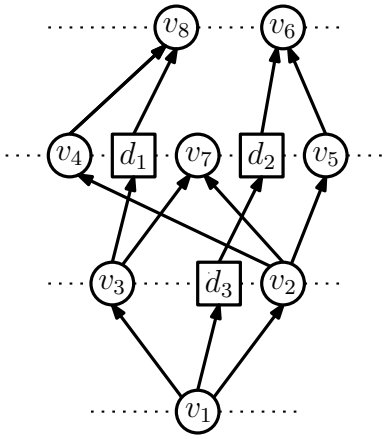
Algorithmus 5: Mittelwert-Heuristik

Eingabe : bipartiter Graph $G = (L_1 \cup L_2, E)$, wobei L_1 die fixierte Lage ist; Sortierung der Knoten in L_1 und L_2
Ausgabe : verbesserte Sortierung der Knoten in L_1

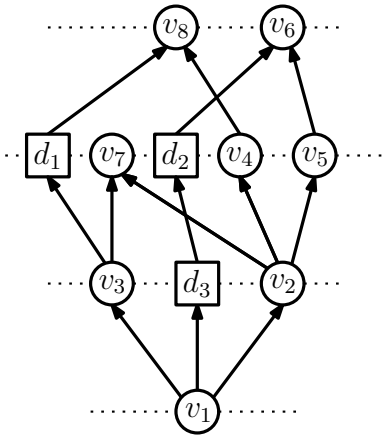
seien die Knoten in $L_2 = \{v_1, \ldots, v_k\}$ aufsteigend nach ihrer Position sortiert
for $i = 1$ **to** k **do**
  if v_i hat mindestens einen Nachbar in L_1 **then**
    $s(v_i) = \frac{1}{|N(v_i)|} \cdot \sum_{u \in N(v_i)} p(u)$
for $i = 1$ **to** k **do**
  if v_i hat keinen Nachbar in L_1 **then**
    if $s(v_{i+1})$ wurde bereits berechnet **then**
      $s(v_i) = \frac{1}{2} \cdot (s(v_{i+1}) + s(v_{i-1}))$
    else
      $s(v_i) = s(v_{i-1})$
sortiere die Knoten in L_2 anhand ihrer s-Werte; bei gleichen s-Werten behalte die alte Reihenfolge bei

Dieses Verfahren wird nun anhand des Beispielgraphen aus den vorhergehenden Kapiteln gezeigt. Abbildung 3.4a zeigt noch einmal den Graph, nachdem die Dummyknoten eingefügt wurden. Der Algorithmus startet damit, dass die unterste Lage fixiert wird und die zweite Lage neu sortiert wird. Hier passiert nichts, da alle Knoten den gleichen s-Wert erhalten und deshalb die ursprüngliche Sortierung beibehalten wird. Interessant wird es jedoch im nächsten Schritt. Hier wird die zweite Lage fixiert und die dritte Lage wird umsortiert. Die s-Werte für die Knoten in der dritten Lage lauten $s(v_4) = 3$, $s(d_1) = 1$, $s(v_7) = 2$, $s(d_2) = 2$ und $s(v_5) = 3$. Die neue Reihenfolge der Knoten ist also $(d_1, v_7, d_2, v_4, v_5)$, wie sie auch in Abbildung 3.4b zu sehen ist. Im nächsten Schritt passiert wieder nichts, womit der erste Vorwärts-Durchgang abgeschlossen ist.

Jetzt wird die oberste Lage fixiert und die dritte Lage wird umgeordnet. Die s-Werte lauten nun $s(d_1) = 1$, $s(d_2) = 2$, $s(v_4) = 1$ und $s(v_5) = 2$. Der Knoten v_7 hat keinen Nachbar in der obersten Lage. Das heißt er erhält als s-Wert den Durchschnitt der Knoten links und rechts von ihm, also d_1 und d_2. Damit gilt $s(v_7) = 1,5$ und die neue Reihenfolge lautet $(d_1, v_4, v_7, d_2, v_5)$. Abbildung 3.5a zeigt die Situation nach diesem Schritt. Als Nächstes wird die dritte Lage fixiert und die zweite umgeordnet, wodurch der Knoten d_3 nach rechts verschoben wird (Abb. 3.5b). Damit ist der Rückwärts-Durchgang abgeschlossen und das Verfahren beginnt von vorne.

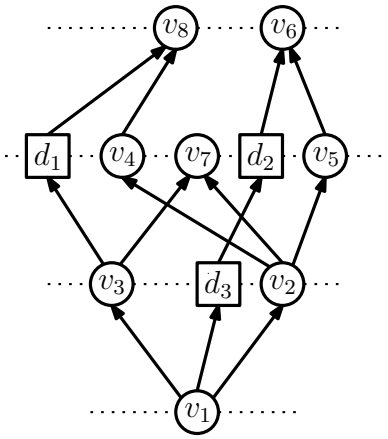


(a) Graph nach Einfügen der Dummyknoten

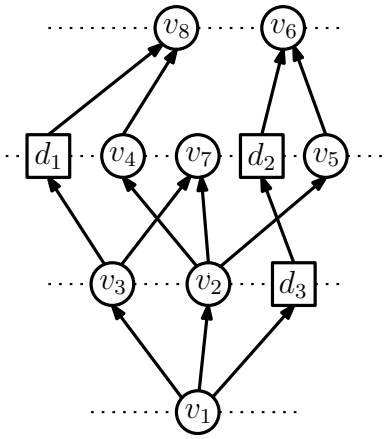


(b) Graph nach einem Vorwärts-Durchgang

Abb. 3.4.: Minimierung von Kantenkreuzungen



(a) Graph nach Umordnen der dritten Lage



(b) Graph nach einem kompletten Durchlauf

Abb. 3.5.: Minimierung von Kantenkreuzungen

Nach einem weiteren Vorwärts-Durchgang sind dann alle Kreuzungen aus der Zeichnung entfernt (Abb. 3.6). Es sei hier noch einmal darauf hingewiesen, dass das Ergebnis des Verfahrens auch von der Startsortierung abhängt. Wären zum Beispiel die Knoten v_2 und v_3 zu Beginn vertauscht gewesen, hätte die Heuristik keine Zeichnung ohne Kreuzungen gefunden.

3.3.3. Lineares Programm für zwei Lagen

Eine weitere Möglichkeit, das Two-Layer Crossing Problem zu lösen, ist das Verwenden eines ganzzahligen linearen Programms. Der Vorteil gegenüber den anderen Verfahren ist, dass so tatsächlich die beste Lösung mit der minimalen Anzahl von Kreuzungen

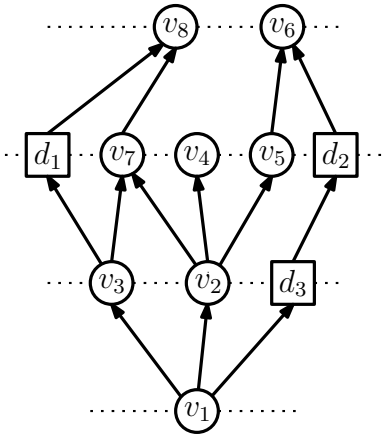


Abb. 3.6.: Graph nach der Kreuzungsminimierung

gefunden wird. Dafür wird jedoch auch mehr Rechenzeit benötigt. Trotzdem bleibt diese im Rahmen, weil in der Praxis die meisten Rechengraphen aufgrund der begrenzten Zeichenfläche nicht allzu viele Knoten in jeder Lage haben. Das ganzzahlige lineare Programm ist folgendermaßen definiert:

Definition 8 (Kreuzungs-ILP)**.** Sei ein bipartiter Graph $G = (L_1 \cup L_2, E)$ gegeben, wobei L_1 die fixierte und L_2 die variable Lage ist. Seien außerdem die Kreuzungszahlen gegeben und sei $p(v)$ die Position eines Knotens v in seiner Lage. Dann findet folgendes ganzzahliges lineares Programm eine Reihenfolge der Knoten in L_2, so dass die Anzahl von Kantenkreuzungen minimal ist.
Variablen:

- $x_{uv} \in \{0, 1\}$ für alle Knoten $u, v \in L_2$ mit $p(u) < p(v)$: gibt die Lage der Knoten zueinander an. Bei $x_{uv} = 1$ liegt in der Lösung u links von v, bei $x_{uv} = 0$ liegt in der Lösung u rechts von v.

Zielfunktion:

- Minimiere $\sum_{p(u)<p(v)\in L_2} (c_{uv} - c_{vu}) \cdot x_{uv}$

Nebenbedingungen:

- $\forall u, v, w \in L_2$ mit $p(u) < p(v) < p(w) : 0 \leq x_{uv} + x_{vw} - x_{uw} \leq 1$

Dabei ist zu beachten, dass der Wert der Zielfunktion nicht die Anzahl von Kantenkreuzungen angibt. Die Anzahl von Kantenkreuzungen wird durch folgenden Term beschrieben:

$$\sum_{p(u)<p(v)\in L_2} (c_{uv} \cdot x_{uv} + c_{vu} \cdot (1 - x_{uv})) = \sum_{p(u)<p(v)\in L_2} (c_{uv} - c_{vu}) \cdot x_{uv} + \sum_{p(u)<p(v)\in L_2} c_{vu}$$

Da $\sum_{p(u)<p(v)\in L_2} c_{vu}$ jedoch eine Konstante ist, kann sie in der Zielfunktion weggelassen werden. Die Nebenbedingungen stellen sicher, dass die in den Variablen festgelegten

Knotenpositionen konsistent sind.

Wird das lineare Programm während eines Layer-by-Layer-Sweep eingesetzt, sollte die gefundene Lösung nur dann übernommen werden, wenn sie tatsächlich besser ist als die bisherige Ordnung. Ansonsten kann es manchmal zu unerwünschten Ergebnissen kommen, zum Beispiel wenn die fixierte Lage nur einen Knoten hat, was beim Zeichnen von Rechengraphen durchaus vorkommt, da in der ersten Lage nur der Startknoten des Graphen liegt. Dann ist die Reihenfolge der Knoten in der variablen Lage unwichtig, weil es nie Kreuzungen gibt. Trotzdem sollte man in diesem Fall die vorhandene Reihenfolge beibehalten, weil sonst die Verbesserungen der vorangegangenen Schritte außer Acht gelassen werden und der Layer-by-Layer-Sweep praktisch wieder komplett von vorne beginnen würde.

3.3.4. Lineares Programm für den ganzen Graph

Eine weitere Möglichkeit, die Kantenkreuzungen zu minimieren, ist die Verwendung eines ganzzahligen linearen Programms für den kompletten Graph. Anders als in den bisher vorgestellten Verfahren wird hier nicht die Layer-by-Layer-Sweep Heuristik verwendet, sondern in einem Schritt die Reihenfolgen der Knoten in allen Lagen berechnet. Dadurch ist garantiert, dass die entstehende Zeichnung des Graphen tatsächlich nur die minimale Anzahl von Kreuzungen enthält. Das lineare Programm ist folgendermaßen definiert:

Definition 9 (Kreuzungs-ILP für den ganzen Graph)**.** Seien ein Graph $G = (V, E)$ mit Lagenzuordnung gegeben und bezeichne $p(v)$ die Position eines Knotens v in seiner Lage. Dann findet folgendes ganzzahliges lineares Programm für jede Lage $L_1, \ldots, L_k$ eine Reihenfolge der Knoten in dieser Lage, so dass die Anzahl von Kantenkreuzungen in G minimal ist.
Variablen:

- $x_{uv} \in \{0, 1\}$ für alle Knoten $u, v \in L_i$ und für alle i: gibt die Position der Knoten zueinander an. Bei $x_{uv} = 1$ liegt in der Lösung u links von v, bei $x_{uv} = 0$ liegt in der Lösung u rechts von v.
- $\mathrm{cr}_{e_1,e_2} \in \{0, 1\}$ für jedes Paar von Kanten e_1 und e_2, die zwischen den gleichen Lagen verlaufen: gibt an, ob sich die Kanten e_1 und e_2 schneiden.

Zielfunktion:

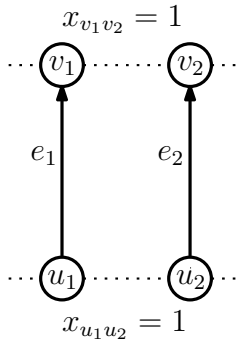
- Minimiere $\sum_{1 \leq i \leq k-1} \sum_{e_1=(u_1,v_1), e_2=(u_2,v_2) \text{ mit } u_1,u_2 \in L_i \text{ und } v_1,v_2 \in L_{i+1}} \mathrm{cr}_{e_1,e_2}$

Nebenbedingungen:

- $\forall i \in \{1, \ldots, k\} : (\forall u, v \in L_i : x_{uv} = 1 - x_{vu})$
- $\forall i \in \{1, \ldots, k-1\} : (\forall u, v, w \in L_i \text{ mit } p(u) < p(v) < p(w) : 0 \leq x_{uv} + x_{vw} - x_{uw} \leq 1)$
- $\forall i \in \{1, \ldots, k-1\} : (\forall e_1 = (u_1, v_1), e_2 = (u_2, v_2) \text{ mit } u_1, u_2 \in L_i \text{ und } v_1, v_2 \in L_{i+1} : \mathrm{cr}_{e_1,e_2} \geq x_{u_1u_2} + (1 - x_{v_1v_2}) - 1 \text{ und } \mathrm{cr}_{e_1,e_2} \geq (1 - x_{u_1u_2}) + x_{v_1v_2} - 1)$

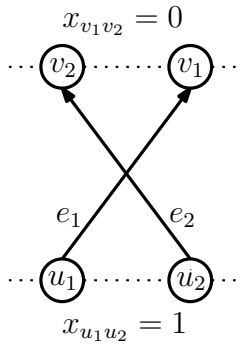
Die Zielfunktion des linearen Programm entspricht der Anzahl von Kantenkreuzungen, die minimiert werden sollen. Die ersten beiden Nebenbedingungen stellen sicher, dass die

in den Variablen festgelegten Knotenpositionen konsistent sind. Die dritte Nebenbedingung sorgt dafür, dass die cr-Werte korrekt sind. Das heißt wenn die Knotenpositionen so festgelegt sind, dass sich zwei Kanten e_1 und e_2 schneiden, muss $\mathrm{cr}_{e_1,e_2} = 1$ gelten. Abbildung 3.7 zeigt die zwei grundlegenden Fälle, die hier auftreten können. Die anderen beiden Fälle ergeben sich durch Umbenennen der Knoten.



$$\mathrm{cr}_{e_1,e_2} \geq x_{u_1u_2} + (1 - x_{v_1v_2}) - 1 = 1 + (1-1) - 1 = 0$$
$$\mathrm{cr}_{e_1,e_2} \geq (1 - x_{u_1u_2}) + x_{v_1v_2} - 1 = (1-1) + 1 - 1 = 0$$
$$\Rightarrow \mathrm{cr}_{e_1,e_2} = 0$$

(a) Kanten schneiden sich nicht $\Rightarrow$ cr $= 0$



$$\mathrm{cr}_{e_1,e_2} \geq x_{u_1u_2} + (1 - x_{v_1v_2}) - 1 = 1 + (1-0) - 1 = 1$$
$$\mathrm{cr}_{e_1,e_2} \geq (1 - x_{u_1u_2}) + x_{v_1v_2} - 1 = (1-1) + 0 - 1 = -1$$
$$\Rightarrow \mathrm{cr}_{e_1,e_2} = 1$$

(b) Kanten schneiden sich $\Rightarrow$ cr $= 1$

Abb. 3.7.: Werte von cr in Abhängigkeit von den Knotenpositionen

Dieses Verfahren ist das Beste, da es eine optimale Lösung liefert. Das Problem ist jedoch, dass es nur für kleine Graphen schnell genug funktioniert und für große Graphen zu langsam ist, um in der Praxis eingesetzt werden zu können. Für einen Graph mit 98 Knoten Knoten und 126 Kanten war die Rechenzeit mit etwa einer Minute gerade noch akzeptabel, für größere Graphen steigt sie aber sehr schnell an. Der Test mit einem Graph bestehend aus 131 Knoten und 171 Kanten wurde nach 30 Minuten abgebrochen, als noch keine Lösung gefunden wurde.

3.4. Festlegen der Knotenpositionen und Zeichnen der Kanten

Wenn für alle Lagen eine gute Reihenfolge der Knoten gefunden wurde, werden im letzten Schritt noch die konkreten Zeichenpositionen der Knoten festgelegt. Hier kann es zum Beispiel ein Ziel sein, dass Knoten, die miteinander verbunden sind, möglichst genau

untereinander gezeichnet werden oder dass Kanten, die Dummyknoten enthalten, trotzdem geradlinig gezeichnet werden. Abbildung 3.8 zeigt ein Beispiel, wie die Knoten und Kanten des Beispielgraphen gezeichnet werden könnten. Hier wurde vor allem darauf geachtet, dass miteinander verbundene Knoten möglichst übereinander gezeichnet werden. Die Dummyknoten wurden entfernt und längere Kanten werden als Kurven dargestellt. Außerdem werden in diesem Schritt auch die am Anfang umgedrehten Kanten wieder in die richtige Richtung gezeichnet. Im Beispiel ist das die Kante (v_6, v_1).

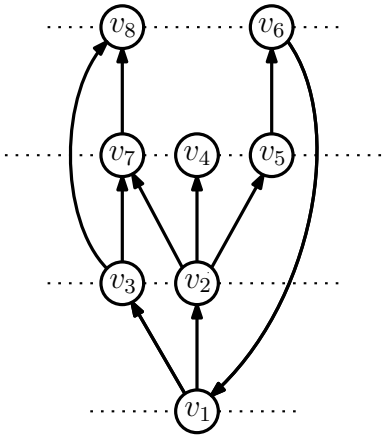


Abb. 3.8.: Fertige hierarchische Zeichnung

4. Hierarchisches Zeichnen von Rechengraphen

Das Sugiyama-Framework ist eine gute Grundlage für einen Algorithmus, der Rechengraphen zeichnen soll. Die Tatsache, dass Kanten in Rechengraphen von links nach rechts und nicht von unten nach oben gezeichnet werden, ändert nichts am generellen Vorgehen. Die Lagen entsprechen jetzt lediglich vertikalen statt horizontalen Linien. Ein Problem ist jedoch, dass die Rechengraphen meist deutlich größer sind als der Platz, der zum Zeichnen zur Verfügung steht. Deshalb muss der Algorithmus noch an einigen Stellen modifiziert und erweitert werden. Außerdem muss beim Zeichnen von Rechengraphen auf die Kantengewichte geachtet werden, welche im Sugiyama-Framework keine Rolle spielen. Es wird jetzt zunächst das generelle Vorgehen (siehe Alg. 6) erläutert, bevor dann in den folgenden Unterkapiteln die einzelnen Schritte genauer erklärt werden.

Algorithmus 6: Zeichenalgorithmus für Rechengraphen

1 sortiere unwichtige Knoten und Kanten aus
2 entferne Kreise
3 ordne jeden Knoten einer Lage zu
4 entferne Knoten, bis jede Lage höchstens so hoch wie die Zeichenfläche ist
5 entferne Lagen, bis alle verbleibenden Lagen in die Zeichenfläche passen
6 füge Dummyknoten ein
7 reduziere die Anzahl von Kantenkreuzungen
8 entferne Kanten, wenn die Zeichnung zu viele Kreuzungen enthält
9 lege die Knotenpositionen fest und zeichne die Knoten
10 zeichne die Kanten

Zuerst werden Knoten und Kanten mit sehr kleinem Gewicht aus dem Graph entfernt (Kap. 4.1). Da diese Knoten und Kanten am Ende mit hoher Wahrscheinlichkeit sowieso nicht gezeichnet werden, führt das frühe Entfernen nicht zu einem signifikanten Informationsverlust, steigert aber die Qualität der Ausgabe. Außerdem sinkt der Aufwand für die folgenden Schritte, da die Größe des Graphen reduziert wird.

Anschließend wird, wie im Sugiyama-Framework üblich, das Entfernen von Kreisen durchgeführt (Kap. 4.3). Der Unterschied zum Verfahren im letzten Kapitel besteht zum einen darin, dass die Kanten nicht umgedreht, sondern entfernt werden. Zum anderen soll nicht die Anzahl der entfernten Kanten minimiert werden, sondern das Gewicht der Kanten. Diese Anpassung ist sinnvoll, weil schwere Kanten im Graph enthalten bleiben sollen.

Bei der nun folgenden Lagenzuordnung (Kap. 4.4) werden Methoden verwendet, die die Anzahl der Knoten pro Lage begrenzen, damit die Knoten möglichst gleichmäßig auf die Lagen verteilt werden. Trotzdem können hier einer Lage mehr Knoten zugeordnet werden als später auf die Zeichenfläche passen. Das Aussortieren von zu viel vorhandenen Knoten erfolgt erst im nächsten Schritt.

Der nächste Teil des Algorithmus ist im Sugiyama-Framework nicht enthalten und wurde speziell für das Zeichnen von Rechengraphen eingefügt. Da nach der Lagenzuordnung im Allgemeinen mehr Lagen vorhanden sind als auf die Zeichenfläche passen und die Lagen zu hoch sind, weil sie zu viele Knoten enthalten, müssen nun die unwichtigsten Knoten aussortiert werden. Das geschieht in zwei Schritten. Zuerst werden aus jeder Lage so lange Knoten entfernt, bis die Lage niedrig genug ist, damit sie auf die Zeichenfläche passt (Kap. 4.5.1). Anschließend ist es häufig der Fall, dass der Graph noch zu viele Lagen enthält, das heißt die Zeichenfläche ist nicht breit genug, um alle Lagen darzustellen. Hier werden nun die unwichtigsten Lagen entfernt, bis alle übrigen Lagen auf die Zeichenfläche passen (Kap. 4.5.2).

Danach werden die Kantenkreuzungen reduziert (Kap. 4.6). Hier wird nicht nur auf die Anzahl von Kreuzungen geachtet, sondern auch auf das Gewicht der beteiligten Kanten. Das ist sinnvoll, da schwere Kanten möglichst ohne Kreuzungen gezeichnet werden sollen, während das bei leichteren Kanten nicht so wichtig ist.

Sollten nach der Kreuzungsminimierung noch zu viele Kreuzungen im Graph sein, werden im nächsten Schritt die unwichtigsten Kanten entfernt (Kap. 4.7). Das sind solche, die viele andere Kanten kreuzen und selbst ein relativ kleines Gewicht haben. Durch diesen Schritt ist garantiert, dass die Ausgabezeichnung nur eine geringe Anzahl von Kantenkreuzungen enthält.

Als Nächstes werden die Knotenpositionen festgelegt. Hier werden nebeneinander liegende Knoten in unterschiedlichen Abständen gezeichnet, abhängig davon, ob es Dummyknoten oder echte Knoten sind. Außerdem wird hier ein Verfahren vorgestellt, das beim Verwenden von orthogonalen Kanten die Anzahl der Kantenknicke reduziert.

Als Letztes werden die Kanten gezeichnet, wobei hier zwei Ansätze näher vorgestellt werden. Im ersten werden Kanten, wie bei Rechengraphen üblich, orthogonal gezeichnet, also bestehend aus horizontalen und vertikalen Segmenten. Im zweiten Ansatz werden die Kanten durch Kurven dargestellt.

Im Folgenden bezeichnet $h(v)$ die Höhe eines Knotens und $b(v)$ die Breite. Die Position von v wird durch $x(v)$ und $y(v)$ angegeben, wobei damit die obere linke Ecke eines Knotens definiert ist. Außerdem bezeichnet $w(v)$ das Gewicht des Knotens und $l(v)$ gibt die Lage an, in der v liegt.

4.1. Aussortieren von unwichtigen Knoten und Kanten

Am Anfang des Zeichenalgorithmus werden zuerst unwichtige Knoten und Kanten entfernt. Dazu gehören zunächst einmal Schlaufen, das heißt Kanten, bei denen Start- und Zielknoten identisch sind. Anschließend werden Knoten und Kanten, deren Gewicht kleiner ist als eine festgelegte Schranke, aussortiert. Diese würden im Verlauf des Algorith-

mus mit hoher Wahrscheinlichkeit sowieso wegfallen. Dadurch, dass sie früher entfernt werden, ergeben sich zwei Vorteile. Zum einen sinkt die benötigte Rechenzeit, da die folgenden Schritte mit einem kleineren Graph ausgeführt werden. Zum anderen ist dadurch einen bessere Lagenzuordnung möglich, wodurch sich die Ausgabe verbessert. In Kapitel 5.3 wird genauer untersucht, wie sich das Entfernen von leichten Knoten und Kanten und der dabei festgelegte Schrankenwert auf das Ergebnis auswirkt.

In diesem Schritt ist es sinnvoll, zuerst die leichten Knoten zu entfernen, weil sich dadurch unter Umständen das Gewicht von anderen Kanten erhöhen kann. Wie das Ändern von Kantengewichten funktioniert, wird im nächsten Kapitel erklärt. Erst wenn die leichten Knoten entfernt wurden und die Kantengewichte angepasst wurden, werden die leichten Kanten aus dem Graph entfernt.

4.2. Umlegen von Gewichten beim Entfernen von Knoten

Wenn während des Algorithmus ein Knoten aus dem Graph entfernt wird, geht damit auch immer ein Informationsverlust einher, da sowohl der Knoten selbst als auch inzidente Kanten entfernt werden. Um diesen Informationsverlust möglichst gering zu halten, wird, wann immer ein Knoten während des Algorithmus entfernt wird, getestet, ob das Gewicht des Knotens bzw. der inzidenten Kanten auf andere Kanten umgelegt werden kann.

Abbildung 4.1 zeigt ein einfaches Beispiel für einen solchen Fall. Hier soll der Knoten v_2 entfernt werden. Durch einfaches Löschen des Knotens und der inzidenten Kanten geht die Information verloren, dass es zehn Schüler gibt, die mit einem Umweg über v_2 von v_1 nach v_3 gekommen sind. Deshalb wird beim Löschen von v_2 das Gewicht der Kante (v_2, v_3) zum Gewicht der Kante (v_1, v_3) hinzuaddiert. Das Gewicht dieser Kante erhöht sich dadurch auf 40 und ein Teil der zuvor vorhandenen Information ist immer noch im Graph enthalten. Nichtsdestotrotz geht bei diesem Vorgehen manche Information verloren. So ist es nach dem Entfernen nicht mehr möglich zu erkennen, dass es zehn Schüler gab, die nicht direkt von v_1 nach v_3 gelangt sind. Außerdem ist nicht mehr erkennbar, dass es zwei Schüler gab, die zwar bis v_2 gekommen sind, dann aber abgebrochen haben.

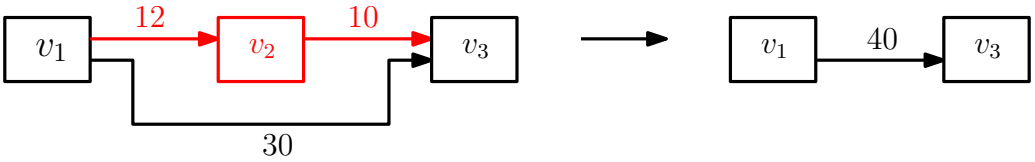


Abb. 4.1.: Umlegen von Kantengewichten

Dieses Verfahren funktioniert in vielen Fällen sehr gut, allerdings existiert nicht immer eine Kante, auf die sich das Gewicht umlegen lässt. Dieses Problem lässt sich mit einer einfachen Erweiterung des Verfahrens lösen. Wenn eine solche Kante nicht existiert, kann sie in den Graph eingefügt werden. Abbildung 4.2 zeigt ein ähnliches Beispiel wie Abbildung 4.1, bei dem aber keine Kante existiert, auf die das Gewicht umgelegt werden

kann. Deshalb wird hier eine neue Kante zwischen v_1 und v_3 eingefügt, die dann das Gewicht der gelöschten Kante (v_2, v_3) erhält.

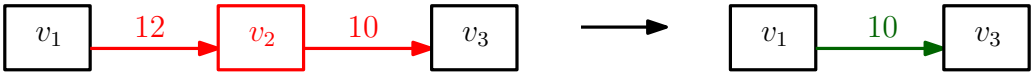


Abb. 4.2.: Einfügen einer neuen Kante

In den bisherigen Beispielen hatte der zu löschende Knoten nur jeweils eine eingehende und ausgehende Kante. In einem solchen Fall ist es einfach, das Gewicht auf eine andere Kante umzulegen oder eine neue Kante einzufügen. Wenn ein Knoten jedoch mehrere eingehende und ausgehende Kanten hat, muss genauer festgelegt werden, wie die Gewichte dieser Kanten umgelegt werden. Eine Möglichkeit wäre, eine Kante von jedem Vorgänger zu jedem Nachfolger des zu löschenden Knotens einzufügen, wodurch viel Information erhalten bleibt. Das ist jedoch nicht sinnvoll, weil dadurch zu viele Kanten eingefügt werden. Eine andere Möglichkeit ist, nur die wichtigste (also schwerste) eingehende und ausgehende Kante zu beachten. Damit ist immer ein einfacher Fall wie in Abbildung 4.1 gegeben. Hier geht allerdings sehr viel Information verloren.

Es wird deshalb ein Mittelweg dieser beiden Möglichkeiten verwendet. Algorithmus 7 beschreibt das Vorgehen im allgemeinen Fall. Dabei werden auch die einfachen Spezialfälle, wie sie in Abbildung 4.1 und Abbildung 4.2 gezeigt werden, abgedeckt. Das Verfahren funktioniert also für alle Fälle gleichermaßen. Abbildung 4.3 zeigt das Vorgehen für einen komplizierteren Fall an einem Beispiel.

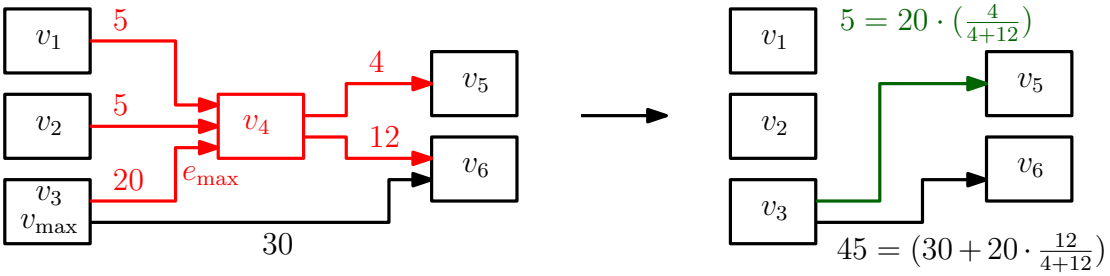


Abb. 4.3.: Gewicht umlegen bei mehreren eingehenden und ausgehenden Kanten

Zuerst wird überprüft, ob die schwerste Kante $e_{\max}$, die zum zu löschenden Knoten v inzident ist, eine eingehende oder ausgehende Kante von v ist. Ist es eine eingehende Kante, wird für jede Kante, die vom Startknoten $v_{\max}$ von $e_{\max}$ zu einem Nachfolger von v (im Beispiel v_5 und v_6) verläuft, geprüft, ob sie schon im Graph existiert. Ist das der Fall (siehe (v_3, v_6)), wird ihr Gewicht um einen gewissen Betrag b erhöht. Existiert die Kante nicht, wird sie in den Graph eingefügt (siehe (v_3, v_5)) und erhält b als Gewicht. Der Betrag b ist abhängig vom Gewicht von $e_{\max}$ und vom Gewicht der Kante von v zum gerade bearbeiteten Nachfolger u von v. Es gilt dann

$$b = w((v_{\max}, v)) \cdot \left\lfloor \frac{w((v,u))}{\sum_{a \in N^+(v)} w((v,a))} \right\rfloor$$

Das heißt das Gewicht der schwersten Kante $e_{\max}$ wird proportional zum Gewicht der alten und bald nicht mehr vorhandenen Kanten von v zu seinen Nachfolgern auf die neuen Kanten von $v_{\max}$ zu den Nachfolgern von v aufgeteilt. Am Ende wird der Knoten v und alle inzidenten Kanten gelöscht. Für den Fall, dass $e_{\max}$ eine ausgehende Kante von v ist, funktioniert das Verfahren analog.

Algorithmus 7: Allgemeiner Fall für das Umlegen von Kantengewichten

Eingabe : gerichteter Graph $G = (V, E, w)$ mit Kantengewichten, zu löschender Knoten v

Ausgabe : Graph G ohne Knoten v und mit angepassten Kantengewichten

setze $e_{\max}$ = schwerste zu v inzidente Kante
setze $v_{\max}$ = anderer inzidenter Knoten zu $e_{\max}$ außer v
if $e_{\max}$ ist eingehende Kante in v **then**
    foreach $u \in N^+(v)$ **do**
        if Kante $(v_{\max}, u)$ existiert in G **then**
            $w((v_{\max}, u)) = w((v_{\max}, u)) + w((v_{\max}, v)) \cdot \left\lfloor \frac{w((v,u))}{\sum_{a \in N^+(v)} w((v,a))} \right\rfloor$
        else
            füge Kante $(v_{\max}, u)$ ein mit
            $w((v_{\max}, u)) = w((v_{\max}, v)) \cdot \left\lfloor \frac{w((v,u))}{\sum_{a \in N^+(v)} w((v,a))} \right\rfloor$
else
    foreach $u \in N^-(v)$ **do**
        if Kante $(u, v_{\max})$ existiert in G **then**
            $w((u, v_{\max})) = w((u, v_{\max})) + w((v, v_{\max})) \cdot \left\lfloor \frac{w((u,v))}{\sum_{a \in N^-(v)} w((a,v))} \right\rfloor$
        else
            füge Kante $(u, v_{\max})$ ein mit
            $w((u, v_{\max})) = w((v, v_{\max})) \cdot \left\lfloor \frac{w((u,v))}{\sum_{a \in N^-(v)} w((a,v))} \right\rfloor$
lösche Knoten v und alle inzidenten Kanten aus G
return G

Um zu verhindern, dass beim Umlegen von Knotengewichten viele Kanten eingefügt werden, die nur sehr geringes Gewicht haben, wird eine Schranke verwendet, die ein Mindestgewicht für eine neue Kante angibt. Empfehlenswert ist hier der Wert, der für das Entfernen von leichten Kanten am Anfang des Algorithmus verwendet wurde (siehe Kapitel 4.1). Es gibt außerdem die Möglichkeit, keine neuen Kanten zuzulassen, sondern Gewichte nur auf existierende Kanten umzulegen. Mit Hilfe dieses Verfahrens kann das Gewicht der Kanten in der Ausgabezeichnung deutlich erhöht werden. In Kapitel 5.4 wird genauer darauf eingegangen, wie sich das Umlegen von Gewichten auswirkt.

4.3. Entfernen von Kreisen

Nachdem die leichten Knoten und Kanten entfernt wurden, werden nun eventuell vorhandene Kreise aus dem Rechengraph entfernt. Dazu wird das in Kapitel 3.1 vorgestellte ganzzahlige lineare Programm leicht angepasst. Das passiert deshalb, weil beim Zeichnen von Rechengraphen die Kantengewichte ein wichtige Rolle spielen und somit das Feedback-Arc-Set mit minimaler Kantenzahl nicht notwendigerweise die beste Lösung ist. Denn dann ist es möglich, dass die Kanten im Feedback-Arc-Set ein sehr hohes Gewicht haben und somit wichtige Kanten aus dem Graph entfernt werden, was eigentlich verhindert werden soll. Eine bessere Lösung kann somit durchaus mehr Kanten enthalten, deren Gewicht aber geringer ist. Deshalb minimiert das lineare Programm nicht die Anzahl der Kanten, sondern das Gesamtgewicht der Kanten im Feedback-Arc-Set. Das lineare Programm ist folgendermaßen definiert:

Definition 10 (Feedback-Arc-ILP mit Kantengewichten)**.** Sei ein gerichteter Graph $G = (V, E, w)$ mit Kantengewichten gegeben und sei $n = |V|$. Dann findet folgendes ganzzahliges lineares Programm ein Feedback-Arc-Set F mit minimalem Gewicht.
Variablen:

- $x_e \in \{0, 1\}$: gibt für die Kante e an, ob sie im Feedback-Arc-Set enthalten ist ($x_e = 1$) oder nicht ($x_e = 0$)
- $1 \leq x_v \leq n$: jeder Knoten erhält eine Nummer zwischen 1 und n, welche benötigt wird, um in den Nebenbedingungen die Kreisfreiheit sicherzustellen

Zielfunktion:

- Minimiere $\sum_{e \in E} x_e \cdot w(e)$

Nebenbedingungen:

- $\forall e = (u, v) \in E : x_u < x_v + n \cdot x_e$

Die Zielfunktion addiert die Gewichte der Kanten, die entfernt werden. Das ist der einzige Unterschied zum ganzzahligen linearen Programm, das in Kapitel 3.1 vorgestellt wurde. Die Lösung ist also ein Feedback-Arc-Set mit minimalem Gewicht.

Wie viel besser dieses Programm ist im Vergleich zu dem, das nicht auf die Gewichte achtet, ist stark von der Eingabe abhängig. Es kann sein, dass sie die gleichen Lösungen liefern, es kann aber auch passieren, dass das Gewicht der entfernten Kanten durch das erweiterte ganzzahlige lineare Programm stark reduziert wird. In Kapitel 5.2 werden die Lösungen der beiden Programme für unterschiedliche Testgraphen miteinander verglichen.

4.4. Lagenzuordnung

Durch das Entfernen von Kanten im letzten Schritt ist nun ein kreisfreier Graph vorhanden. Damit ist es jetzt möglich, die Lagenzuordnung vorzunehmen, wobei ein Verfahren verwendet wird, das die Anzahl der Knoten pro Lage begrenzt. Das ist nötig, weil sonst

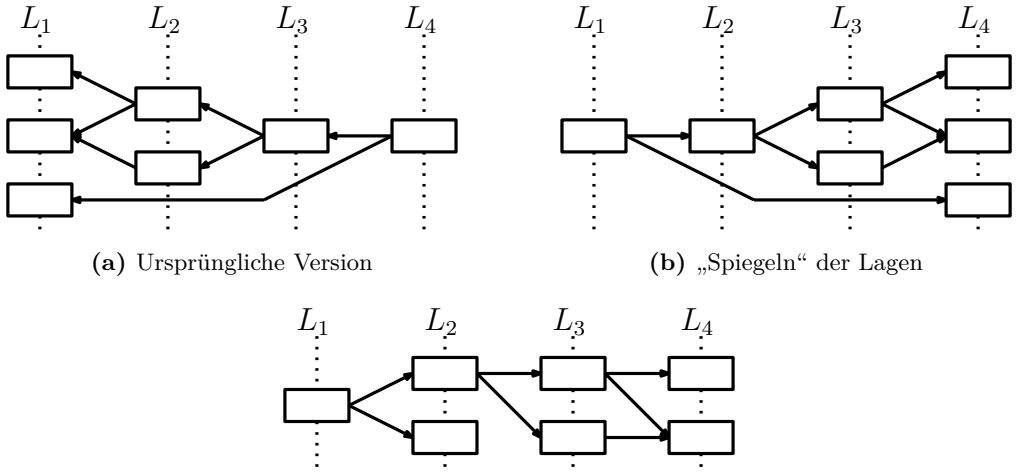
aufgrund der Struktur von Rechengraphen in vielen Fällen die ersten Lagen sehr viel mehr Knoten enthalten würden als spätere Lagen. Das liegt daran, dass einerseits zu einem frühen Zeitpunkt in der Berechnung oft viele verschiedene korrekte Rechenschritte möglich sind und andererseits von den Schülern gerade hier oft viele verschiedene Fehler gemacht werden. Für die maximale Lagenbreite B wird folgender Wert benutzt:

$$B = \left\lceil \frac{\text{Anzahl der Knoten im Graph}}{\text{Länge des längsten Wegs im Graph}} \right\rceil$$

Die Anzahl der minimal benötigten Lagen entspricht der Länge des längsten Wegs im Graph. Damit sorgt diese Definition der maximalen Lagenbreite dafür, dass in jeder Lage ungefähr gleich viele Knoten enthalten sind.

4.4.1. Coffman-Graham

Für die Lagenzuordnung wurden zwei verschiedene Algorithmen verwendet. Zuerst wurde das Verfahren von Coffman und Graham (siehe 3.2.3) getestet, bei dem die Knotengewichte keine Rolle spielen. Der ursprüngliche Algorithmus wurde dabei leicht angepasst (siehe Alg. 8). Das ist nötig, weil der Algorithmus normalerweise die Senken des Graphen in die ersten Lagen setzt (Abb. 4.4a). Zeichnet man die Lagen von links nach recht, würden fast alle Kanten von rechts nach links zeigen. Durch einfaches „Spiegeln“ der Lagen kann man das Problem leicht beheben. Das Problem ist aber, dass dann die vorderen Lagen oft wenig Knoten enthalten (Abb. 4.4b). Der Algorithmus wurde deshalb so angepasst, dass er von Beginn an von links nach rechts zeichnet, damit die vorderen Lagen immer möglichst voll besetzt sind (Abb. 4.4c). Dazu wird beim Verteilen der Labels von den Senken aus begonnen und beim Zuordnen der Lagen wird dann mit den Quellen begonnen.



(a) Ursprüngliche Version

(b) „Spiegeln“ der Lagen

(c) Angepasste Version

Abb. 4.4.: Coffman-Graham

Algorithmus 8: angepasste Lagenzuordnung nach Coffman und Graham

Eingabe : gerichteter Graph $G = (V, E)$, maximale Knoten pro Lage B
Ausgabe : gültige Lagenzuordnung mit maximaler Lagenbreite B

anfangs sind alle Knoten ohne Label
for $i = 1$ **to** $|V|$ **do**
    wähle einen Knoten v ohne Label so, dass alle Nachfolger von v ein Label haben und die Menge $\{\pi(u) \mid (v, u) \in E\}$ minimal ist
    $\pi(v) = i$
$k = 1$; $L_1 = \emptyset$; $U = \emptyset$
while $U \neq V$ **do**
    $u =$ Knoten in $V \setminus U$, so dass $N^-(u) \subseteq U$ gilt und $\pi(u)$ maximiert wird
    if $L_k < B$ und für jeden Knoten $w \in N^-(u)$ gilt: $w \in L_1 \cup L_2 \cup \ldots \cup L_{k-1}$ **then**
        $L_k = L_k \cup \{u\}$
    else
        $k = k + 1$; $L_k = \{u\}$
    $U = U \cup \{u\}$

4.4.2. PCMPS mit Knotengewichten

Neben dem Algorithmus von Coffman und Graham wurde noch ein anderes Verfahren für die Lagenzuordnung getestet. Hierzu wurde der PCMPS-Algorithmus (Kap. 3.2.2) angepasst, um die Knotengewichte zu beachten. Die Anpassung besteht darin, dass nicht mehr das erste Element aus der Liste mit Knoten, die einer Lage zugeordnet werden können, ausgewählt wird, sondern der Knoten mit dem meisten Gewicht. Diese Knoten sind wichtiger und haben oft viele Nachfolger, weshalb sich durch diese Änderung die Lagenzuordnung verbessern könnte. Algorithmus 9 zeigt diese angepasste Version.

In Kapitel 5.5 wird untersucht, wie gut die beiden Algorithmen beim Zeichnen von Rechengraphen funktionieren. Zudem wurde hier auch der Algorithmus getestet, der eine Lagenzuordnung mit minimaler Höhe findet (siehe Kap. 3.2.1). Dabei stellte sich heraus, dass alle drei Algorithmen im Endeffekt ähnlich gut funktionieren. Das liegt aber auch daran, dass im weiteren Verlauf des Zeichenalgorithmus bereits entfernte Knoten wieder eingefügt werden können, was in Kapitel 4.8 näher erklärt wird.

4.5. Reduzierung der Größe des Graphen

Die Lagenzuordnung für alle Knoten des Graphen ist nun bekannt. Damit lässt sich jetzt ungefähr abschätzen, wie hoch und breit die Lagen sind und ob der gesamte Graph auf die Zeichenfläche passt. Genau lässt sich das allerdings noch nicht sagen, da sich der Graph durch das Einfügen der Dummyknoten noch verändern wird. Passen jetzt schon alle Knoten auf die Zeichenfläche, kann direkt mit dem Einfügen der Dummyknoten und der Reduzierung von Kantenkreuzungen fortgefahren werden. Im Allgemeinen brauchen

Algorithmus 9: Lagenzuordnung durch PCMPS mit Knotengewichten

Eingabe : gerichteter Graph $G = (V, E, w)$ mit Knotengewichten, maximale Knoten pro Lage B

Ausgabe : gültige Lagenzuordnung mit maximaler Breite B

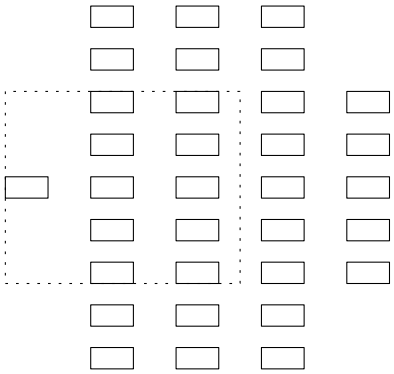
```
1  U = ∅; k = 1; L_1 = ∅
2  füge alle Quellen von G in U ein
3  while U ist nicht leer do
4      u = Knoten in U, so dass w(u) maximal ist
5      if |L_k| < B und für jeden Knoten w ∈ N⁻(u) gilt: w ∈ L_1 ∪ L_2 ∪ ... ∪ L_{k-1}
       then
6          L_k = L_k ∪ {u}
7      else
8          k = k + 1; L_k = {u}
9      foreach v ∈ N⁺(u) do
10         if v ∉ U und alle Vorgänger von v sind einer Lage zugeordnet then
11             U = U ∪ {v}
12     entferne u aus U
```

die Rechengraphen jedoch deutlich mehr Platz als die Zeichenfläche zur Verfügung stellt. Das heißt es gibt mehr Lagen als aufgrund der Breite der Zeichenfläche gezeichnet werden können und es gibt Lagen, die mehr Knoten enthalten als auf die Zeichenfläche passen. Deshalb folgen jetzt zwei Schritte, die Knoten aus dem Graph entfernen bis er auf die Zeichenfläche passt. Diese Schritte sind im Sugiyama-Framework nicht enthalten und wurden speziell für das Zeichnen von Rechengraphen eingebaut.

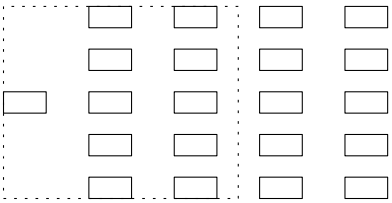
Abbildung 4.5 zeigt leicht vereinfacht, wie das Entfernen von Knoten in den meisten Fällen abläuft. Nach der Lagenzuordnung sieht die Situation für große Graphen normalerweise ähnlich aus wie in Abbildung 4.5a. Die erste Lage enthält nur den Startknoten, weil dieser die einzige Quelle des Graphen ist. In den folgenden Lagen wird die maximale Anzahl von Knoten pro Lage häufig erreicht, wobei dies von der Struktur des Graphen abhängt. Es kann hier durchaus vorkommen, dass manche Lagen nicht voll besetzt sind. Die letzte Lage hingegen hat fast nie die maximale Anzahl von Knoten.

Im ersten Schritt werden nun aus jeder Lage Knoten entfernt, bis die Höhe aller Lagen kleiner ist als die Höhe der Zeichenfläche. Dabei ist auch zu beachten, dass die Knoten, anders als beim einfachen Fall in der Abbildung, unterschiedlich groß sein können. Nach diesem Schritt hat man eine Situation wie in Abbildung 4.5b. Die Lagen sind alle niedrig genug, aber der Graph passt nicht auf die Zeichenfläche, weil es noch zu viele Lagen gibt. Deshalb werden nun im zweiten Schritt komplette Lagen entfernt, bis alle Knoten auf die Zeichenfläche passen (siehe Abb. 4.5c).

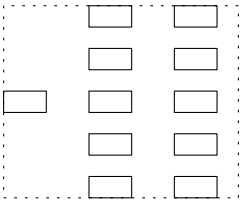
Der Grund dafür, dass zuerst die Knoten aus den Lagen entfernt werden, liegt darin, dass dadurch im zweiten Schritt die tatsächliche Wichtigkeit der Lagen besser einschätzbar ist. Angenommen man hat einen Fall wie in Abbildung 4.6. Es gibt zwei Lagen, von



(a) Graph nach Lagenzuordnung



(b) Höhe der Lagen wurde reduziert



(c) Lagen wurden entfernt

Abb. 4.5.: Reduzierung der Größe des Graphen

denen nur eine auf die Zeichenfläche passt. Beide Lagen haben ein Gesamtgewicht von acht. Somit ist nicht klar, welche Lage entfernt werden soll. Entfernt man nun zuerst die Knoten, wird klar, dass die zweite Lage schwerer ist und somit die erste Lage entfernt werden sollte.

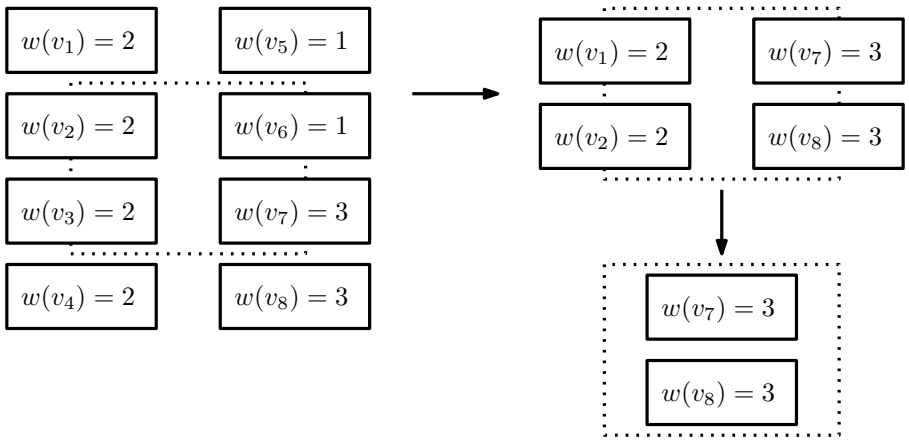


Abb. 4.6.: Knoten werden zuerst entfernt

Die Frage ist nun, wie die Knoten und Lagen ausgewählt werden, die entfernt werden sollen. Hier gibt es verschiedene Möglichkeiten, die in den nächsten beiden Unterkapiteln erläutert werden.

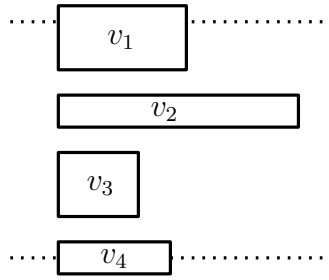
4.5.1. Entfernen von Knoten

Um zu entscheiden, welche Knoten entfernt werden sollen, wird für jeden Knoten v ein Wert $I(v)$ berechnet, der seine Wichtigkeit beschreibt. Desto kleiner dieser Wert, desto unwichtiger der Knoten. Mit Hilfe dieses Werts kann dann leicht entschieden werden, welcher Knoten als Nächstes entfernt werden soll.

Es gibt nun unterschiedliche Möglichkeiten, wie man diese Knotenwichtigkeit definieren kann. Einige grundlegende Eigenschaften sollte die Knotenwichtigkeit jedoch immer haben. Sie sollte größer werden, je schwerer der Knoten ist. Das macht Sinn, da es ja gerade das Ziel des Zeichenalgorithmus ist, eine Menge von Knoten zu zeichnen, die möglichst viel Gewicht hat. Außerdem sollte die Wichtigkeit sinken, je höher ein Knoten ist. So können Knoten, die zwar vielleicht ein wenig schwerer sind als andere, dabei aber deutlich mehr Platz benötigen, vernünftig bewertet werden. Diese beiden Anhaltspunkte führen zu einer ersten einfachen Definition der Knotenwichtigkeit.

$$I_1(v) = \frac{w(v)}{h(v)}$$

Diese Definition hat jedoch das Problem, dass die Breite der Knoten außer Acht gelassen wird. Denn in manchen Fällen kann durch das Entfernen eines Knotens nicht nur die Höhe der Lage verringert werden, sondern auch deren Breite. Dadurch kann es dann möglich werden, dass eine ganze zusätzliche Lage auf die Zeichenfläche passt.



	$w(v)$	$h(v)$	$b(v)$	I_1	$I = \frac{w}{h \cdot b}$	I_2	$I_3 \cdot 1000$
v_1	100	40	80	2,5	$\approx 0,031$	2,5	12,12
v_2	120	20	150	6	0,04	1,33	8,36
v_3	110	40	50	2,75	0,055	2,75	13,33
v_4	300	20	70	15	$\approx 0,214$	15	57,14

Abb. 4.7.: Unterschiedliche Kriterien beim Entfernen von Knoten

Angenommen man hat einen Fall wie in Abbildung 4.7. Hier muss noch ein Knoten entfernt werden, damit die restlichen auf die Zeichenfläche passen. Der Knoten v_2 ist deutlich breiter als alle anderen und nur etwas schwerer als die Knoten v_1 und v_3. Nach der bisherigen Definition würde der Knoten v_1 entfernt werden, obwohl es hier sinnvoller wäre, den Knoten v_2 zu entfernen, weil dadurch die Breite der Lage deutlich verkleinert werden könnte. Es ist nun naheliegend, statt der Breite eines Knotens die Fläche eines Knotens zu benutzen. Doch auch das hilft hier nicht. Wenn man die Knotenwichtigkeit als Gewicht pro Fläche betrachtet, würde immer noch der Knoten v_1 entfernt werden.

Um in einem solchen Fall den „richtigen“ Knoten zu entfernen, muss darauf geachtet werden, um wie viel sich die Lagenbreite verringert, wenn der Knoten entfernt würde. Das führt zu folgender Definition:

$$I_2(v) = \frac{w(v)}{h(v) + \text{Unterschied der Lagenbreite, wenn } v \text{ entfernt würde}}$$

Ein Problem der beiden obigen Definitionen ist es, dass die tatsächlich gesparte Höhe nicht wie in der Definition proportional zur Höhe des entfernten Knotens ist. Das kommt daher, dass durch das Entfernen eines Knotens immer auch einmal der Mindestabstand zwischen zwei Knoten gespart wird. In Abbildung 4.7 ist der Knoten v_3 nur halb so hoch wie der Knoten v_4. Die bisherigen Definitionen würden also davon ausgehen, dass durch das Entfernen von v_3 doppelt so viel Höhe eingespart wird wie durch das Entfernen von v_4. Tatsächlich wird hier aber nur ca. 1,6-mal so viel Höhe eingespart, wobei dieser Faktor von dem festgelegten Mindestabstand abhängt. Das führt zu folgender Definition der Knotenwichtigkeit, die sowohl auf die tatsächliche Höhenersparnis achtet als auch auf die Lagenbreite, die sich durch das Entfernen eines Knotens verringern könnte:

$$I_3(v) = \frac{w(v)}{\text{aktuelle Lagenfläche} \; - \; \text{Lagenfläche ohne } v}$$

Die Lagen werden in diesem Schritt von links nach rechts durchlaufen. Das macht Sinn, da durch das Entfernen von Knoten in den vorderen Lagen auch Knoten in den hinteren Lagen wegfallen können, wenn sie keine eingehenden Kanten mehr haben. So kann es passieren, dass dann in den hinteren Lagen weniger Knoten entfernt werden müssen. Bei der Verwendung von I_2 oder I_3 muss außerdem darauf geachtet werden, dass die Werte nach dem Entfernen eines Knotens neu berechnet werden müssen, da die Wichtigkeit hier von den anderen Knoten abhängt. Wenn I_1 verwendet wird genügt es hingegen, die Werte nur einmal zu berechnen.

In Kapitel 5.6 wird untersucht, wie sich die unterschiedlichen Definitionen auswirken, wobei hier kein Unterschied zwischen den Wichtigkeiten zu erkennen ist. Es wurde außerdem überprüft, ob es sinnvoll ist, das Quadrat des Gewichts zu verwenden, um das Gewicht stärker zu betonen. Doch auch das macht in der Praxis keinen Unterschied.

4.5.2. Entfernen von Lagen

Nachdem im letzten Schritt für alle Lagen die Höhe so weit verringert wurde, dass die Lagen auf die Zeichenfläche passen, muss nun noch entschieden werden, welche Lagen letztendlich gezeichnet werden sollen und welche entfernt werden. Das Prinzip ist das Gleiche wie beim Entfernen von Knoten. Jede Lage bekommt einen Wert zugeordnet, der ihre Wichtigkeit beschreibt. Dann werden so lange einzelne Lagen entfernt, bis alle verbleibenden Lagen auf die Zeichenfläche passen. Die Lagen-Wichtigkeit einer Lage L mit Breite $b(L)$ ist folgendermaßen definiert:

$$I(L) = \frac{\sum_{v \in L} w(v)}{b(L)}$$

Hier kann es passieren, dass durch das Entfernen einer Lage Knoten in anderen Lagen wegfallen. Wenn das passiert, ändern sich die Wichtigkeiten der entsprechenden Lagen und müssen neu berechnet werden. Auch hier wurden Tests mit dem Quadrat des Lagengewichts durchgeführt, doch wie schon bei den Knotenwichtigkeiten konnte dabei kein Unterschied festgestellt werden.

4.6. Reduzierung von Kantenkreuzungen

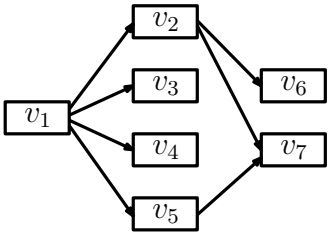
Durch das Entfernen von Knoten und Lagen im letzten Schritt wurde die Größe des Graphen so weit reduziert, dass die jetzt noch vorhandenen Knoten auf die Zeichenfläche passen. In den folgenden Schritten geht es nun vor allem darum, diese Knoten so anzuordnen, dass die entstehende Zeichnung für den Betrachter möglichst leicht verständlich ist.

Dazu wird zunächst die Anzahl von Kantenkreuzungen reduziert. Das Vorgehen ist hier das Gleiche wie im Sugiyama-Framework. Zuerst werden für lange Kanten die Dummyknoten eingefügt, so dass nur noch Kanten zwischen benachbarten Lagen existieren. Da beim Zeichnen von Rechengraphen der Platz beschränkt ist, tritt hier jedoch ein Problem auf. Durch das Einfügen der Dummyknoten werden die Lagen höher, wodurch manche Lagen möglicherweise nicht mehr auf die Zeichenfläche passen. Dieses Problem wird dadurch gelöst, dass am Ende des Algorithmus noch einmal die Höhe aller Lagen so weit wie nötig reduziert wird (siehe Kapitel 4.9).

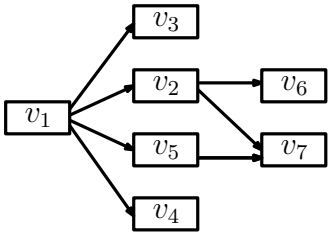
Nachdem die Dummyknoten eingefügt sind, wird die Anzahl der Kreuzungen mit den verschiedenen Verfahren, die in Kapitel 3.3 vorgestellt wurden, minimiert. Beim Zeichnen von Rechengraphen kann jedoch sinnvoll sein, nicht die Anzahl der Kreuzungen zu reduzieren, sondern das Gewicht der Kreuzungen. Das führt dann dazu, dass wahrscheinlich mehr Kreuzungen im Graph sind, die wichtigen Kanten jedoch ohne Kreuzungen gezeichnet werden und so die Zeichnung letztendlich besser lesbar ist. Um das Gewicht der Kreuzungen zu minimieren, wird die Adjacent-Exchange-Heuristik bzw. die darin verwendeten Kreuzungszahlen leicht modifiziert. Eine Kreuzungszahl c_{uv} gibt dann nicht mehr an, wie viele Kreuzungen die zu u und v inzidenten Kanten verursachen, sondern das Gewicht der an Kreuzungen beteiligten Kanten.

Außerdem wird vor dem Anwenden der Heuristiken noch ein weiterer Schritt ausgeführt. Dieser hat zwei Ziele: Zum einen soll damit eine gute Startsortierung der Knoten gefunden werden, damit die Heuristiken besser funktionieren. Zum anderen soll dafür gesorgt werden, dass wichtige Knoten, die viele Nachfolger haben, möglichst in der Mitte der Lagen gezeichnet werden. Abbildung 4.8 zeigt, was damit gemeint ist. Beide Zeichnungen haben keine Kantenkreuzungen. Trotzdem ist die Zeichnung in Abbildung 4.8b besser, weil die Knoten v_2 und v_5 in der Mitte der Lage platziert sind und die Knoten v_3 und v_4 ohne Nachfolger außen liegen.

Das Verfahren funktioniert folgendermaßen: Zuerst wird der Startknoten aus dem Graph entfernt. Dadurch zerfällt der Graph wahrscheinlich in mehrere schwache Zusammenhangskomponenten. Anschließend werden die Knoten innerhalb der Lagen so umsortiert, dass die Knoten in der größten Zusammenhangskomponente jeweils in der



(a) Schlechte Zeichnung ohne Kreuzungen



(b) Gute Zeichnung ohne Kreuzungen

Abb. 4.8.: Unterschiedliche Zeichnungen ohne Kreuzungen

Mitte ihrer Lage positioniert sind. Die restlichen Knoten werden dann innerhalb ihrer Lagen so darüber und darunter platziert, dass die Knoten weiter außen liegen, je kleiner die Zusammenhangskomponente ist, in der sie enthalten sind. Am Ende wird der Startknoten und die inzidenten Kanten wieder eingefügt. Das Verfahren hat außerdem zur Folge, dass man eine gute Startsortierung für die Heuristiken zur Kreuzungsminimierung gefunden hat, da zwischen Kanten, die in unterschiedlichen Zusammenhangskomponenten liegen, keine Kreuzungen mehr auftreten und die Heuristiken jetzt nur noch innerhalb der Komponenten die Positionen tauschen müssen.

4.7. Entfernen von Kanten

Obwohl im letzten Schritt die Anzahl von Kantenkreuzungen reduziert wurde, kann es in ungünstigen Fällen immer noch viele Kreuzungen zwischen zwei Lagen geben. Das passiert, wenn zwischen den Lagen sehr viele Kanten verlaufen oder wenn die verwendete Heuristik in diesem Fall nicht gut funktioniert. Deshalb folgt jetzt ein Schritt, der die Kantenkreuzungen zwischen zwei Lagen nochmals reduziert. Anders als im letzten Schritt wird hier jedoch der Graph verändert, indem Kanten entfernt werden, was zu einem Informationsverlust führt. Um diesen möglichst gering zu halten, wird ähnlich wie beim Entfernen von Knoten und Lagen eine Wichtigkeit für jede Kante definiert. Sie lautet folgendermaßen:

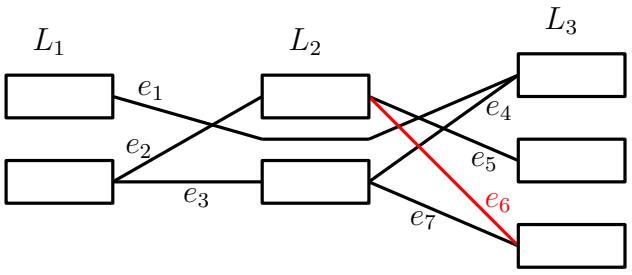
$$I(e) = \frac{w(e)}{\text{Summe des Gewichts der Kanten, die } e \text{ schneiden}}$$

Diese Definition der Wichtigkeit hat zwei Hintergründe. Zum einen sollen Kanten mit hohem Gewicht nach Möglichkeit im Graph enthalten bleiben, weshalb die Wichtigkeit mit dem Gewicht wächst. Andererseits sollen Kanten mit hohem Gewicht nur von wenigen anderen Kanten gekreuzt werden, damit der Betrachter den wichtigen Kanten leichter folgen kann. Deshalb wird die Wichtigkeit einer Kante kleiner, je schwerer die Kanten sind, die sie schneidet. Hat eine Kante e keine Kreuzungen mit anderen Kanten gilt $I(e) = \infty$. Das macht Sinn, da das Löschen dieser Kante die Anzahl von Kreuzungen nicht reduzieren würde und sie somit keinesfalls entfernt werden soll.

Außerdem ist zu beachten, dass hier die enthaltenen Dummyknoten keine Rolle spielen

und eine Kante somit über mehrere Lagen hinweg verlaufen kann. Es werden also auch die Gewichte von Kanten hinzuaddiert, die die Kante e zwischen anderen Lagen schneiden als den beiden aktuell betrachteten Lagen. Das ist sinnvoll, weil durch das Entfernen der Kante auch zwischen anderen Lagen Kreuzungen reduziert werden, was mit dieser Definition der Wichtigkeit abgedeckt wird.

Ab wann zwischen zwei Lage „zu viele“ Kantenkreuzungen vorhanden sind, lässt sich unterschiedlich definieren. Eine sinnvolle Schranke ist zum Beispiel das Minimum der Anzahl von Knoten in den beiden Lagen. Eine andere Möglichkeit wäre, einen konstanten Wert zu wählen oder sogar Kantenkreuzungen komplett zu verbieten.



e	$w(e)$	$I(e)$
e_1	100	$\frac{100}{100+40+60} = 0.5$
e_2	100	–
e_3	60	–
e_4	35	$\frac{35}{40+60} = 0.35$
e_5	60	$\frac{60}{100+35} \approx 0.44$
e_6	40	$\frac{40}{100+35} \approx 0.30$
e_7	20	∞

Abb. 4.9.: Beispiel für das Entfernen von Kanten

Abbildung 4.9 zeigt ein Beispiel für das Entfernen von Kanten. Zwischen den Lagen L_1 und L_2 gibt es nur eine Kreuzung, weshalb hier keine Kanten entfernt werden müssen. Zwischen den Lagen L_2 und L_3 gibt es hingegen vier Kreuzungen. Angenommen die Schranke ist das Minimum der Knoten in den benachbarten Lagen. Dann sind in diesem Fall nur zwei Kreuzungen erlaubt und es muss eine Kante entfernt werden. Die Tabelle zeigt die Gewichte und die daraus folgenden Wichtigkeiten der Kanten. Für die Kanten e_2 und e_3 müssen keine Wichtigkeiten berechnet werden, weil sie nicht zwischen L_2 und L_3 verlaufen. In diesem Fall würde die Kante e_6 entfernt werden, obwohl sie nicht die Kante mit dem geringsten Gewicht ist. Das liegt daran, dass sie im Gegensatz zur leichteren Kante e_4 eine Kreuzung mit der schweren Kante e_1 hat.

Nach diesem Schritt kann dann noch einmal Kreuzungsminimierung durchgeführt werden, wodurch sich in vielen Fällen noch einmal Kreuzungen sparen lassen. Man sollte sich jedoch bewusst sein, dass dadurch theoretisch zwischen zwei Lagen wieder mehr Kreuzungen auftreten können als der Schrankenwert zulässt, obwohl die Gesamtzahl von Kreuzungen im Graph sinkt. Dieser Fall ist beim Testen mit unterschiedlichen Eingaben jedoch nie aufgetreten.

4.8. Einfügen von entfernten Knoten und Kanten

Zum jetzigen Zeitpunkt des Algorithmus ist bereits die Grundstruktur der späteren Zeichnung vorgegeben. Die Knoten wurden den Lagen zugeordnet und innerhalb der Lagen so sortiert, dass möglichst wenig Kreuzungen vorhanden sind. Durch das Entfernen von Knoten, Lagen und Kanten in den vorherigen Schritten ist es aber möglich, dass der vorhandene Platz auf der Zeichenfläche nicht komplett ausgenutzt wird. Deshalb folgt

jetzt noch ein Schritt, in dem Knoten, die zuvor aus dem Graph gelöscht wurden, wieder eingefügt werden, wenn noch Platz auf der Zeichenfläche ist. Auf diese Weise können Knoten zu vorhandenen Lagen hinzugefügt werden oder es können sogar neue Lagen eingefügt werden.

Damit ein Knoten eingefügt werden kann, muss mindestens einer seiner Vorgänger aus der Eingabe noch im Graph vorhanden sein, denn es macht keinen Sinn, Knoten ohne Vorgänger im Graph zu haben (abgesehen vom Startknoten). Die Lage, in die ein Knoten eingefügt werden kann, hängt dabei von seinen Vorgängern und Nachfolgern ab. Sie muss größer als als das Maximum der Lagen aller Vorgänger und kleiner als das Minimum der Lagen aller Nachfolger sein. Wenn solch eine Lage existiert und in dieser noch Platz ist, kann der Knoten wieder eingefügt werden. Hier muss aber auch noch darauf geachtet werden, ob sich durch das Einfügen eines Knotens die Lage verbreitern würde. Ist das der Fall, muss sichergestellt werden, dass auch nach dem Einfügen noch alle Lagen auf die Zeichenfläche passen.

Die Reihenfolge, in der die entfernten Knoten betrachtet werden, hängt dabei ähnlich wie beim Entfernen von der Wichtigkeit der Knoten ab. Einerseits sollen schwere Knoten bevorzugt eingefügt werden, weil durch sie der Informationsgewinn am größten ist. Andererseits sollen kleine Knoten ausgewählt werden, weil dadurch möglicherweise mehr Knoten eingefügt werden können. Anders als beim Entfernen von Knoten kann hier jedoch nicht darauf geachtet werden, wie sich die Breite oder die Fläche der entsprechenden Lagen durch das Einfügen der Knoten verändert, weil beim Sortieren der Knoten noch nicht klar ist, in welche Lage die Knoten eingefügt werden. Die Wichtigkeit eines Knotens v ist deshalb folgendermaßen definiert:

$$I(v) = \frac{w(v)}{h(v)}$$

Durch das Einfügen von Knoten in diesem Schritt können Kanten hinzukommen, die Knoten verbinden, die weiter als eine Lage voneinander entfernt sind. Außerdem können durch das Hinzufügen neue Kantenkreuzungen entstehen. Deshalb müssen jetzt einige zuvor schon ausgeführte Schritte wiederholt werden.

Zuerst werden Dummyknoten eingefügt, wo sie nötig sind. Anschließend wird die Anzahl von Kreuzungen reduziert und es wird getestet, ob die maximale Anzahl von Kantenkreuzungen zwischen zwei Lagen irgendwo überschritten wird. Ist dies der Fall, werden Kanten wie im letzten Kapitel beschrieben entfernt, bis die Anzahl von Kreuzungen genügend reduziert wurde. Anschließend wird noch einmal Kreuzungsminimierung durchgeführt.

4.9. Festlegen der Knotenpositionen

Mit den bisher durchgeführten Schritten wurde festgelegt, welche Knoten gezeichnet werden sollen und welcher Lage sie zugeordnet sind. Allerdings ist erst zum jetzigen Zeitpunkt klar wie die Knoten innerhalb einer Lage in der Ausgabe geordnet sein werden. Das heißt es kann auch erst jetzt die tatsächliche Höhe einer Lage berechnet werden. Das

liegt daran, dass es sinnvoll ist, den Abstand zwischen zwei beliebigen, nebeneinander liegenden Knoten u und v davon abhängig zu machen, ob u und v echte Knoten oder Dummyknoten sind. Es lassen sich hier vier Fälle unterscheiden, von denen abhängt, wie nah die Knoten beieinander gezeichnet werden:

1. Die Knoten u und v sind echte Knoten: Hier sollte der Abstand d_1 zwischen den Knoten am größten sein, um die Knoten gut unterscheiden zu können.

2. Der Knoten u ist echt, v ist ein Dummyknoten: Der Abstand d_2 zwischen u und v sollte kleiner sein als im ersten Fall, weil sonst zu große Lücken entstehen (vor allem wenn nur ein Dummyknoten zwischen zwei echten Knoten liegt).

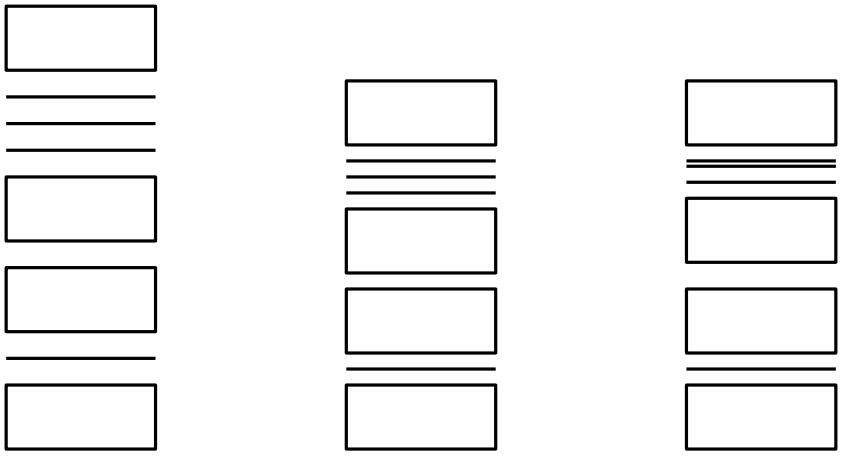
3. Die Knoten u und v sind Dummys: Der Abstand d_3 sollte kleiner sein als im ersten Fall und kleiner oder gleich sein als im zweiten Fall.

4. Die Knoten u und v sind Dummys, deren echte Kanten den gleichen Start- oder Zielknoten haben: Hier ist es sinnvoll den Abstand d_4 deutlich kleiner zu wählen als im dritten Fall. Dadurch wird dem Betrachter sofort klar, dass die beiden Knoten (bzw. Kanten) zueinander gehören und es ist leichter, parallel verlaufenden Kanten zu folgen.

Abbildung 4.10 zeigt, wie sich die unterschiedlichen Abstände auf die Zeichnung auswirken und warum für die verschiedenen Fälle unterschiedliche Abstände gewählt werden sollten. In Abbildung 4.10a wurden für alle Fälle gleiche Abstände verwendet, wobei diese relativ groß sind ($d_1 = d_2 = d_3 = d_4 = 10$). Hier ist der Abstand für die beiden nebeneinander liegenden echten Knoten gut gewählt, allerdings sind die Dummyknoten zu weit voneinander entfernt, so dass hier die Lage recht hoch ist und der verfügbare Platz nicht optimal ausgenutzt wird. In Abbildung 4.10b werden auch für alle Fälle die gleichen, aber kleinere Abstände benutzt ($d_1 = d_2 = d_3 = d_4 = 6$). Damit benötigen die Dummyknoten nicht so viel Platz, allerdings sind hier die echten Knoten zu nahe beieinander gezeichnet. Eine bessere Zeichnung mit unterschiedlichen Abständen zeigt Abbildung 4.10c. Hier wird der Abstand für zwei echte Knoten recht groß gewählt ($d_1 = 10$), während die Dummyknoten nahe beieinander gezeichnet werden ($d_2 = d_3 = 6$). Außerdem sieht man hier den vierten Fall, bei dem zwei Dummyknoten, deren echte Kanten den gleichen Start- oder Zielknoten haben, sehr nah beieinander gezeichnet werden ($d_4 = 2$).

Durch die unterschiedlichen Abstände ist also erst jetzt klar, wie hoch die einzelnen Lagen der Zeichnung tatsächlich sind. Hier ist es noch möglich, dass einzelne Lagen höher sind als die Zeichenfläche, was, wie bereits erwähnt, durch das Einfügen der Dummyknoten zustande kommt. Deshalb ist jetzt noch ein Schritt nötig, um zu garantieren, dass wirklich alle Knoten auf die Zeichenfläche passen. Um das zu erreichen werden noch einmal Knoten aus dem Graph entfernt. Hier wird nur auf echte Knoten geachtet, weil durch das Entfernen von Dummyknoten nur sehr wenig Platz gespart wird.

Das Verfahren funktioniert ähnlich wie das Entfernen von Knoten in Kapitel 4.5.1, als die Höhe der Lagen zum ersten Mal reduziert wurde. Wieder erhält jeder Knoten einen Wert, der seine Wichtigkeit beschreibt, und es werden so lange die unwichtigsten



(a) große Abstände **(b)** kleine Abstände **(c)** unterschiedliche Abstände

Abb. 4.10.: Unterschiedliche Abstände zwischen nebeneinander liegenden Knoten

Knoten einer Lage entfernt, bis alle Knoten der Lage auf die Zeichenfläche passen. Die Definition dieser Wichtigkeit ist dabei einfacher als in Kapitel 4.5.1, weil die Lagenbreite keine Rolle mehr spielt und deshalb nur auf die Höhe der Knoten geachtet werden muss. Die Wichtigkeit eines Knotens v ist deshalb folgendermaßen definiert:

$$I(v) = \frac{w(v)}{h(v)}$$

Die Lagen werden dabei von links nach rechts betrachtet, da durch das Entfernen von Knoten in frühen Lagen möglicherweise Knoten in späteren Lagen wegfallen und das Problem dann dort gar nicht mehr auftritt. Wenn dann alle Lagen auf die Zeichenfläche passen, werden die Knoten mit den festgelegten Abständen voneinander so positioniert, dass ober- und unterhalb der Lage genauso viel Platz bis zum Rand der Zeichenfläche ist.

4.10. Reduzierung der Anzahl von Knicken

Zeichnet man den Rechengraph mit den im letzten Kapitel festgelegten Knotenpositionen und orthogonalen Kanten, gibt es ein kleines Problem. Echte Kanten, die über mehrere Lagen hinweg verlaufen und durch viele Dummyknoten dargestellt werden, haben sehr viele Knicke. Um dieses Problem zu lösen wurde eine Verfahren von Sander [San96] implementiert, das die Knotenpositionen so festlegt, dass jede echte Kante maximal vier Knicke hat. Dazu werden sogenannte *lineare Segmente* verwendet. Das Verfahren ändert dabei die Positionen der Knoten, die Reihenfolge der Knoten innerhalb einer Lage bleibt jedoch erhalten.

Definition 11. Ein lineares Segment s ist eine maximale Sequenz von Knoten $v_1, \ldots, v_k$, die folgende Eigenschaften erfüllt:

- $l(v_i) = l(v_{i+1}) - 1$ für $i = 1, \ldots, k-1$
- $|N^-(v_1)| \leq 1$, $|N^+(v_k)| \leq 1$, $N^+(v_1) = \{v_2\}$, $N^-(v_k) = \{v_{k-1}\}$
- $N^-(v_i) = \{v_{i-1}\} \wedge N^+(v_i) = \{v_{i+1}\}$ für $i = 2, \ldots, k-1$

Der Rechengraph wird nun in diese linearen Segmente aufgeteilt, wobei jeder Knoten zu höchstens einem linearen Segment gehört. Ziel ist es nun, alle Knoten innerhalb eines linearen Segments auf der gleichen Höhe zu zeichnen. Gelingt dies, so hat eine echte Kante e mit Startknoten u, Zielknoten v und den Dummyknoten $d_1, \ldots, d_k$ nur maximal vier Knicke. Das kommt daher, dass die Dummyknoten ein lineares Segment bilden und nur jeweils zwei Knicke in den Kanten (u, d_1) und (d_k, v) auftreten.

Damit es möglich ist, alle Knoten der linearen Segmente auf die jeweils gleiche Höhe zu setzen und dabei die Reihenfolge der Knoten innerhalb der Lagen beizubehalten, dürfen sich zwei lineare Segmente nicht kreuzen. Das muss beim Minimieren der Kantenkreuzungen sichergestellt werden.

Um den Algorithmus zum Festlegen der Knotenpositionen zu vereinfachen, werden Knoten, die in keinem linearen Segment enthalten sind, einem trivialen linearen Segment zugeordnet, das nur einen Knoten enthält. Damit lässt sich der ganze Rechengraph nun in disjunkte lineare Segment zerlegen. Um die Knotenpositionen festzulegen, wird jetzt ein sogenannter *Segmentgraph* SG $= (S, F)$ aufgebaut, dessen Knotenmenge aus den linearen Segmenten besteht. Die gerichteten Kanten im Segmentgraph geben an, wie die linearen Segmente relativ zueinander gezeichnet werden sollen. Eine Kante (s_1, s_2) gibt an, dass die Knoten im linearen Segment s_1 unterhalb von den Knoten in s_2 gezeichnet werden müssen. Algorithmus 10 zeigt, wie die Kanten in den Segmentgraph eingefügt werden. Das lineare Segment, in dem ein Knoten v liegt, wird dabei mit $s(v)$ bezeichnet.

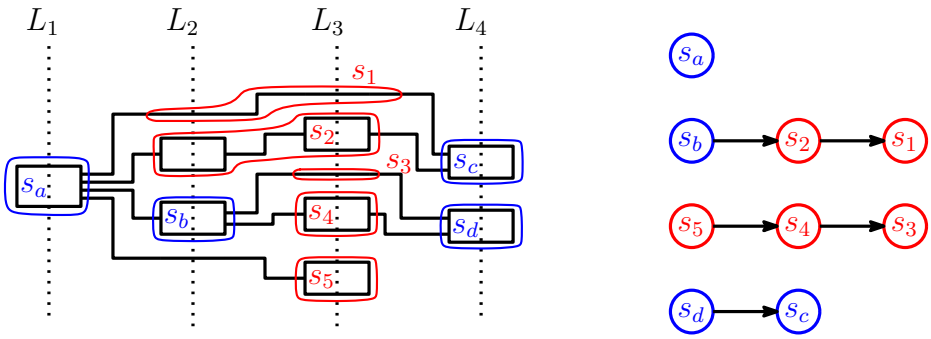
Algorithmus 10: Einfügen der Kanten in den Segmentgraph

Eingabe : Segmentgraph SG $= (S, F)$ mit $F = \emptyset$
Ausgabe : Segmentgraph mit eingefügten Kanten

```
1 foreach Lage L do
2     seien v_1, ..., v_k die Knoten in L, aufsteigend sortiert nach ihrer aktuellen
      Position in L
3     for i = 2 to k do
4         F = F ∪ {(s(v_{i-1}), s(v_i))}
5 return SG
```

Abbildung 4.11 zeigt ein Beispiel für einen Rechengraph und den dazugehörigen Segmentgraph. Die linearen Segmente sind mit $s_1, \ldots, s_5$ bezeichnet, die trivialen Segmente, die übrig bleiben, wenn man den Rechengraph in lineare Segmente zerlegt, sind mit $s_a, \ldots, s_d$ bezeichnet.

Der entstehende Segmentgraph ist azyklisch, da sich im Rechengraph keine linearen Segmente kreuzen. Damit ist es möglich eine topologische Sortierung T des Segmentgraphen zu erstellen. Eine *topologische Sortierung* eines Graphen ist eine Nummerierung



(a) Rechengraph **(b)** Segmentgraph

Abb. 4.11.: Rechengraph und dazugehöriger Segmentgraph

der Knoten, so dass alle Knoten verschiedene Nummern haben und jede Kante von einem Knoten mit kleinerer Nummer zu einem Knoten mit größerer Nummer verläuft. Mit dieser Sortierung von SG können die Knotenpositionen so festgelegt werden, dass alle Knoten eines linearen Segments auf der gleichen Höhe liegen und somit jede Kante maximal vier Knicke hat. Algorithmus 11 zeigt, wie beim Festlegen der Knotenpositionen grundsätzlich vorgegangen wird. In der Implementierung musste noch auf die Höhe der Knoten geachtet werden, damit auch lineare Segmente, in denen sowohl echte Knoten als auch Dummyknoten enthalten sind, ohne Knicke gezeichnet werden. Die „minimale y-Position“ in Zeile 4 des Algorithmus ist von drei Faktoren abhängig: von der Position des darunter liegenden Knotens, von den festgelegten Mindestabständen zwischen Knoten und von der Höhe des Knotens selbst.

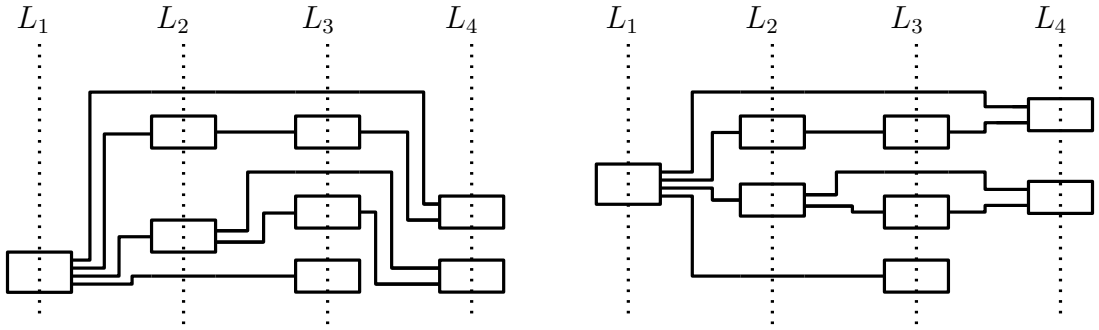
Algorithmus 11: Festlegen der Knotenpositionen

Eingabe : Segmentgraph SG $= (S, F)$
Ausgabe : Rechengraph mit festgelegten Knotenpositionen

```
1 seien s_1, ..., s_k die Knoten des Segmentgraphen, aufsteigend sortiert nach T
2 for i = 1 to k do
3     foreach v ∈ s_i do
4         y_min(v) = minimale y-Position von v
5     Y_min = max{y_min(v) | v ∈ s_i}
6     foreach v ∈ s_i do
7         y(v) = Y_min
```

Das Ergebnis des Verfahrens zeigt Abbildung 4.12a. Hier sind nun alle Knoten so weit wie möglich nach unten, aber soweit wie nötig nach oben gesetzt, damit alle linearen Segmente jeweils auf der gleichen Höhe liegen und alle echten Kanten maximal vier Knicke haben. Um eine schönere Zeichnung zu erhalten, folgt nun üblicherweise ein Schritt, der einzelne Knoten abhängig von den Positionen seiner Nachbarn verschiebt, und so die Zeichnung „ausbalanciert“. Abbildung 4.12b zeigt, wie der Graph dann aussehen könnte.

Dieser Schritt wurde jedoch nicht implementiert, da bereits abzusehen war, dass durch das Verfahren der Platzbedarf der Zeichnung zu groß wird. Im Beispielgraph wird die Zeichnung zwar nicht größer, bei realen Eingaben vergrößert sich die Zeichnung jedoch meist um das zwei- bis dreifache. Damit ist das Verfahren für das Zeichnen von Rechengraphen ungeeignet, weil dabei zu viel Platz verschwendet wird. Nur bei sehr kleinen Eingaben, die nicht die ganze Zeichenfläche benötigen, könnte das Verfahren eingesetzt werden.



(a) Ergebnis vor dem Ausbalancieren **(b)** Ergebnis nach dem Ausbalancieren

Abb. 4.12.: Ergebnisse nach der Knickreduzierung

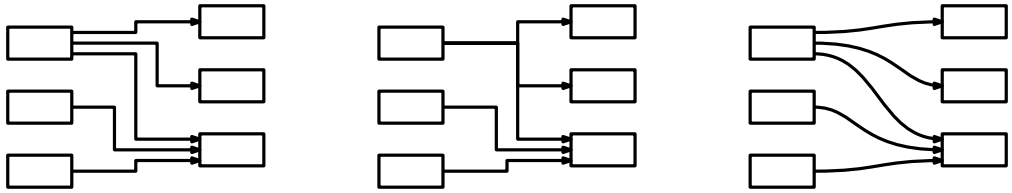
4.11. Zeichnen der Kanten

Zum jetzigen Zeitpunkt des Algorithmus ist die Ausgabezeichnung bereits zu einem großen Teil festgelegt. Es wurde entschieden, welche Knoten in der Zeichnung enthalten sind und wo sie gezeichnet werden sollen. Was jetzt noch fehlt sind die Kanten, wobei sich hier zunächst die Frage nach dem generellen Zeichenstil stellt.

Da beim Zeichnen von Rechengraphen üblicherweise orthogonale Kanten verwendet werden, wurden zunächst zwei Varianten dieses Stils implementiert und getestet. In der ersten Variante verlässt jede Kante ihren Startknoten auf einer anderen Höhe als die übrigen ausgehenden inzidenten Kanten und es werden alle Kanten einzeln gezeichnet (Abb. 4.13a). In der zweiten Variante verlassen alle ausgehenden eines Knotens diesen auf der selben Höhe. Sie werden also anfangs aufeinander gezeichnet und trennen sich erst im weiteren Kantenverlauf (Abb. 4.13b). Zuletzt wird noch ein dritter Kantenstil vorgestellt, der beim Zeichnen von Rechengraphen bisher nicht eingesetzt wurde. Hier werden die Kanten als Kurven dargestellt (Abb. 4.13c).

4.11.1. Allgemeines zum Zeichnen der Kanten

Bevor näher auf die einzelnen Kantenstile eingegangen wird, folgen nun noch einige allgemeine Informationen zum Zeichnen von Kanten, die nichts mit den unterschiedlichen Stilen an sich zu tun haben.



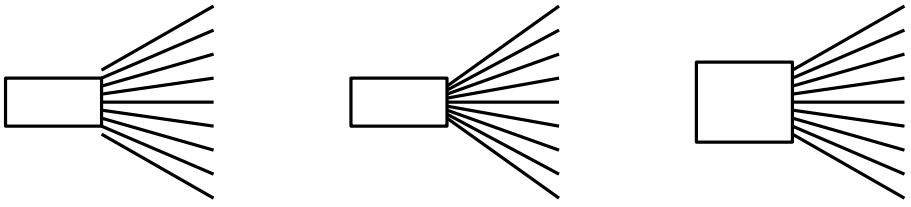
(a) Mehrere Ausgangsports **(b)** Einzelner Ausgangsport **(c)** Kanten als Kurven

Abb. 4.13.: Unterschiedliche Kantenstile

Vor dem Zeichnen der Kanten ist es zunächst nötig, die Positionen zu bestimmen, an denen eine Kante ihren Startknoten verlässt und an ihrem Zielknoten ankommt. Die Position, an der eine Kante e ihren Startknoten verlässt, wird als Startport $p_s(e)$ bezeichnet, die Position, an der eine Kante an ihrem Zielknoten ankommt, wird als Zielport $p_z(e)$ der Kante bezeichnet. Die Werte von $p_s(e)$ und $p_z(e)$ geben dabei nur die y-Koordinate des Punkts an, wo die Kante am Knoten ankommt oder den Knoten verlässt. Das ist ausreichend, da die x-Koordinaten der Ports durch die Positionen und Breiten der Knoten eindeutig festgelegt sind.

Zunächst muss die Reihenfolge der Ports an einem Knoten festgelegt werden. Das ist relativ einfach, da die Nachbarn in der daneben liegenden Lage geordnet sind. Die Reihenfolge dieser Nachbarn bestimmt dann die Reihenfolge der Ports. Damit ist garantiert, dass sich geradlinig gezeichnete Kanten, die einen gemeinsamen Start- oder Zielknoten haben, nicht schneiden. Die konkreten Ports werden dann so berechnet, dass sie einen festgelegten Abstand voneinander haben. Für Zielports ist dieser Abstand üblicherweise größer als bei Startports, weil hier die Pfeilspitzen mehr Platz brauchen.

Da die Knoten eine festgelegte Höhe haben, kann es beim Berechnen der Ports jedoch zu dem Problem kommen, dass nicht genug Platz für alle Ports ist (siehe Abb. 4.14a). Dann müssen die Abstände zwischen den Ports so weit reduziert werden, bis alle Ports an den Knoten passen, auch wenn dadurch die Ports sehr nahe beieinander liegen (Abb. 4.14b). Am häufigsten tritt dieser Fall jedoch beim Startknoten auf, weil dieser fast immer die meisten Nachfolger hat. Hier lässt sich das Problem besser lösen. Da ober- und unterhalb des Startknotens keine anderen Knoten liegen, kann hier die Höhe des Knotens so angepasst werden, dass alle Ports Platz finden (Abb. 4.14c).



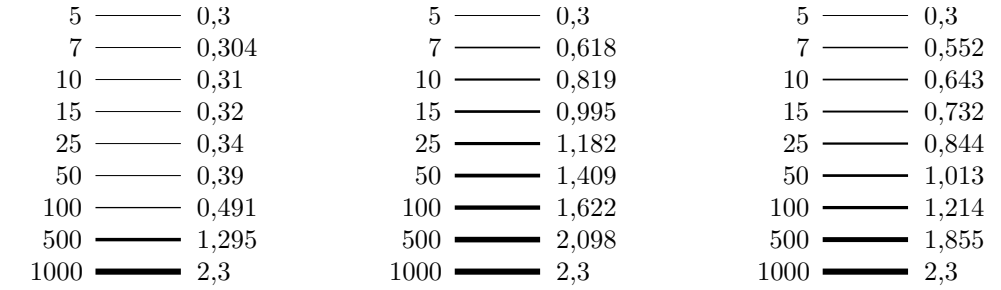
(a) Zu wenig Platz für Ports **(b)** Verschieben der Ports **(c)** Knotenhöhe vergrößern

Abb. 4.14.: Problem beim Berechnen der Ports

Damit der Betrachter der Zeichnung auf den ersten Blick erkennt, welche Teile der Zeichnung die Wichtigsten sind, werden die Kanten unterschiedlich dick gezeichnet. Je schwerer eine Kante ist, desto breiter soll sie gezeichnet werden. Dabei sollen eine Minimalbreite $b_{\min}$ und eine Maximalbreite $b_{\max}$ der Kanten festgelegt werden können, so dass die leichteste Kante die Minimalbreite hat und die schwerste Kante die Maximalbreite. Um herauszufinden, wie die Abhängigkeit zwischen Gewicht und Breite der Kanten am besten auszusehen hat, wurden verschiedene Varianten untersucht. Zuerst wurde ein linearer Zusammenhang getestet ($w_{\min}$ und $w_{\max}$ sind die Gewichte der leichtesten und schwersten Kante im Rechengraph):

$$b_1(e) = b_{\min} + (b_{\max} - b_{\min}) \cdot \frac{w(e) - w_{\min}}{w_{\max} - w_{\min}}$$

Es stellte sich jedoch schnell heraus, dass diese Berechnung der Breite ungeeignet ist. Das liegt daran, dass es bei Rechengraphen oft sehr viele leichte Kanten und nur wenig schwere Kanten gibt (meist diejenigen, die auf einem richtigen Lösungsweg liegen). Das führt bei einem linearen Zusammenhang dazu, dass einzelne Kanten sehr breit gezeichnet werden, zwischen allen anderen jedoch fast kein Unterschied in der Breite zu erkennen ist. Abbildung 4.15 zeigt die unterschiedlichen Definitionen der Kantenbreite an einem Beispiel. Wie man in Abbildung 4.15a erkennt, sind sich bei einer linearen Abhängigkeit die leichten Kanten zu ähnlich.



(a) lineare Abhängigkeit **(b)** logarithmische Abhängigkeit **(c)** Abhängigkeit von der dritten Wurzel

Abb. 4.15.: Verschiedene Kantenbreiten ($w_{\min} = 5$; $w_{\max} = 1000$; $b_{\min} = 0{,}3$; $b_{\max} = 2{,}3$)

Es ist also eine Funktion nötig, die schon für leichte Kanten eine recht hohe Breite liefert, damit auch Kanten mit kleinem Gewicht unterscheidbar sind. Ein Beispiel für solch eine Funktion ist der Logarithmus. Das führte zu folgender Definition der Kantenbreite:

$$b_2(e) = b_{\min} + (b_{\max} - b_{\min}) \cdot \frac{\ln(1 + w(e) - w_{\min})}{\ln(1 + w_{\max} - w_{\min})}$$

Das Problem an dieser Definition ist, dass beim Logarithmus die Kantenbreiten zu schnell ansteigen, wodurch leichte Kanten zu dick gezeichnet werden (Abb. 4.15b). Auch das macht die Zeichnung unübersichtlich, weil es zu viele dicke Kanten gibt. Einen gu-

ten Kompromiss aus dem linearen und logarithmischen Zusammenhang bietet folgende Definition:

$$b_3(e) = b_{\min} + (b_{\max} - b_{\min}) \cdot \left(\frac{w(e) - w_{\min}}{w_{\max} - w_{\min}} \right)^{\frac{1}{3}}$$

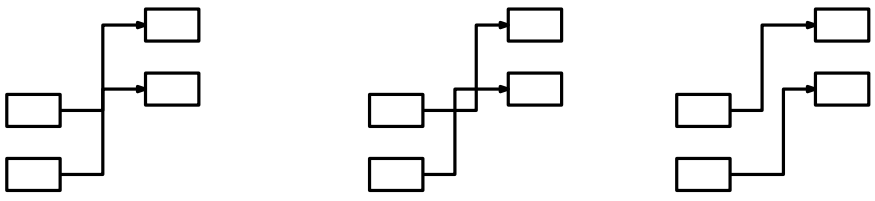
Hier steigen die Kantenbreiten nicht so schnell an, die leichten Kanten lassen sich aber deutlich besser voneinander unterscheiden als bei der linearen Abhängigkeit (Abb. 4.15c). Trotzdem ist eine gute Einstellung der Kantenbreite auch von der Eingabe abhängig, weshalb die dritte Definition auch nicht immer optimale Ergebnisse liefert. Für die meisten getesteten Eingaben war sie jedoch am besten geeignet.

4.11.2. Orthogonale Kanten mit mehreren Startports pro Knoten

Beim Zeichnen von Rechengraphen werden die Kanten üblicherweise in einem orthogonalen Stil gezeichnet, das heißt Kanten bestehen nur aus vertikalen und horizontalen Abschnitten. Der Vorteil im Gegensatz zu geradlinigen Kanten besteht darin, dass Kantenkreuzungen übersichtlicher sind, da zwei sich schneidende Abschnitte immer im 90 Grad Winkel zueinander stehen. Dadurch wird das Verfolgen einer Kante für den Betrachter vereinfacht.

Um eine übersichtliche Zeichnung zu erreichen, sollen außerdem möglichst wenig Kantenknicke auftreten. Dadurch, dass eine lagenbasierte Zeichnung verwendet wird und alle Kanten zwischen Knoten in benachbarten Lagen verlaufen, kann jetzt jede Kante mit maximal zwei Knicken gezeichnet werden. Dabei legen die beiden Ports der Kante fest, auf welcher Höhe die horizontalen Segmente gezeichnet werden. Um die endgültige Zeichnung zu erhalten, muss nun lediglich für jede Kante e die x-Koordinate des vertikalen Segments, bezeichnet mit $\mathrm{vs}(e)$, festgelegt werden.

Die vertikalen Segmente der Kanten können jedoch nicht beliebig positioniert werden. Zum einen sollen sich die vertikalen Segmente zweier Kanten nicht überlappen (Abb. 4.16a), da es sonst für den Betrachter unmöglich wäre, einer der beiden Kanten zu folgen. Außerdem können durch ungeschicktes Festlegen der vertikalen Segmente unnötige Kantenkreuzungen entstehen (Abb. 4.16b). Um das zu verhindern, wird ein sogenannter *Kantengraph* erstellt, mit dessen Hilfe passende Positionen für die vertikalen Segmente aller Kanten festgelegt werden (Abb. 4.16c).



(a) vertikale Segmente aufeinander **(b)** falsche Positionen **(c)** richtige Positionen

Abb. 4.16.: Festlegen der vertikalen Segmente

Die Knoten im Kantengraph entsprechen dabei den Kanten im Rechengraph. Die gerichteten Kanten im Kantengraph geben dann an, wo die vertikalen Segmente gezeichnet werden müssen, um unnötige Kreuzungen zu vermeiden. Eine Kante (e_1, e_2) bedeutet, dass das vertikale Segment von e_1 links vom vertikalen Segment von e_2 gezeichnet werden muss. Es kann außerdem sein, dass sowohl die Kante (e_1, e_2) als auch die Kante (e_2, e_1) im Kantengraph enthalten sind. Man sagt dann es existiert eine *Doppelkante* zwischen e_1 und e_2. Das passiert immer dann, wenn sich e_1 und e_2 unabhängig von der Position der vertikalen Segmente schneiden. Dann ist es für die Anzahl von Kreuzungen egal, wo die vertikalen Segmente liegen. Diese Information wird jedoch später ausgenutzt, um festzulegen, wo sich die Segmente schneiden.

Zum Einfügen der Kanten in den Kantengraph müssen alle Paare von Kanten, die zwischen den gleichen Lagen liegen, betrachtet werden. Anhand der Positionen der Ports der Kanten wird dann entschieden, ob Kanten eingefügt werden und in welcher Richtung. Dabei gibt es vier Fälle, die unterschieden werden (Abb. 4.17), wobei hier ohne Beschränkung der Allgemeinheit e_1 immer die Kante ist, deren Startport weiter oben liegt, das heißt es gilt immer $p_s(e_1) > p_s(e_2)$:

1. Die vertikalen Segmente zweier Kanten e_1 und e_2 können unabhängig voneinander festgelegt werden. Dann wird keine Kante zwischen e_1 und e_2 eingefügt. Abbildung 4.17a zeigt ein Beispiel. Dieser Fall tritt ein, wenn folgende Bedingung erfüllt ist:

   $$\min\{p_s(e_1), p_z(e_1)\} > \max\{p_s(e_2), p_z(e_2)\}$$

2. Zwei Kanten e_1 und e_2 schneiden sich. Dann wird sowohl die Kante (e_1, e_2) als auch die Kante (e_2, e_1) in den Kantengraph eingefügt. Abbildung 4.17b zeigt diesen Fall. Hier ist es egal, wo die vertikalen Segmente liegen, so lange sie nicht aufeinander gezeichnet werden. Dieser Fall tritt ein, wenn folgende Bedingung erfüllt ist:

   $$p_z(e_1) < p_z(e_2)$$

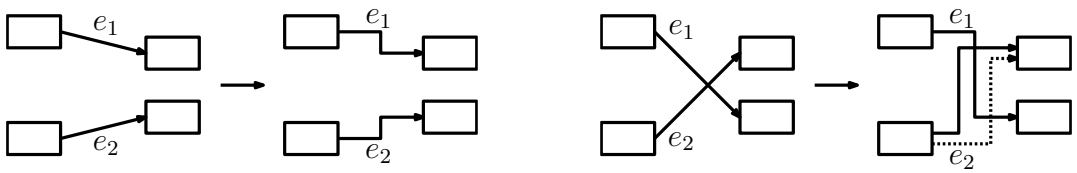
3. Die Ports von zwei Kanten e_1 und e_2 liegen so, dass durch geschicktes Platzieren der vertikalen Segmente unnötige Kreuzungen vermieden werden können und die Zielports der Kanten liegen jeweils weiter oben als die entsprechenden Startports. Dann wird die Kante (e_1, e_2) in den Kantengraph eingefügt. Abbildung 4.17c zeigt dafür ein Beispiel. Der Fall tritt ein, wenn folgende Bedingung erfüllt ist:

   $$p_z(e_1) > p_z(e_2) \wedge p_s(e_1) < p_z(e_2)$$

4. Ähnlich wie der dritte Fall, mit dem Unterschied, dass die Zielports weiter unten liegen als die entsprechenden Startports. Dann wird die Kante (e_2, e_1) eingefügt (siehe Abb. 4.17d). Dieser Fall tritt ein, wenn folgende Bedingung erfüllt ist:

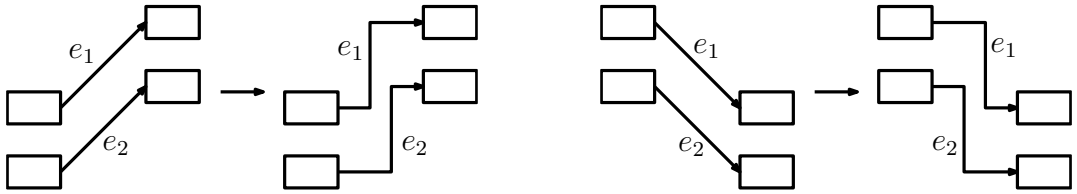
   $$p_z(e_1) > p_z(e_2) \wedge p_s(e_2) > p_z(e_1)$$

Mit Hilfe des erzeugten Kantengraphs wird nun festgelegt, wo die vertikalen Segmente der Kanten gezeichnet werden. Dazu wird zunächst die Reihenfolge der vertikalen Segmente von links nach rechts bestimmt, die konkreten Positionen werden später festgelegt. Algorithmus 12 zeigt das Vorgehen.



(a) Keine Kante wird eingefügt

(b) Doppelkante wird eingefügt



(c) Kante (e_1, e_2) wird eingefügt

(d) Kante (e_2, e_1) wird eingefügt

Abb. 4.17.: Einfügen von Kanten in den Kantengraph

Algorithmus 12: Berechenen der Reihenfolge der vertikalen Segmente

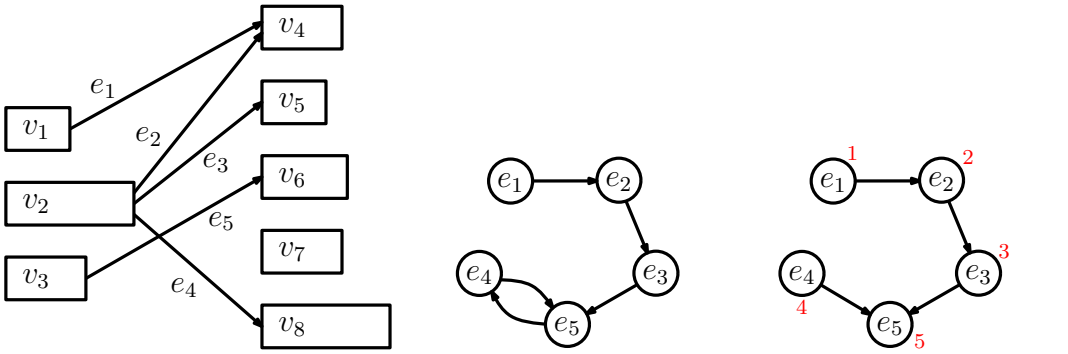
Eingabe : Kantengraph KG
Ausgabe : Reihenfolge der vertikalen Segmente der Kanten

```
1 sei Z die Menge der schwachen Zusammenhangskomponenten von KG
2 foreach z ∈ Z do
3     entferne Doppelkanten aus Z
4     berechne topologische Sortierung von z
5     füge von jeder entfernten Doppelkante den Teil wieder ein, der die topologische
      Sortierung nicht verletzt
6     topologische Sortierung entspricht dann der Reihenfolge der vertikalen
      Segmente in z
```

Anfangs ist es sinnvoll, den Kantengraph in seine schwachen Zusammenhangskomponenten zu zerlegen, da die Reihenfolge für jede Komponente separat berechnet werden kann. Anschließend werden die Doppelkanten aus dem Graph entfernt. Denn bei diesen ist es, wie bereits erwähnt, egal, in welcher Reihenfolge die vertikalen Segmente liegen. Zu diesem Zeitpunkt ist der Kantengraph kreisfrei, weil nur noch Abhängigkeiten wie im dritten oder vierten Fall vorhanden sind. Deshalb ist es jetzt möglich, eine topologische Sortierung des Graphen bzw. seiner Zusammenhangskomponenten zu berechnen. Diese entspricht dann der Reihenfolge der vertikalen Segmente. Am Ende wird noch von jeder Doppelkante der Teil wieder in den Graph eingefügt, der die topologische Sortierung nicht verletzt. Das ist nötig, damit in späteren Schritten klar ist, dass die vertikalen Segmente der beiden Kanten nicht aufeinander gezeichnet werden dürfen.

Abbildung 4.18 zeigt ein Beispiel für einen Rechengraph, den dazugehörigen Kan-

tengraph und den Kantengraph nach Anwendung von Algorithmus 12. Die roten Ziffern geben dabei die topologische Sortierung, also die Reihenfolge für die vertikalen Segmente in der Zeichnung an.

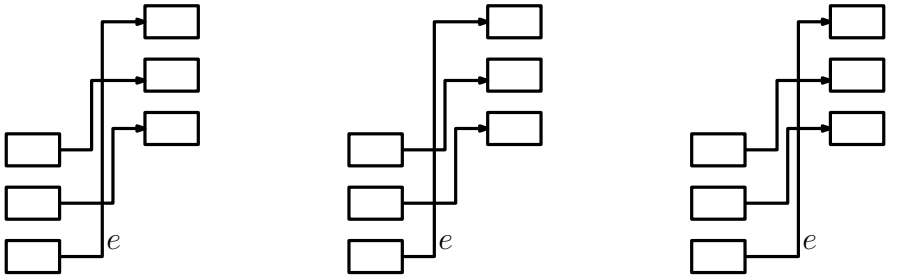


(a) Rechengraph **(b)** Kantengraph **(c)** Kantengraph nach dem Festlegen der Reihenfolge

Abb. 4.18.: Beispiel für einen Rechengraph und den zugehörigen Kantengraph

Mit Hilfe des veränderten Kantengraphs und der Sortierung kann nun eine Zeichnung des Rechengraphen mit orthogonalen Kanten erstellt werden, so dass es genauso viele Kantenkreuzungen gibt wie beim Verwenden von geradlinigen Kanten. Es kann dabei jedoch ein kleines Problem auftreten. Denn für die Lesbarkeit der Zeichnung ist es nicht nur wichtig, wie viele Kantenkreuzungen es gibt, sondern auch, wo diese liegen. Abbildung 4.19a zeigt ein Beispiel, in dem die Positionen der vertikalen Segmente zwar so gesetzt sind, dass es möglichst wenig Kreuzungen gibt, die Zeichnung aber trotzdem nicht optimal ist. Das liegt daran, dass das vertikale Segment von e in der Mitte liegt. Besser wäre es, wenn es ganz links (Abb. 4.19b) oder ganz rechts liegen würde (Abb. 4.19c). Allgemein kann man sagen, dass die vertikalen Segmente von Kanten, die viele Kreuzungen verursachen, möglichst weit links oder rechts liegen sollen. Deshalb wird bei der Berechnung der topologischen Sortierung darauf geachtet, wie viele Doppelkanten ein Knoten hatte und dieser wird dann so früh oder so spät wie möglich nummeriert. Damit ist auch dieses Problem gelöst.

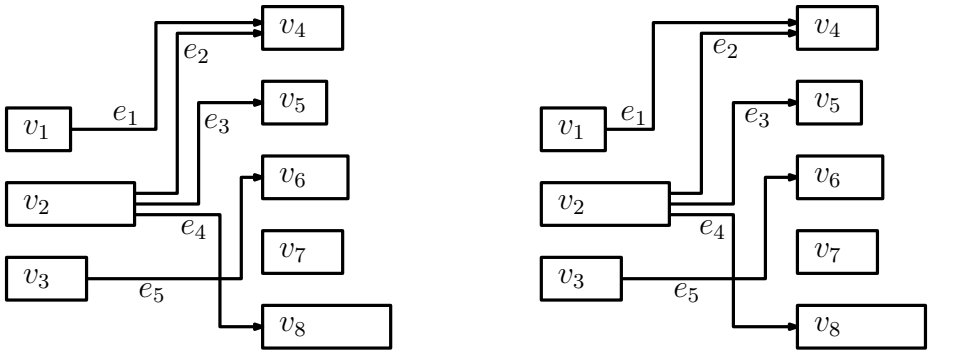
Jetzt ist also für jede Zusammenhangskomponente des Kantengraphs eine Reihenfolge der vertikalen Segmente der Kanten so festgelegt, dass es nur so viele Kantenkreuzungen gibt wie beim Verwenden von geradlinigen Kanten. Außerdem wurde die Reihenfolge durch das Beachten der Doppelkanten so gewählt, dass die Kreuzungen an guten Positionen liegen. Die vertikalen Segmente können nun entsprechend der berechneten Reihenfolge gezeichnet werden. Dazu werden die Segmente gleichmäßig auf die Räume zwischen den Lagen aufgeteilt (siehe Abb. 4.20a). Hier wird nicht darauf geachtet, wie breit die einzelnen Knoten in den Lagen sind, sondern die Lagenbreite entspricht dem breitesten Knoten. Deshalb kann der Platz oberhalb von v_2 und rechts von v_1 nicht genutzt werden, obwohl es besser wäre das vertikale Segment von e_1 deutlich weiter nach links zu schieben. Außerdem wird durch die Sortierung jeder Kante in einer Zusammenhangskomponente des Kantengraphs eine andere Position für das vertikale Segment



(a) Schlechte Position **(b)** Bessere Position **(c)** Bessere Position

Abb. 4.19.: Positionen von Kreuzungen

zugewiesen. Das ist jedoch nicht immer nötig. So könnten sich zum Beispiel e_3 und e_4 die Position für das vertikale Segment teilen, da sie nicht voneinander abhängig sind. Abbildung 4.20b zeigt, wie eine verbesserte Version der Zeichnung aussehen könnte.



(a) Erste Aufteilung der vertikalen Segmente **(b)** Nach dem kräftebasierten Verschieben

Abb. 4.20.: Festlegen der vertikalen Segmente

Um die beiden angesprochenen Verbesserungen einzubauen, wird ein sogenannter *kräftebasierter Algorithmus* verwendet. Dieser funktioniert folgendermaßen: Am Anfang ist eine Ausgangsposition gegeben, in der alle vertikalen Segmente festgelegt sind. Dann wird für jedes Segment eine Kraft berechnet, die von den umliegenden Segmenten abhängt. Anschließend werden die Kräfte auf die Segmente angewendet, das heißt die Position eines Segments ändert sich abhängig von Richtung und Stärke der entsprechenden Kraft. Dies wird so lange wiederholt, bis sich ein Gleichgewicht einstellt, die Kräfte also so klein werden, dass sich die Zeichnung kaum noch verändert. Das Entscheidende an diesem Verfahren ist die Definition der Kräfte. Um diese für das Verschieben der vertikalen Segmente festzulegen, ist es zuvor noch nötig, die Grenzen zu definieren, innerhalb derer das vertikale Segment einer Kante verschoben werden kann. Diese wiederum sind abhängig von den sogenannten *linken* und *rechten Nachbarn* der Kante. Der linke Nachbar einer Kante e ist dabei der Vorgänger von e im Kantengraph mit dem am weitesten

rechts liegenden rechten Kantenrand beim vertikalen Teil der Kante. Der rechte Nachbar einer Kante e ist der Nachfolger von e im Kantengraph mit dem am weitesten links liegenden linken Kantenrand. Man beachte, dass nicht jede Kante einen linken und rechten Nachbar haben muss. Formal lautet die Definition der Nachbarn folgendermaßen:

Definition 12 (Linker und rechter Nachbar einer Kante)**.** Sei $G = (V, E)$ ein Rechengraph und $\mathrm{KG} = (E, H)$ der zugehörige Kantengraph. Dann ist der linke Nachbar einer Kante e definiert als

$$n_l(e) = \underset{f \in E | (f,e) \in H}{\arg\max} \{\mathrm{vs}(f) + \frac{b(f)}{2}\}$$

Analog ist der rechte Nachbar einer Kante e definiert als

$$n_r(e) = \underset{f \in E | (e,f) \in H}{\arg\min} \{\mathrm{vs}(f) - \frac{b(f)}{2}\}$$

Mit den Nachbarn können nun die linken und rechten Grenzen bestimmt werden, die die Bewegung der vertikalen Segmente begrenzen. In vielen Fällen entspricht die linke Grenze dem rechten Kantenrand des linken Nachbarn und rechte Grenze dem linken Kantenrand des rechten Nachbarn. Wenn eine Kante jedoch relativ nahe an den Knoten liegt, können auch die Knoten die Bewegung der Kante begrenzen. Dabei sind allerdings nur die Knoten relevant, die in dem Bereich liegen, der durch die Höhe der Start- und Zielports der Kante begrenzt wird. Außerdem ist es sinnvoll, einen Sicherheitsabstand festzulegen, so dass sich Knoten und vertikale Segmente nicht zu nahe kommen können. Formal sind die Grenzen folgendermaßen definiert:

Definition 13 (Linke und rechte Grenze einer Kante)**.** Sei $G = (V, E)$ ein Rechengraph und sei $d_{\min} > 0$ der Sicherheitsabstand zwischen Knoten und vertikalen Segmenten. Dann ist die linke Grenze einer Kante $e = (u, v)$ definiert als

$$g_l(e) = \begin{cases} \max\{\mathrm{vs}(n_l(e)) + \frac{b(n_l(e))}{2}, x(v_a) + b(v_a) + d_{\min} \mid v_a \in A\}, & \text{falls } n_l(e) \text{ existiert} \\ \max\{x(v_a) + b(v_a) + d_{\min} \mid v_a \in A\}, & \text{sonst} \end{cases}$$

mit

$$A = \{w \in V \mid l(w) = l(u), y(w) > \min\{p_s(e), p_z(e)\}, y(w) - h(w) < \max\{p_s(e), p_z(e)\}.$$

Analog ist die rechte Grenze einer Kante $e = (u, v)$ definiert als

$$g_r(e) = \begin{cases} \min\{\mathrm{vs}(n_r(e)) - \frac{b(n_r(e))}{2}, x(v_b) - d_{\min} \mid v_b \in B\}, & \text{falls } n_r(e) \text{ existiert} \\ \min\{x(v_b) - d_{\min} \mid v_b \in B\}, & \text{sonst} \end{cases}$$

mit

$$B = \{w \in V \mid l(w) = l(v), y(w) > \min\{p_s(e), p_z(e)\}, y(w) - h(w) < \max\{p_s(e), p_z(e)\}.$$

Man beachte hierbei, dass die linken und rechten Nachbarn von den konkreten Positionen der vertikalen Segmente abhängig sind. Dadurch können sich die Grenzen während des kräftebasierten Algorithmus ändern, das heißt die Grenzen müssen in jeder Iteration neu bestimmt werden.

Mit Hilfe der linken und rechten Grenzen ist es jetzt möglich, die Kraft $f(e)$ auf eine Kante e im kräftebasierten Algorithmus festzulegen. Diese wirkt auf das vertikale Segment von e und ist so definiert, dass das vertikale Segment von e in die Mitte zwischen die beiden Grenzen von e geschoben wird:

$$f(e) = c \cdot \left(\frac{g_r(e) + g_l(e)}{2} - \mathrm{vs}(e) \right)$$

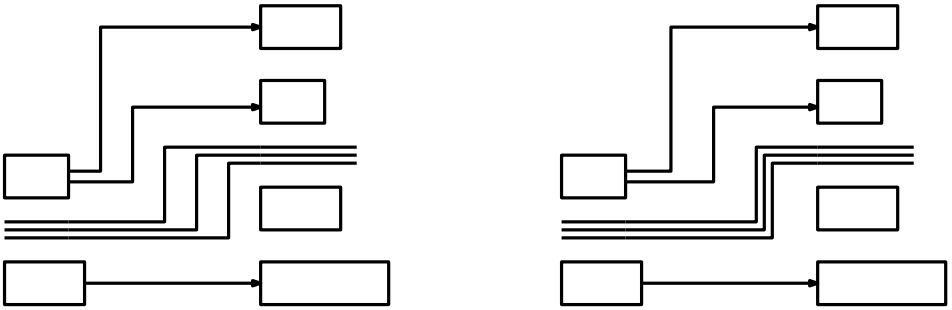
Die Konstante c dient dazu, die Stärke der Kräfte anzupassen, damit die Positionsänderungen in einer Iteration nicht zu groß sind, wodurch ein „Herumspringen“ der vertikalen Segmente verhindert wird. Es muss dabei $c < 1$ gelten. Für die Tests wurde ein Wert von $c = 0.3$ verwendet, was gut funktioniert hat.

Durch Anwendung des kräftebasierten Algorithmus werden die beiden angesprochenen Probleme gelöst. Einerseits können sich nun mehrere Kanten die Position für ihre vertikalen Segmente teilen, wenn sie nicht voneinander abhängig sind, da bei der Bestimmung der Nachbarn nicht mehr auf die topologische Sortierung des Kantengraphs geachtet wird. Zum anderen kann jetzt der Platz neben schmalen Knoten genutzt werden, da, wenn es keinen Nachbar gibt, auf die relevanten Knotenbreiten geachtet wird und nicht mehr nur auf die Lagenbreite. Das kann außerdem zur Folge haben, dass sich die vertikalen Segmente besser verteilen, also mehr Platz zwischen ihnen ist, was die Zeichnung zusätzlich besser macht.

Die Verteilung der vertikalen Segmente ist jetzt bereits sehr gut. Es gibt aber noch ein kleines Problem: Wie in Kapitel 4.9 beschrieben sollen nebeneinander liegende Dummyknoten, deren echte Kanten den gleichen Start- oder Zielknoten haben, nahe beieinander gezeichnet werden. Dadurch kann es häufig zu einer Situation wie in Abbildung 4.21a kommen, in der die Kanten parallel verlaufen. In solch einem Fall ist es dann gerade nicht erwünscht, dass die vertikalen Segmente der Kanten gut verteilt sind. Besser ist es hier, wenn die vertikalen Segmente den gleichen Abstand haben wie die entsprechenden Dummyknoten (Abb. 4.21b).

Um das zu erreichen, wird nach der Ausführung des kräftebasierten Algorithmus geprüft, ob es zu weit auseinander liegende vertikale Segmente gibt. Ist das der Fall, werden diese so zusammengesetzt, dass die Abstände gleich denen zwischen den Dummyknoten sind. Anschließend wird noch einmal der kräftebasierte Algorithmus ausgeführt, wobei die eben zusammengesetzten Segmente nicht mehr unabhängig voneinander, sondern zusammen verschoben werden. Das wird dadurch realisiert, dass Mengen von den Kanten gebildet werden, für die dann jeweils nur eine Kraft berechnet wird. Dazu müssen die Definitionen der Nachbarn und Grenzen leicht angepasst werden.

Definition 14 (Linker und rechter Nachbar einer Kantenmenge)**.** Sei $G = (V, E)$ ein Rechengraph und $\mathrm{KG} = (E, H)$ der zugehörige Kantengraph. Dann ist der linke Nachbar



(a) Schlechte Positionen der vertikalen Segmente **(b)** Bessere Positionen

Abb. 4.21.: Positionen der vertikalen Segmente bei parallelen Kanten

einer Kantenmenge M definiert als

$$n_l(M) = \underset{e|e\in(E\setminus M)\wedge\exists m\in M \text{ mit } (e,m)\in H}{\arg\max} \{\mathrm{vs}(e) + \frac{b(e)}{2}\}$$

Analog ist der rechte Nachbar einer Kantenmenge M definiert als

$$n_r(M) = \underset{e|e\in(E\setminus M)\wedge\exists m\in M \text{ mit } (m,e)\in H}{\arg\min} \{\mathrm{vs}(e) - \frac{b(e)}{2}\}$$

Definition 15 (Linke und rechte Grenze einer Kantenmenge)**.** Sei $G = (V, E)$ ein Rechengraph, sei $d_{\min} > 0$ der Sicherheitsabstand und sei M eine Kantenmenge, deren Kanten von der Lage L_1 zur Lage L_2 verlaufen. Sei zudem

$$Y_{\min} = \min\{p_s(m), p_z(m) \mid m \in M\} \text{ und } Y_{\max} = \max\{p_s(m), p_z(m) \mid m \in M\}.$$

Dann ist die linke Grenze der Kantenmenge M definiert als

$$l_l(M) = \begin{cases} \max\{\mathrm{vs}(n_l(M)) + \frac{b(n_l(M))}{2}, x(v) + b(v) + d_{\min} \mid v \in A\}, & \text{falls } n_l(M) \text{ existiert} \\ \max\{x(v) + b(v) + d_{\min} \mid v \in A\}, & \text{sonst} \end{cases}$$

mit

$$A = \{w \in V \mid l(w) = L_1, y(w) > Y_{\min}, y(w) - h(w) < Y_{\max}.$$

Analog ist die rechte Grenze der Kantenmenge M definiert als

$$l_r(M) = \begin{cases} \min\{\mathrm{vs}(n_r(M)) - \frac{b(n_r(M))}{2}, x(v) - d_{\min} \mid v \in B\}, & \text{falls } n_r(M) \text{ existiert} \\ \min\{x(v) \mid v \in -d_{\min}B\}, & \text{sonst} \end{cases}$$

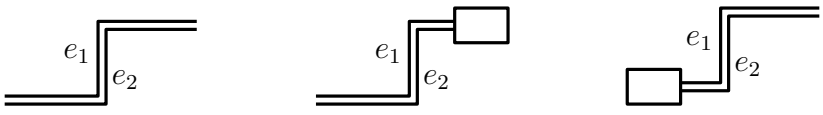
mit

$$B = \{w \in V \mid l(w) = L_2, y(w) > Y_{\min}, y(w) - h(w) < Y_{\max}.$$

Mit diesen Definitionen der Nachbarn und Grenzen kann der kräftebasierte Algorithmus nun so ausgeführt werden, dass die Kantenmengen zusammen verschoben werden. Damit bleibt nur noch die Frage, wie die Kantenmengen gebildet werden. Dazu werden zunächst alle Kanten jeweils einer neuen Menge hinzugefügt, so dass es genauso viele einelementige Mengen wie Kanten gibt. Anschließend werden so oft wie möglich zwei Mengen zusammengefasst. Zwei Mengen E_1 und E_2 können zusammengefasst werden, wenn zwei Kanten $e_1 = (u_1, v_1) \in E_1$ und $e_2 = (u_2, v_2) \in E_2$ existieren, für die folgende Bedingungen gelten:

- $l(u_1) = l(u_2)$ und $l(v_1) = l(v_2)$
- es gibt keinen Weg von e_1 nach e_2 im Kantengraph abgesehen von der direkten Verbindung (e_1, e_2)
- es gilt eine der folgenden Bedingungen:
  - u_1, u_2, v_1, v_2 sind Dummyknoten; $p(u_1) = p(u_2) \pm 1$; $p(v_1) = p(v_2) \pm 1$; die echten Kanten von e_1 und e_2 haben den gleichen Start- oder Zielknoten.
  - u_1, u_2 sind Dummyknoten; $v_1 = v_2$; $p(u_1) = p(u_2) \pm 1$
  - v_1, v_2 sind Dummyknoten; $u_1 = u_2$; $p(v_1) = p(v_2) \pm 1$

Um zwei Mengen zusammenzufassen müssen also die Kanten zwischen den gleichen Lagen verlaufen. Außerdem darf es im Kantengraph keinen Weg von e_1 nach e_2 außer der direkten Verbindung geben. Denn gäbe es einen solchen Weg, würden die vertikalen Segmente der Kanten, über die der Weg verläuft, zwischen den vertikalen Segmenten von e_1 und e_2 „eingeschlossen“ werden. Die letzte Bedingung beschreibt die Fälle, in denen die vertikalen Segmente nah beieinander gezeichnet werden sollen. Abbildung 4.22 zeigt diese drei Fälle. Im ersten Fall sind die Endknoten von e_1 und e_2 alles Dummyknoten, wobei die entsprechenden echten Kanten einen gemeinsamen Start- oder Zielknoten haben müssen. Im zweiten und dritten Fall haben e_1 und e_2 den gleichen Start- bzw. Zielknoten.



(a) Fall 1 **(b)** Fall 2 **(c)** Fall 3

Abb. 4.22.: Zusammenfassen von Kantenmengen

Mit dem in diesem Kapitel vorgestellten Verfahren zum Zeichnen der orthogonalen Kanten lassen sich nun gute Zeichnungen von Rechengraphen erstellen. Es gibt allerdings immer noch kleine Probleme, die in seltenen Fällen auftreten können. Diese wurden im Rahmen dieser Arbeit nicht mehr näher betrachtet, weil noch die anderen Kantenstile, die in den folgenden beiden Kapitel beschrieben werden, untersucht wurden. Abbildung 4.23 zeigt zwei Probleme, die manchmal auftreten können. Zum einen kann es vorkommen, dass Kanten nur sehr kurze vertikale Segmente haben, wenn die Höhe

der Eingangs- und Ausgangsports nah beieinander liegen (Abb. 4.23a). Außerdem kann es durch ungünstige Ports dazu kommen, dass Teile von Kanten aufeinander gezeichnet werden (Abb. 4.23b). Um diese beiden Sachen zu verhindern, ist es vermutlich am sinnvollsten, das Verfahren zum Festlegen der Ports anzupassen, da beide Probleme durch geschicktes Setzen der Ports vermieden werden können.

(a) Kurze vertikale Segmente **(b)** Überlappung von Kanten

Abb. 4.23.: Bestehende Probleme

4.11.3. Orthogonale Kanten mit einem Startport pro Knoten

Eine Variante des im letzten Kapitel vorgestellten Zeichenstils ist es, die Startports von mehreren Kanten, die den selben Startknoten haben, zu bündeln. Dadurch kommt es nicht mehr vor, dass bei zu vielen Startports an einem Knoten die Knotenhöhe angepasst werden muss oder die Ports zusammen geschoben werden müssen. Außerdem ist bei diesem Kantenstil wahrscheinlich weniger Platz zwischen den Lagen nötig, da die vertikalen Segmente von mehreren Kanten mit gleichem Startknoten aufeinander gezeichnet werden. Es hat sich jedoch schnell herausgestellt, dass Zeichnungen in diesem Stil unübersichtlich werden, wenn zwischen zwei Lagen viele Kanten gezeichnet werden müssen. Da zudem erste Zeichnungen, in denen die Kanten durch Kurven dargestellt wurden, deutlich vielversprechender aussahen, wurde der orthogonale Kantenstil mit einem Startport pro Knoten nicht weiter untersucht.

4.11.4. Kanten als Bézierkurven

Durch Verwenden des orthogonalen Kantenstils mit mehreren Startports können bereits gute Zeichnungen von Rechengraphen erzeugt werden. Trotzdem enthalten diese noch viele Knicke, was es teilweise schwer machen kann, einer Kante zu folgen. Dadurch entstand die Idee, die Kanten als Bézierkurven darzustellen, wodurch die Zeichnung keine Kantenknicke mehr enthält.

Eine *Bézierkurve* ist eine parametrisch modellierte Kurve, definiert durch einen Startpunkt und Endpunkt sowie beliebig viele Kontrollpunkte. Zum Zeichnen der Kanten werden kubische Bézierkurven verwendet, welche zwei Kontrollpunkte besitzen. Sie heißen kubisch, weil das die Kurve definierende Polynom ein Polynom dritten Grades ist. Die Punkte, die auf einer kubischen Bézierkurve liegen, sind folgendermaßen festgelegt:

$$B_3(t) = (1-t)^3 \cdot P_0 + 3(1-t)^2 t \cdot P_1 + 3(1-t)t^2 \cdot P_2 + t^3 \cdot P_3 \text{ mit } t \in [0;1]$$

Dabei ist P_0 der Startpunkt der Kurve und P_3 der Endpunkt. P_1 und P_2 sind die Kontrollpunkte. Die Umwandlung der orthogonalen Kanten in Bézierkurven ist simpel. Der Startpunkt der Kurve entspricht dem Startport der Kante, der Endpunkt entspricht dem Zielport. Der erste Kontrollpunkt liegt an der Position des ersten Knicks, der zweite Kontrollpunkt liegt an der Position des zweiten Knicks. Abbildung 4.24 zeigt dafür ein Beispiel. Wie man sieht, kommen die Kanten so ohne Knicke aus und auch beim Übergang einer Kante in einen Dummyknoten gibt es keinen Knick. Das liegt daran, dass Bézierkurven den Startknoten immer in Richtung des ersten Kontrollpunkts verlassen und dieser auf der gleichen Höhe liegt wie der Dummyknoten. Für den zweiten Kontrollpunkt und den Endpunkt gilt das Gleiche. Im Folgenden wird die vertikale Strecke, auf der die Kontrollpunkte liegen, weiterhin als vertikales Segment bezeichnet, auch wenn die Kanten eigentlich keine vertikalen Segmente mehr haben.

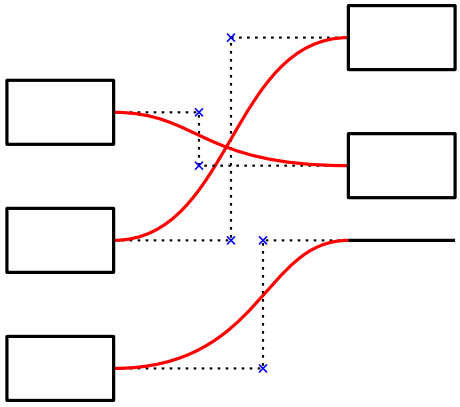
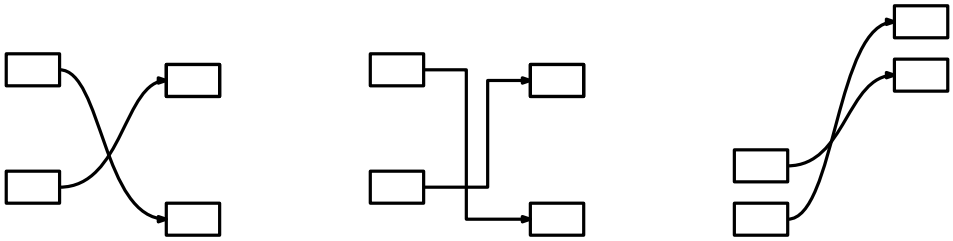


Abb. 4.24.: Bézierkurven anstatt orthogonaler Kanten

Durch den Einsatz von Bézierkurven kommt die Zeichnung nun völlig ohne Knicke aus, womit es für den Betrachter deutlich einfacher wird, dem Verlauf einer Kante zu folgen. Es ist aber jetzt nicht mehr, wie im orthogonalen Stil, garantiert, dass sich Kanten immer im 90 Grad Winkel schneiden. Das hat aber nur selten negative Auswirkungen auf die Zeichnung. Denn viele Kantenkreuzungen sehen so aus wie in Abbildung 4.25a. Hier ist der Winkel, in dem sich die Kanten schneiden, zwar kleiner, aber immer noch groß genug. In diesem Fall ist die Zeichnung mit Bézierkurven sogar besser, da sich die Kanten vor und nach dem Schnitt schneller voneinander entfernen als im orthogonalen Fall (Abb. 4.25b) und man sie so besser auseinander halten kann. Es gibt aber auch Fälle, in denen das Verwenden von Bézierkurven schlechter funktioniert. Abbildung 4.25c zeigt einen solchen Fall. Hier ist der Winkel, in dem sich die Kanten schneiden, sehr klein, was die Zeichnung unübersichtlich macht. Diese Fälle kommen jedoch nur selten vor, wodurch das Problem nicht allzu gravierend ist.

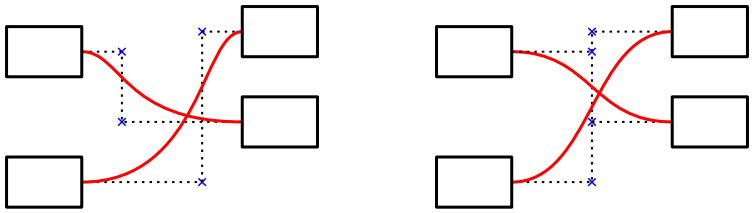
Das Verfahren zum Erstellen der Zeichnung mit Bézierkurven funktioniert grundsätzlich genauso wie im orthogonalen Fall. Es wird jedoch an einzelnen Stellen leicht verändert und erweitert. Zunächst werden, nachdem die topologische Sortierung des Kantengraphs berechnet wurde, die entfernten Doppelkanten nicht wieder in eine Richtung



(a) Schnitt mit Bézierkurven **(b)** Schnitt mit orthogonalen Kanten **(c)** Kleiner Schnittwinkel mit Bézierkurven

Abb. 4.25.: Vergleich von Schnitten mit Bézierkuven und orthogonalen Kanten

eingefügt. Das war im orthogonalen Fall nötig, damit die vertikalen Segmente der Kanten nicht aufeinander gezeichnet werden. Bei der Verwendung von Bézierkurven ist das nicht mehr nötig und es ist besser, wenn die Kontrollpunkte möglichst weit ein der Mitte zwischen den Lagen liegen, weil dann der Kurvenverlauf besser ist. Abbildung 4.26 zeigt das an einem etwas überspitzten Beispiel.



(a) Mit Doppelkanten **(b)** Ohne Doppelkanten

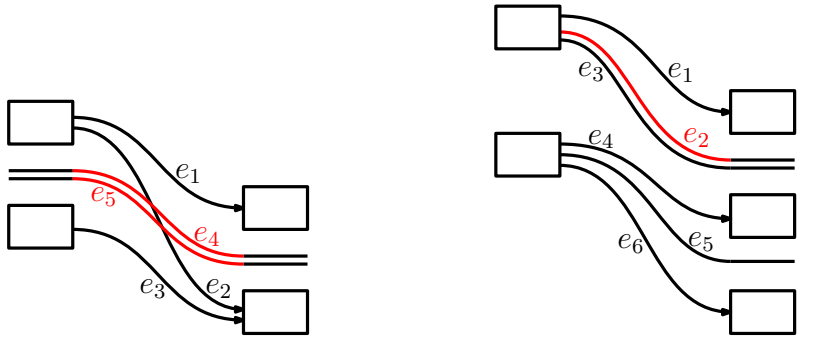
Abb. 4.26.: Beispiel zum Weglassen der Doppelkanten

Auch die zweite Erweiterung hat das Ziel, dass die Kontrollpunkte möglichst mittig zwischen die Lagen gesetzt werden können. Hier wird vor dem ersten Durchlauf des kräftebasierten Algorithmus geprüft, ob sich Kanten, die voneinander abhängig sind, trotzdem die vertikalen Segmente teilen können. Das ist dann möglich, wenn einer der beiden folgenden Fälle eintritt:

- Die Kanten haben einen gemeinsamen Startknoten und die Zielknoten sind keine nebeneinander liegenden Dummyknoten
- Die Kanten haben einen gemeinsamen Zielknoten und die Startknoten sind keine nebeneinander liegenden Dummyknoten

Es ist jedoch nicht möglich, beide Bedingungen im Algorithmus zu benutzen, weil es dann zu Widersprüchen kommen kann. Abbildung 4.27a zeigt einen solchen Fall. Hier würden sowohl die Kanten e_1 und e_2 als auch die Kanten e_2 und e_3 die Bedingung erfüllen. Damit würden e_1, e_2 und e_3 die gleichen vertikalen Segmente zugewiesen. Da aber die vertikalen Segmente der Kanten e_4 und e_5 links von e_1 und rechts von e_3

gezeichnet werden müssen, würden alle Kanten die gleichen vertikalen Segmente erhalten. Das darf aber nicht passieren, da die vertikalen Segmente von e_4 und e_5 nicht aufeinander liegen dürfen, damit sich die Kanten nicht zu nahe kommen. Es kann also nur eine Bedingung im Algorithmus benutzt werden. Da es häufiger vorkommt, dass Kanten einen gemeinsamen Startknoten haben, wird die erste Bedingung benutzt.



(a) Beide Bedingungen werden benutzt **(b)** Ports nicht nebeneinander

Abb. 4.27.: Widersprüche bei gemeinsamen vertikalen Segmenten

Es muss allerdings noch eine kleine Einschränkung gemacht werden: Die Ports der Kanten am Startknoten müssen nebeneinander liegen, denn sonst kann auch hier ein Widerspruch entstehen (Abb. 4.27b). Würde man nicht auf die Ports achten, könnten e_1 und e_3 die gleichen vertikalen Segmente zugewiesen werden. Das darf aber nicht passieren, da sonst e_2 das gleiche vertikale Segment hätte wie e_3, was nicht erlaubt ist. Diese Bedingung verhindert allerdings nicht, dass auch mehr als zwei Kanten das gleiche vertikale Segment haben dürfen, wenn die Ports wie bei den Kanten e_4, e_5 und e_6 nebeneinander liegen. Dieser Fall ist erlaubt.

Die vertikalen Segmente von zusammengehörigen Kanten werden dann aufeinander gesetzt und schon im ersten Durchlauf des kräftebasierten Verfahrens zusammen verschoben. Anschließend wird wieder geprüft, ob es Kanten gibt, deren vertikale Segmente in einem gewissen Abstand voneinander gemeinsam verschoben werden müssen. Ist das der Fall, werden die entsprechenden Kantenmengen zusammengefasst, die vertikalen Segmente werden entsprechend gesetzt und das kräftebasierte Verfahren wird nochmals ausgeführt.

Das Ziel der beiden vorgestellten Erweiterungen ist es, dass weniger unterschiedliche Positionen von vertikalen Segmenten nötig sind und so die Kontrollpunkte möglichst mittig zwischen den Lagen liegen können. Durch den kräftebasierten Algorithmus werden die vertikalen Segmente jedoch so zurecht geschoben, dass sie gleichmäßig zwischen den Lagen verteilt sind. Deshalb ist es beim Verwenden von Bézierkurven nötig, den Sicherheitsabstand beim Berechnen der Grenzen höher zu setzen als im orthogonalen Fall. Er muss allerdings in Abhängigkeit von dem Abstand von zwei Lagen passend gewählt werden, da sonst der kräftebasierte Algorithmus nicht mehr funktioniert. Beim Testen hat ein Sicherheitsabstand, der ein Drittel des Lagenabstandes war, gut funktioniert.

5. Ergebnisse

In den letzten beiden Kapiteln wurde der Zeichenalgorithmus für Rechengraphen vorgestellt. Für viele Schritte des Algorithmus wurden hier mehrere Möglichkeiten gezeigt, wie sie konkret ausgeführt werden können:

- Entfernen von leichten Knoten und Kanten: Hier können verschiedene Schrankenwerte festgelegt werden.
- Umlegen von Gewichten beim Entfernen von Knoten: Das Umlegen von Gewichten kann an oder ausgeschaltet werden. Außerdem kann festgelegt werden, ob beim Umlegen der Gewichte neue Kanten eingefügt werden dürfen.
- Lagenzuordnung: Hier kann Coffman-Graham oder PCMPS mit Knotengewichten verwendet werden. Außerdem kann der Algorithmus eingesetzt werden, der eine Lagenzuordnung mit minimaler Höhe findet.
- Reduzierung der Größe des Graphen: Hier können drei verschiedene Knotenwichtigkeiten benutzt werden.
- Reduzierung von Kantenkreuzungen: Es kann dafür die Adjacent-Exchange-Heuristik verwendet werden, die entweder die Anzahl oder das Gewicht der Kreuzungen minimieren kann. Außerdem kann die Mittelwert-Heuristik eingesetzt werden.

Um herauszufinden, welche Einstellungen für das Verfahren am besten funktionieren, wurden Tests mit 33 verschiedenen Graphen durchgeführt, die zwischen 17 und 1031 Knoten enthielten. Es stellte sich jedoch schnell heraus, dass kleine Graphen zum Testen der Einstellungen nicht geeignet sind. Das liegt daran, dass diese Graphen meist komplett auf die Zeichenfläche passen und so kaum Unterschiede zwischen den verschiedenen Einstellungen sichtbar werden. Für die folgenden Tests wurden daher nur Graphen verwendet, die mindestens 100 Knoten enthalten. Tabelle 5.1 zeigt die eingesetzten Graphen. Die Zeichenfläche bei den Tests entspricht ungefähr der Größe eines Din-A4-Blattes. Es ist außerdem anzumerken, dass bei allen Tests der Algorithmus mehrmals ausgeführt werden muss, da die Ergebnisse aufgrund der verwendeten Heuristiken auch vom Zufall abhängig sind.

Im nächsten Unterkapitel wird untersucht, welche Einstellungen allgemein am wichtigsten sind und am besten funktionieren. In den folgenden Unterkapiteln wird dann noch näher auf die Auswirkungen der einzelnen Parameter eingegangen. Anschließend wird überprüft, um wie viel das Gewicht der Ergebnisse bei gleichen Einstellungen schwankt, also wie verlässlich der Algorithmus ist, wenn er nur einmal ausgeführt wird. Außerdem wird getestet, wie viel Zeit der Algorithmus und seine einzelnen Teile zur Ausführung benötigen, bevor am Ende des Kapitels einige Beispielausgaben gezeigt werden.

Knotenanzahl	Kantenanzahl	Knotenanzahl	Kantenanzahl
107	177	530	833
143	243	563	913
170	276	663	1452
203	289	755	1575
331	565	775	1303
358	663	799	1242
406	900	1031	1549

Tab. 5.1.: Größe der Testgraphen

5.1. Allgemeines zu den Parametereinstellungen

Um herauszufinden, welche Parametereinstellungen die Wichtigsten sind und welche am besten funktionieren, wurden alle Kombinationen von Parametern getestet. Um verlässliche Ergebnisse zu erhalten, passierte dies jeweils 100 mal für jeden der 14 Testgraphen. Anschließend wurde für jeden Graph überprüft, welche fünf Parameterkombinationen die besten Ergebnisse liefern. Wie gut ein Ergebnis ist, hängt dabei vor allem vom Knotengewicht ab, das am Ende noch im Graph vorhanden ist. Aber auch das Gewicht der Kanten, die noch enthalten sind, ist wichtig.

Für den Schrankenwert, bis zu welchem leichte Knoten und Kanten am Anfang aussortiert werden, wurden zwei Werte getestet. Einmal wurden anfangs keine Knoten und Kanten entfernt, dann wurden alle mit einem Gewicht von fünf oder weniger aussortiert. Letztere Einstellung funktionierte dabei deutlich besser. Für die kleineren Graphen war es jedoch manchmal besser, keine Knoten und Kanten zu entfernen. Das lässt vermuten, dass der optimale Schrankenwert von der Größe des Graphen abhängt, was in Kapitel 5.3 näher untersucht wird.

Über das Umlegen von Gewichten beim Entfernen von Knoten kann hier keine Aussage getroffen werden, da die korrekte Implementierung des Umlegens zum Testzeitpunkt nicht fertig war [1]. Es wurden jedoch nachträglich zusätzliche Tests durchgeführt, die zeigen, dass sich das Umlegen von Gewichten positiv auswirkt. In Kapitel 5.4 wird näher darauf eingegangen.

Die Lagenalgorithmen funktionieren alle ähnlich gut, sogar das Verfahren, das die Lagenbreite nicht begrenzt und eine Lagenzuordnung mit minimaler Höhe findet, funktioniert gut. Außerdem ist die Qualität der Lagenalgorithmen vor allem von der Eingabe abhängig. Das zeigt sich daran, dass bei fast allen Eingaben genau ein Lagenalgorithmus in den fünf besten Parameterkombinationen verwendet wird. Genauere Testergebnisse zu den Lagenalgorithmen finden sich in Kapitel 5.5.

Die Einstellungen zur Knotenwichtigkeit scheinen für das Gewicht der Ausgabe keine große Rolle zu spielen. Das erkennt man daran, dass bei fast allen Eingaben jede Knotenwichtigkeit mindestens einmal unter den fünf besten Parameterkombinationen zu finden

[1] Als die Tests durchgeführt wurden, befand sich ein Fehler in der Implementierung, so dass nur in seltenen Fällen tatsächlich Gewicht umgelegt wurde. Da der Fehler erst spät erkannt wurde, konnten die Tests nicht mehr wiederholt werden.

war. Die anderen Parameter scheinen also für die Qualität der Ausgabe entscheidender zu sein. In Kapitel 5.6 wird dies genauer überprüft.

Bei der Kreuzungsminimierung sind alle drei Verfahren häufig vertreten. Für ein maximales Gewicht der Knoten sind die beiden Adjacent-Exchange-Heuristiken etwas besser geeignet, für maximales Kantengewicht hingegen ist die die Mittelwert-Heuristik etwas besser geeignet. Auf die unterschiedlichen Methoden zur Kreuzungsminimierung wird in Kapitel 5.7 noch näher eingegangen.

5.2. Entfernen von Kreisen

Zum Entfernen von Kreisen aus dem Rechengraph wurden zwei ganzzahlige lineare Programme vorgestellt. Das erste minimierte die Anzahl der Kanten, die im Feedback-Arc-Set enthalten sind. Weil dabei nicht auf die Kantengewichte geachtet wird und so auch sehr wichtige Kanten gelöscht werden können, ist es beim Zeichnen von Rechengraphen wahrscheinlich sinnvoller, ein möglichst leichtes Feedback-Arc-Set zu suchen. Es soll nun untersucht werden, wie die beiden ganzzahligen linearen Programme in der Praxis funktionieren. Dazu wurden die Programme für jede der 14 Eingaben ausgeführt, die Ergebnisse zeigt Tabelle 5.2.

	minimale Größe		minimales Gewicht	
Knotenanzahl	Kantenanzahl	Gewicht	Kantenanzahl	Gewicht
107	0	0	0	0
143	4	29	4	16
170	0	0	0	0
203	1	3	1	1
331	3	1078	4	50
358	2	7	4	6
406	9	29	9	15
530	5	1244	5	8
563	0	0	0	0
663	8	808	8	14
755	5	18	7	7
775	5	28	5	9
799	6	511	7	7
1031	5	12	5	5

Tab. 5.2.: Vergleich der Feedback-Arc-Sets mit minimaler Größe und minimalem Gewicht

Für viele Eingaben funktionieren beide Varianten ähnlich gut. Es gibt jedoch Fälle, in denen das Programm, welches die Größe das Feedback-Arc-Sets minimiert, deutlich schlechter abschneidet, weil hier auch sehr schwere Kanten entfernt werden (siehe Eingaben mit 311, 530, 663 und 799 Knoten). Zudem ist die Anzahl der Kanten im Feedback-Arc-Set oft gleich oder nur minimal höher, wenn auf das Gewicht geachtet wird. Damit lässt sich feststellen, dass beim Zeichnen von Rechengraphen das ganzzah-

lige lineare Programm, welches das Gewicht des Feedback-Arc-Sets minimiert, deutlich besser geeignet ist.

5.3. Aussortieren von leichten Knoten und Kanten

Zunächst soll herausgefunden werden, welchen Einfluss der Schrankenwert, bis zu dem leichte Knoten und Kanten am Anfang des Algorithmus entfernt werden, auf das Ergebnis hat. Dazu wurde der Algorithmus für jede Eingabe mit einem Schrankenwert von 0 bis 20 jeweils 100 mal ausgeführt. Die anderen Einstellungen waren Folgende: Gewichte entfernter Knoten sollen nicht umgelegt werden, als Lagenalgorithmus wurde PCMPS verwendet, als Knotenwichtigkeit wurde I_3 verwendet und zur Kreuzungsminimierung wurde die Mittelwert-Heuristik eingesetzt. Die Ergebnisse dieser Tests zeigt Tabelle 5.3.

	maximales Knotengewicht		maximales Kantengewicht	
Knotenanzahl	Schrankenwert	Verbesserung	Schrankenwert	Verbesserung
107	7	4,49 %	8	7,15 %
143	2	0,45 %	2	0,57 %
170	7	9,21 %	8	14,80 %
203	2	2,24 %	2	3,20 %
331	4	3,32 %	4	6,17 %
358	4	1,92 %	4	3,12 %
406	14	7,12 %	14	11,33 %
530	7	6,49 %	7	11,26 %
563	7	9,32 %	8	16,60 %
663	8	4,96 %	8	9,33 %
755	9	6,76 %	7	9,50 %
775	8	5,02 %	8	7,87 %
799	10	10,35 %	10	17,28 %
1031	11	6,79 %	10	8,64 %

Tab. 5.3.: Tests zum Entfernen von leichten Knoten und Kanten

Wie man sieht, hängt es stark von der Eingabe ab, wie sich das Entfernen von leichten Knoten und Kanten auf das Endergebnis auswirkt. Beim Graph mit 143 Knoten bringt das Entfernen fast nichts, das Knotengewicht am Ende ist lediglich 0,45 Prozent besser als ohne das Entfernen und das Kantengewicht verbessert sich nur um 0,57 Prozent. Bei dem Graph mit 799 Knoten hingegen steigt das Knotengewicht durch das Entfernen um 10,35 Prozent und das Kantengewicht sogar um 17,28 Prozent, was eine deutliche Verbesserung ist. Über alle 14 Testgraphen steigt das Knotengewicht am Ende um durchschnittlich 5,60 Prozent und das Kantengewicht um 9,06 Prozent.

Eine andere Frage ist, was der optimale Schrankenwert ist, bis zu dem leichte Knoten und Kanten entfernt werden sollen. Auch das ist von der Eingabe abhängig, wobei es einen Zusammenhang zu geben scheint zwischen der Größe des Graphen und dem optimalen Schrankenwert. Generell kann man sagen, dass der optimale Schrankenwert mit

der Knotenanzahl wächst, auch wenn die Eingaben mit 107, 170 und 406 Knoten hier aus dem Rahmen fallen.

Außerdem ist anzumerken, dass sich in manchen Fällen sehr große Differenzen zwischen den Knoten- und Kantengewichten ergeben können, obwohl der Schrankenwert nur um eins verändert wurde. Zwar ist es meist so, dass das Gewicht des Ergebnisses durch das Erhöhen des Schrankenwerts bis zum optimalen Wert immer ein wenig besser wird und anschließend mit jedem Erhöhen etwas schlechter wird. Es kommt aber auch vor, dass die Endgewichte hier starke Sprünge machen. So liegt das Knotengewicht des Graphen mit 406 Knoten mit einer Schranke von 14 noch bei 9329, mit einer Schranke von 15 jedoch nur noch bei 8472. Woran das aber liegt, lässt sich nur schwer beurteilen.

5.4. Umlegen von Gewichten beim Entfernen von Knoten

Um zu untersuchen, welche Auswirkungen das Umlegen von Gewichten beim entfernen von Knoten hat, wurden Test mit den drei möglichen Varianten durchgeführt. Für die anderen Parameter wurden dabei folgende Einstellungen gewählt: Leichte Knoten und Kanten werden bis zu einem Gewicht von fünf aussortiert, zur Lagenzuordnung wurde PCMPS verwendet, die Knotenwichtigkeit ist I_3 und zur Kreuzungsminimierung wurde die Mittelwert-Heuristik eingesetzt. Bei diesen Test zeigte sich, dass das Umlegen von Gewichten einen spürbaren Einfluss auf die Knotengewichte und Kantengewichte hat. Tabelle 5.4 zeigt die Ergebnisse.

	mittleres Knotengewicht	Änderung
kein Umlegen	6814,91	0
Umlegen auf existierende Kanten	6814,81	+ 0,001 %
Umlegen auch auf neue Kanten	6792,99	– 0,290 %

	mittleres Kantengewicht	Änderung
kein Umlegen	6450,37	0
Umlegen auf existierende Kanten	6503,38	+ 0,768 %
Umlegen auch auf neue Kanten	6599,85	+ 2,140 %

Tab. 5.4.: Verschiedene Einstellungen für das Umlegen von Knoten

Das Knotengewicht ändert sich beim Umlegen von Gewichten auf existierende Kanten quasi gar nicht. Lässt man es jedoch zu, dass beim Umlegen auch neue Kanten eingefügt werden, sinkt das Knotengewicht leicht. Das könnte daran liegen, dass es mehr Kanten gibt und deshalb nicht so viele entfernte Knoten wieder eingefügt werden können, weil es sonst zu viele Kantenkreuzungen geben würde. Das Kantengewicht hingegen profitiert deutlich vom Umlegen von Gewichten. Fügt man keine neue Kanten ein, erhöht sich das Kantengewicht um durchschnittlich 0,77 Prozent. Lässt man neue Kanten zu, erhöht sich das Gewicht sogar um durchschnittlich 2,14 Prozent. Hier ist anzumerken, dass die Gewichtszunahme recht stark von der Eingabe abhängt. Oft liegt die Zunahme bei weniger als einem Prozent, für einzelne Graphen liegt sie aber bei bis zu 10 Prozent.

5.5. Algorithmen zur Lagenzuordnung

Um herauszufinden, welcher der drei vorgestellten Algorithmen zur Lagenzuordnung am besten funktioniert, wurden für die 14 Graphen Tests mit folgenden Einstellungen durchgeführt: Leichte Knoten und Kanten werden bis zu einem Gewicht von fünf entfernt, Gewichte werden nicht umgelegt, als Knotenwichtigkeit wird I_3 verwendet und zur Kreuzungsminimierung wird die Mittelwert-Heuristik eingesetzt. Dabei zeigte sich, dass die Ergebnisse mit den unterschiedlichen Algorithmen sehr nahe beieinander liegen, sowohl beim Gewicht der Knoten als auch beim Gewicht der Kanten (Tab. 5.5).

	Knotengewicht	Änderung	Kantengewicht	Änderung
minimale Höhe	6828,35	0	6487,41	0
Coffman	6824,20	-0,05	6478,37	- 0,14
PCMPS	6808,22	-0,28	6482,81	- 0,07
mittlere Spannweite	52,27	-	66,02	-

Tab. 5.5.: Vergleich der Lagenalgorithmen mit dem Wiedereinfügen von Knoten und dem Entfernen von leichten Knoten und Kanten

Das Verfahren, das eine Lagenzuordnung mit minimaler Höhe erzeugt, funktioniert hier sogar am besten, obwohl dabei nicht auf die Lagenbreite geachtet wird. Coffman-Graham sowie PCMPS sind allerdings nur 0,05 und 0,28 Prozent schlechter, wenn man die Knotengewichte betrachtet. Auch bei den Kantengewichten sind sie nur 0,14 und 0,07 Prozent schlechter. Zudem ist die Spannweite vom schlechtesten zum besten Verfahren für jede Eingabe recht klein. Das heißt, dass die Unterschiede der Algorithmen in den einzelnen Eingaben tatsächlich gering sind und nicht durch das Betrachten des durchschnittlichen Gewichts „herausgemittelt“ werden. Die kleinen Unterschiede kommen auch dadurch zustande, dass leichte Knoten und Kanten am Anfang entfernt werden und entfernte Knoten wieder eingefügt werden können. Lässt man diese beiden Optimierungen weg, kommt man auf Ergebnisse wie in Tabelle 5.6.

	Knotengewicht	Änderung	Kantengewicht	Änderung
minimale Höhe	6081,06	0	5561,63	0
Coffman	6046,44	-0,55	5540,08	- 0,36
PCMPS	5983,98	-1,29	5454,63	- 1,48
mittlere Spannweite	222,53	-	196,33	-

Tab. 5.6.: Vergleich der Lagenalgorithmen ohne das Wiedereinfügen von Knoten und das Entfernen von leichten Knoten und Kanten

Dann liegen die durchschnittlichen Knoten- und Kantengewichte zwar immer noch recht nahe beieinander, die Spannweite erhöht sich jedoch stark. Das heißt die Unterschiede der drei Algorithmen sind für die einzelnen Eingaben deutlich größer. Das ist jedoch kein Problem, da für das Erzeugen von guten Ausgabezeichnungen das Wiedereinfügen von Knoten und das Entfernen von leichten Knoten und Kanten immer aktiviert

sein sollten. Dann sind die verschiedenen Verfahren zwar immer noch unterschiedlich gut, die Ergebnisse liegen aber sehr nahe beieinander.

5.6. Knotenwichtigkeiten

Um zu testen, wie sich die unterschiedlichen Knotenwichtigkeiten auswirken, wurden folgende Einstellungen benutzt: Es werden leichte Knoten und Kanten mit einem Gewicht von weniger als fünf entfernt, es wird kein Gewicht umgelegt beim Entfernen von Knoten, für die Lagenzuordnung wurde der Algorithmus verwendet, der eine Zuordnung mit minimaler Höhe liefert, und zur Kreuzungsminimierung wurde die Mittelwert-Heuristik benutzt. Dabei zeigte sich, dass es keine relevanten Unterschiede zwischen den verschiedenen Knotenwichtigkeiten gibt. Die Unterschiede der Knoten- und Kantengewichte liegen im Durchschnitt bei weniger als 0,012 Prozent. In Einzelfällen erhöht sich das Gewicht um bis zu 0,33 Prozent, aber selbst diese Gewichtszunahme ist in der Praxis kaum relevant.

Die geringen Unterschiede könnten aber auch dadurch zustande kommen, dass nach dem Aussortieren der leichten Knoten und Kanten nur wenige Knoten übrig bleiben und so der restliche Algorithmus nicht mehr viel tun muss. Deshalb wurde der Test wiederholt mit dem Unterschied, dass keine leichten Knoten und Kanten entfernt wurden. Doch auch mit dieser Einstellung funktionieren alle drei Knotenwichtigkeiten quasi gleich gut. Somit lässt sich feststellen, dass die verschiedenen Knotenwichtigkeiten im Algorithmus keine Auswirkung auf das Ergebnis haben.

5.7. Reduzierung von Kantenkreuzungen

Zum Reduzieren der Kantenkreuzungen wurden vier verschiedene Verfahren vorgestellt: Die Adjacent-Exchange-Heuristik, die Mittelwert-Heuristik, das ganzzahlige lineare Programm für zwei Lagen und das ganzzahlige lineare Programm für den kompletten Graph. Abgesehen von der Mittelwert-Heuristik können diese Verfahren auch dazu benutzt werden, das Gewicht der Kreuzungen anstatt die Anzahl der Kreuzungen zu minimieren. Das ganzzahlige lineare Programm für den kompletten Graph ist in der Praxis jedoch nicht einsetzbar, weil die benötigte Rechenzeit zu hoch ist. Das ganzzahlige lineare Programm für zwei Lagen ist schnell genug, konnte jedoch nicht vernünftig getestet werden, da Gurobi bei zu vielen Aufrufen abstürzte.

Somit bleiben drei Möglichkeiten für die Kreuzungsminimierung übrig, die im Algorithmus eingesetzt werden können. Um diese zu vergleichen, werden die Knoten- und Kantengewichte sowie die Anzahl und das Gewicht der Kreuzungen betrachtet. Für diese Test wurden folgende Einstellungen verwendet: Leichte Knoten und Kanten werden bis zu einem Gewicht von fünf entfernt, Gewichte werden nicht umgelegt, die Lagenzuordnung hat minimale Höhe und die Knotenwichtigkeit ist I_3. Tabelle 5.7 zeigt die durchschnittlichen Ergebnisse für die 14 Testgraphen.

Wie man sieht, hat die Heuristik zur Kreuzungsminimierung kaum Einfluss auf das Knotengewicht. Das Kantengewicht ist bei der Adjacent-Exchange-Heuristik etwas nied-

	Knotengewicht	Kantengewicht
Adjacent-Exchange	6841,25	6447,48
Adjacent-Exchange mit Gewichten	6843,14	6459,70
Mittelwert	6845,78	6463,01

	Kreuzungen	Gewicht der Kreuzungen
Adjacent-Exchange	14,41	3041,54
Adjacent-Exchange mit Gewichten	15,10	1219,68
Mittelwert	13,65	3114,88

Tab. 5.7.: Vergleich der Heuristiken zur Kreuzungsminimierung

riger als bei den anderen beiden, der Unterschied ist aber so gering, dass er in der Praxis kaum auffallen wird. Auch bei der Anzahl der Kantenkreuzungen sind alle Heuristiken ähnlich gut, wobei die Mittelwert-Heuristik hier etwas besser funktioniert. Einen großen Unterschied gibt es jedoch bei dem Gewicht der Kreuzungen. Wie zu erwarten, schneidet hier die Adjacent-Exchange-Heuristik mit Gewichten deutlich besser ab als die beiden anderen Heuristiken.

In der Praxis muss der Benutzer entscheiden, welche Heuristik er einsetzt. Wenn es ihm wichtig ist, dass der Graph eine minimale Anzahl von Kantenkreuzungen hat, sollte die Mittelwert-Heuristik eingesetzt werden. Normalerweise ist es aber sinnvoller, die durchschnittlich 1,5 zusätzlichen Kreuzungen in Kauf zu nehmen und damit das Gewicht der Kreuzungen um ca. 60 Prozent zu reduzieren.

5.8. Entfernen von Kanten

In der Ausgabezeichnung dürfen zwischen zwei Lagen maximal so viele Kantenkreuzungen existieren wie das Minimum der Anzahl von Knoten in den beiden Lagen. So wird verhindert, dass es zu viele Kreuzungen gibt. Noch besser wäre es allerdings, eine Zeichnung komplett ohne Kreuzungen zu erzeugen, was aber nur möglich ist, indem mehr Kanten gelöscht werden. Es ist nun interessant zu wissen, wie viel Gewicht dabei verloren geht, wenn man keine Kantenkreuzungen mehr erlaubt. Denn ist dieser Verlust gering, kann es sinnvoll sein, zugunsten einer kreuzungsfreien Zeichnung auf ein wenig Gewicht zu verzichten.

Dazu wurden Test durchgeführt, bei denen leichte Knoten und Kanten bis zu einem Gewicht von fünf aussortiert werden und kein Gewicht umgelegt wird. Für die Lagenzuordnung kam der Algorithmus mit minimaler Höhe zum Einsatz und als Knotenwichtigkeit wurde I_3 verwendet. Tabelle 5.8 zeigt die Testergebnisse. Wie man sieht, sinken die Knotengewichte nicht sehr viel, das Gewicht der Kanten sinkt jedoch recht stark. Das liegt daran, dass durch das Löschen von Kanten das Kantengewicht immer verloren geht, Knotengewichte aber nur indirekt davon betroffen sind, da Knoten erst dann gelöscht werden, wenn sie keine eingehenden Kanten mehr haben.

Außerdem zeigt sich in den Tests, dass die Adjacent-Exchange-Heuristik mit Gewichten hier deutlich besser funktioniert als die Mittelwert-Heuristik. Das kommt daher, dass

	Knotengewicht	Kantengewicht
mit Kreuzungen (Mittelwert-Heuristik)	6828,36	6487,41
ohne Kreuzungen (Mittelwert-Heuristik)	6754,45	6129,81
ohne Kreuzungen (Adj-Ex mit Gewichten)	6778,64	6280,43

Tab. 5.8.: Auswirkungen, wenn keine Kreuzungen erlaubt sind

nach Anwenden der Adjacent-Exchange-Heuristik zwar etwas mehr Kreuzungen vorhanden sind als bei der Mittelwert-Heuristik, aber die daran beteiligten Kanten deutlich leichter sind und somit das Gewicht der Kanten, die gelöscht werden müssen, kleiner ist. Der Benutzer hat also die Wahl, ob er den Gewichtsverlust für eine kreuzungsfreie Zeichnung in Kauf nimmt. Für eine solche Zeichnung sollte aber in jedem Fall die Adjacent-Exchange-Heuristik eingesetzt werden.

5.9. Einfügen von entfernten Knoten und Kanten

In diesem Kapitel soll geklärt werden, wie nützlich es ist, wenn bereits entfernte Knoten wieder eingefügt werden können. Dazu wurden Test mit folgenden Einstellungen durchgeführt: Leichte Knoten und Kanten werden bis zu einem Gewicht von fünf aussortiert, Gewicht wird nicht umgelegt, die Lagenzuordnung hat minimale Höhe, als Knotenwichtigkeit wurde I_3 benutzt und zur Kreuzungsminimierung wurde die Mittelwert-Heuristik verwendet. Tabelle 5.9 zeigt die Ergebnisse dieser Tests.

Wie man sieht, hat das nachträgliche Wiedereinfügen von Knoten recht große Auswirkungen sowohl auf das Knotengewicht wie auch auf das Kantengewicht. Das Knotengewicht steigt um durchschnittlich 3,72 Prozent und das Kantengewicht steigt um durchschnittlich 3,2 Prozent. Somit sollte das Wiedereinfügen von Knoten immer aktiviert sein, denn auch die Laufzeit des Algorithmus steigt dadurch nur leicht (siehe auch Kapitel 5.11).

	Knotengewicht	Kantengewicht
Ohne Wiedereinfügen	6580,46	6288,12
Mit Wiedereinfügen	6828,35	6487,41
Gewichtsänderung	3,72 %	3,20 %

Tab. 5.9.: Auswirkungen des Wiedereinfügens von Knoten

5.10. Varianz der Ergebnisse

Da die Ergebnisse des Algorithmus durch die verwendeten Heuristiken auch vom Zufall abhängig sind, ist es nun interessant herauszufinden, wie stark die Ergebnisse streuen. Dazu wurde der Algorithmus für jede Eingabe 500 mal ausgeführt und für Knoten- und Kantengewichte jeweils der Mittelwert und die Standardabweichung berechnet. Die Test

wurden mit folgenden Einstellungen ausgeführt: Leichte Knoten und Kanten werden bis zu einem Gewicht von fünf aussortiert, Gewicht wird auf neue Kanten umgelegt, die Lagenzuordnung soll minimale Höhe haben, die Knotenwichtigkeit ist I_3 und für die Kreuzungsminimierung wurde die Mittelwert-Heuristik verwendet. Tabelle 5.10 zeigt die Ergebnisse dieser Tests.

	Knotengewicht		Kantengewicht	
Knotenzahl	Gewicht	Standardabweichung	Gewicht	Standardabweichung
107	7290,23	11,16	7089,79	66,11
143	6008	0	5954	0
170	7258,04	18,19	7008,50	82,38
203	4561	0	4511,64	1,46
331	6639,46	6,89	6381,93	17,51
358	6018	0	5874,14	7,49
406	8999,44	38,65	8684,12	49,88
530	6863,67	65,01	6849,32	120,83
563	7358,04	34,52	6969,26	88,92
663	7440,02	23,17	7115,26	54,11
755	5919,67	23,38	5579,03	73,90
775	7232,71	35,49	6781,90	75,37
799	6930,79	2,59	6684,46	10,27
1031	6673,89	10,73	6585,68	20,41
Durchschnitt	6799,50	19,27	6576,36	47,76

Tab. 5.10.: Varianz der Ergebnisse

Wie man erkennen kann, ist die Standardabweichung bei den Knotengewichten deutlich kleiner als bei den Kantengewichten. Das ist dadurch zu erklären, dass sich das Kantengewicht direkt ändert, wenn etwas aus dem Graph gelöscht wird, während sich das Knotengewicht nur dann ändert, wenn ein Knoten oder alle darin eingehenden Kanten entfernt werden. Deshalb sind sich die Knotengewichte über mehrere Algorithmus-Durchläufe ähnlicher als die Kantengewichte.

Außerdem sieht man in der Tabelle, dass die Varianz der Ergebnisse auch stark von der Eingabe abhängt. Bei einigen Eingaben sind Abweichungen vom Mittelwert sehr gering oder sogar gar nicht vorhanden (Graphen 143, 203 und 358 Knoten), während bei anderen Eingaben die Abweichungen recht groß sind (Graphen mit 530 oder 563 Knoten).

Insgesamt kann man sagen, dass es in jedem Fall Sinn macht, den Algorithmus mehrmals laufen zu lassen und die besten Ausgaben auszuwählen. So kann man nahezu ausschließen, dass durch Zufall eine schlechte Ausgabezeichnung generiert wurde. Es ist auch kein Problem, den Algorithmus mehrmals durchlaufen zu lassen, da die benötigte Rechenzeit auch für große Graphen sehr gering ist, wie im nächsten Kapitel gezeigt wird.

5.11. Laufzeitmessungen

In diesem Kapitel soll geklärt werden, wie die unterschiedlichen Einstellungen die Laufzeit des Algorithmus beeinflussen und wie lange der Algorithmus für die einzelnen Eingaben braucht. Für die Test wurde ein Dual-Core Prozessor mit 3 GHz und 4 GB Ram benutzt, wobei der Algorithmus nicht für mehrere Prozessoren optimiert wurde und somit nur ein Rechenkern benutzt wird.

Es wird nun zuerst geklärt, wie die Laufzeit des Algorithmus von den Einstellungsmöglichkeiten abhängt und wie viel Zeit die einzelnen Teile benötigen. Dazu wurde die größte Eingabe mit 1031 Knoten benutzt, wobei am Anfang keine leichten Knoten und Kanten aussortiert werden. Tabelle 5.11 zeigt die Ergebnisse.

Einlesen des Graphen: 1 s	
Entfernen von Kreisen: 0,2 s	
Coffman-Graham: 1,6 s	PCMPS, minimale Höhe: 0,3 s
Entfernen von Knoten und Lagen: 0,5 s	
Adjacent-Exchange-Heuristik: 0,6 s	Mittelwert-Heuristik: 0,2 s
Einfügen von entfernten Knoten: 0,5 s	
kräftebasierter Algorithmus und Erstellen der Zeichnung: 2 s	

Tab. 5.11.: Laufzeiten der einzelnen Teile

Zunächst einmal benötigt der Algorithmus ungefähr eine Sekunde, um die Eingabe einzulesen (die Eingabedatei hat ca. 22700 Zeilen). Wie bereits erwähnt, benötigt das ganzzahlige lineare Programm zum entfernen der Kreise nur wenig Zeit. Selbst für die größte Eingabe sind es hier nur etwa 0,2 Sekunden. Für die anschließende Lagenzuordnung benötigt der PCMPS-Algorithmus ca. 0,3 Sekunden und der Algorithmus von Coffman-Graham ca. 1,6 Sekunden. Da sie ähnlich gut funktionieren, sollte Coffman-Graham deshalb nicht eingesetzt werden. Der Algorithmus für minimale Höhe ist ähnlich schnell wie PCMPS. Danach werden die Knoten und Lagen entfernt, bis der Graph auf die Zeichenfläche passt. Dieser Schritt benötigt ungefähr 0,5 Sekunden.

Für die Kreuzungsminimierung benötigt die Adjacent-Exchange-Heuristik ca. 0,6 Sekunden und die Mittelwert-Heuristik ca. 0,2 Sekunden. Da dieser Unterschied gering ist, sollte bei der Auswahl der Heuristik nicht die Laufzeit entscheiden, sondern, wie in Kapitel 5.7 beschrieben, das Verhältnis von Anzahl und Gewicht der Kreuzungen. Das Einfügen von entfernten Knoten benötigt anschließend ca. 0,5 Sekunden und der kräftebasierte Algorithmus benötigt zusammen mit dem Erstellen der Zeichnung ungefähr 2,5 Sekunden. Es ist aber anzumerken, dass einzelnen Laufzeiten variieren können, je nachdem wie der Graph zum entsprechenden Zeitpunkt gerade aussieht. Vor allem der kräftebasierte Algorithmus ist davon abhängig, die benötigte Zeit schwankt hier zwischen 2 und 4 Sekunden. Wie bereits erwähnt, wurden diese Test ausgeführt, ohne leichte Knoten und Kanten auszusortieren. Entfernt man diese bis zu einem Gewicht von fünf, sinkt die Laufzeit um ca. 30 Prozent, da viele Schritte mit einem kleineren Graph ausgeführt werden.

Als nächstes soll geklärt werden, wie die Laufzeit von der Eingabe abhängt. Tabelle 5.12 zeigt die Ergebnisse dieser Laufzeit-Tests. Es wurden dabei folgende Einstellungen benutzt: Leichte Kanten und Knoten werden bis zu einem Gewicht von fünf aussortiert, Gewicht wird auch auf neue Kanten umgelegt, die Lagenzuordnung hat minimale Höhe, die Knotenwichtigkeit ist I_3 und zur Kreuzungsminimierung wird die Adjacent-Exchange-Heuristik mit Gewichten eingesetzt.

Knotenzahl	Laufzeit (Sekunden)	Knotenzahl	Laufzeit (Sekunden)
107	2,68	530	4,19
143	1,04	563	3,73
170	2,66	663	3,42
203	1,36	755	3,10
331	2,24	775	3,77
358	2,06	799	3,20
406	4,27	1031	3,45

Tab. 5.12.: Gesamtlaufzeit für unterschiedliche Eingaben

Interessant ist hier, dass die Laufzeit nicht direkt von der Größe der Eingabe abhängt. Zwar benötigen die Graphen ab 406 Knoten deutlich mehr Zeit als die kleineren Graphen, es macht aber kaum einen Unterschied, ob die Eingabe 400 oder 1000 Knoten hat. Das liegt wohl daran, dass im Algorithmus die Anzahl der Knoten recht schnell reduziert wird, so dass alle enthaltenen Knoten auf die Zeichenfläche passen. Viele Schritte wie die Kreuzungsminimierung oder das kräftebasierte Verfahren werden dann immer mit ähnlich vielen Knoten ausgeführt.

5.12. Beispielausgaben des Algorithmus

Es werden nun drei Beispielausgaben vorgestellt, die den Graph mit 358 Knoten zeigen. Dabei wurden leichte Knoten und Kanten bis zu einem Gewicht von vier entfernt, da dies die beste Einstellung für den Graph ist. Die Gewichte von entfernten Knoten und Kanten wurden umgelegt, wobei auch neue Kanten eingefügt werden durften. Für die Lagenzuordnung wurde der Algorithmus für minimale Höhe eingesetzt, die Knotenwichtigkeit war I_3 und zur Kreuzungsminimierung wurde die Adjacent-Exchange-Heuristik mit Gewichten eingesetzt.

Abbildung 5.1 zeigt die Ausgabe des Algorithmus für den Stil mit nur einem Ausgangsport pro Knoten. Hier wurde das kräftebasierte Verfahren nicht eingesetzt, da es für diesen Kantenstil nicht implementiert wurde. Der Abstand zwischen zwei Lagen wurde hier recht groß gewählt, damit die vertikalen Segmente der Kanten genug Platz haben. Von den anfangs 358 Knoten sind in der Zeichnung nur noch 38 Knoten vorhanden, das Gewicht der Knoten ist aber noch zum Großteil vorhanden. In der Eingabe beträgt das Gewicht 6576, in der Zeichnung immer noch 5993. Somit werden nur 10,6 Prozent der Knoten dargestellt, die aber 91,1 Prozent des Gewichts ausmachen.

Die Abbildungen 5.2 und 5.3 zeigen die Ausgabe des Algorithmus für den orthogonalen Stil mit mehreren Ausgangsports pro Knoten und den Stil mit Bézierkurven. Diese beiden Ausgaben wurden im gleichen Durchlauf des Algorithmus erstellt, das heißt sie enthalten die gleichen Knoten und die Knoten liegen an den gleichen Positionen. Der Unterschied besteht nur darin, wie die Kanten gezeichnet werden. Da hier das kräftebasierte Verfahren eingesetzt wurde, konnte der Abstand zwischen zwei Lagen etwas geringer gewählt werden. Das hat zur Folge, dass eine Lage mehr auf die Zeichenfläche passt und sich so die Knoten- und Kantengewichte leicht erhöhen. Das Knotengewicht beträgt nun 6042 statt 5993 und das Kantengewicht steigt von 5857 auf 5886.

In Anhang A befinden sich noch weitere Beispielzeichnungen von unterschiedlichen Graphen. Die Parametereinstellungen sind die Gleichen, die Schranke für das Entfernen von leichten Knoten und Kanten wird aber jeweils angepasst, um gute Ausgaben zu erhalten.

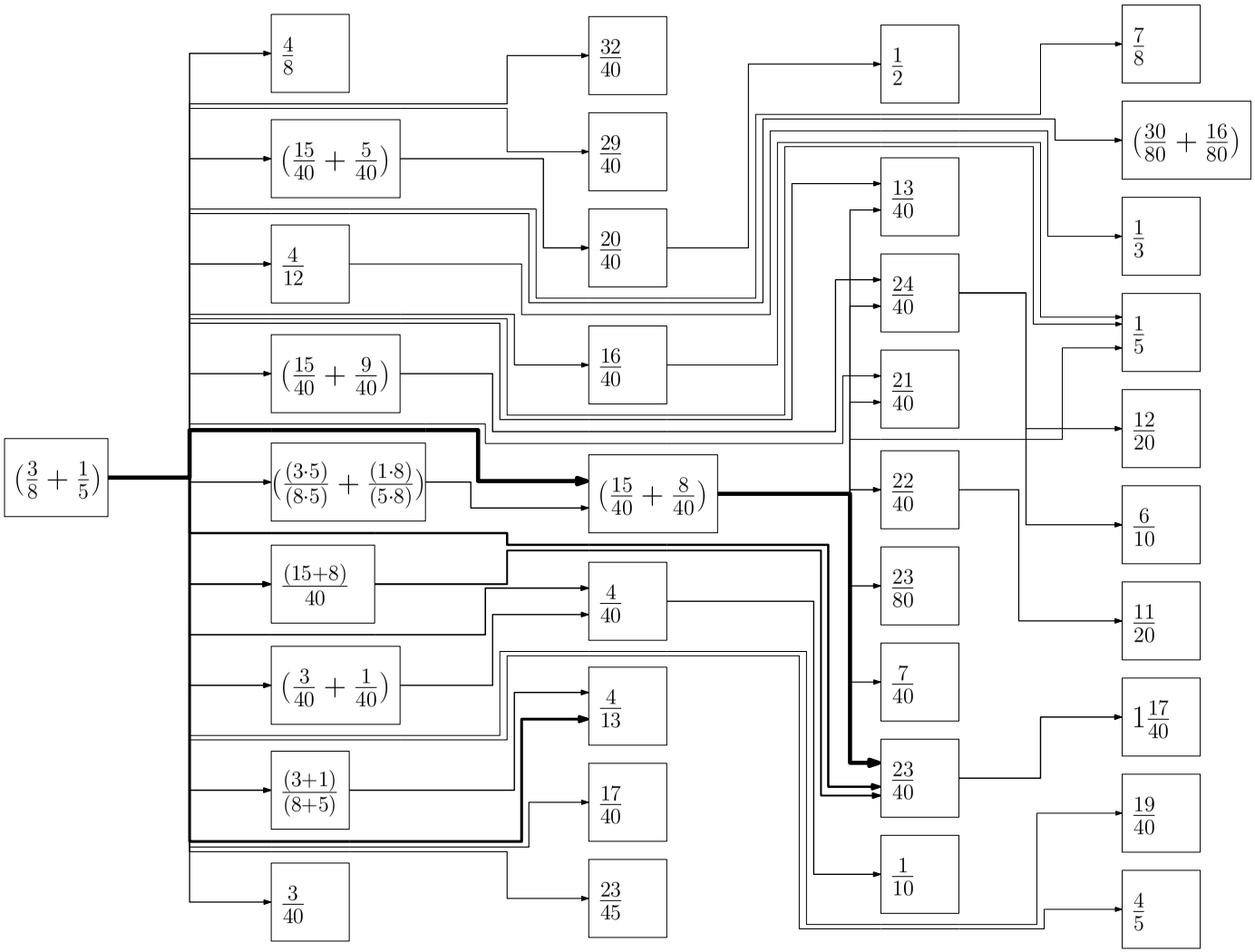


Abb. 5.1.: Orthogonaler Stil mit einem Ausgangsport pro Knoten

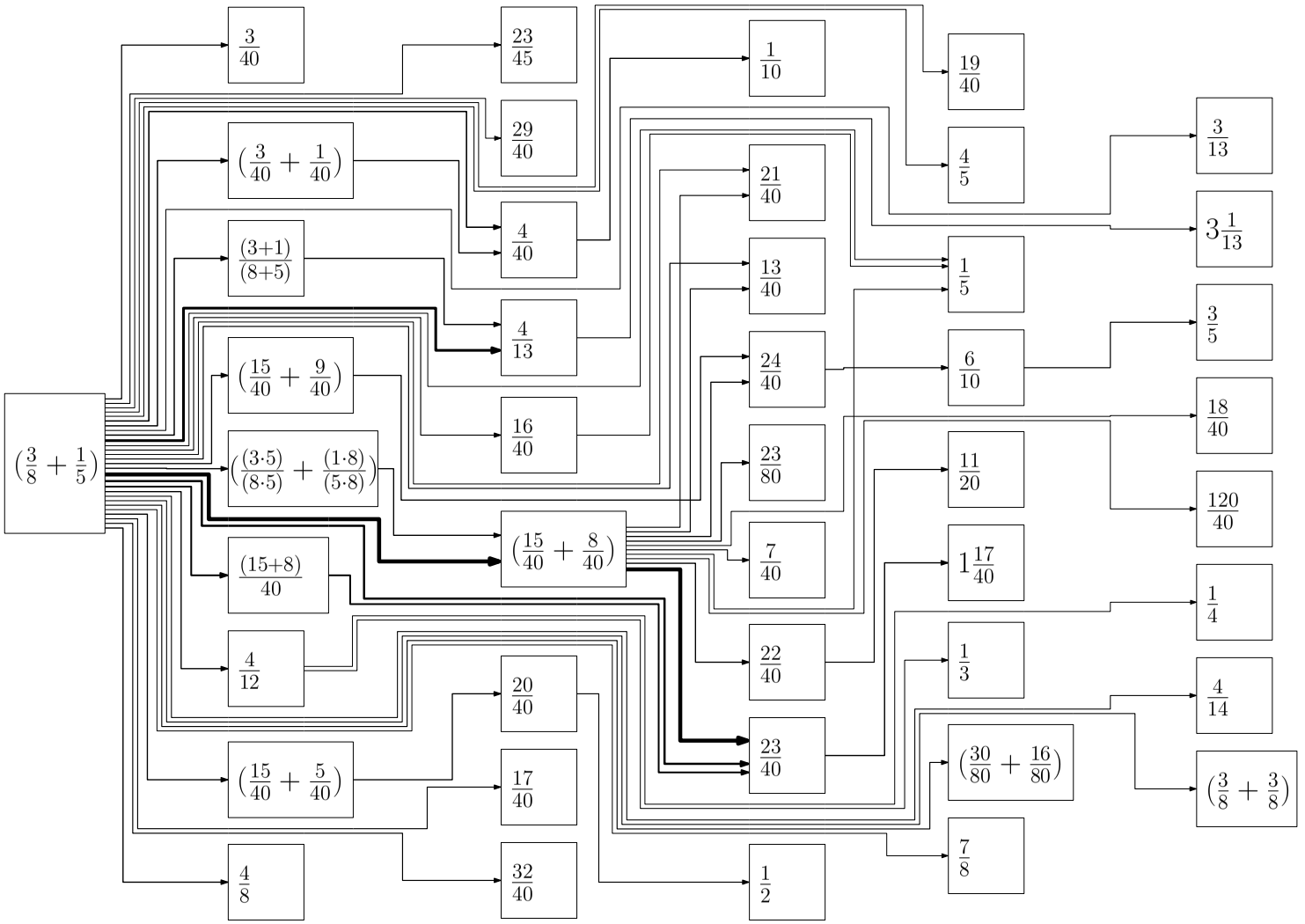


Abb. 5.2.: Orthogonaler Stil mit mehreren Ausgangsports pro Knoten

Abb. 5.3.: Kanten als Bézierkurven

6. Fazit und Ausblick

Basierend auf dem Sugiyama-Framework wurde ein Algorithmus entwickelt, der einen Teil eines Rechengraphen auf einer vorgegebenen Zeichenfläche darstellt. Weil das Zeichnen von Teilgraphen auf einer begrenzten Fläche ein bisher kaum untersuchtes Problem ist, mussten dazu Methoden entwickelt werden, welche die wichtigsten Teile des Rechengraphen herausfiltern können. Nach dem Auswählen der Knoten, die gezeichnet werden sollen, müssen diese so angeordnet werden, dass die Zeichnung für den Betrachter leicht verständlich ist. Dazu wurden unter anderem bekannte Verfahren zum Reduzieren von Kantenkreuzungen so angepasst, dass sie auch das Gewicht der Kanten, die an Kreuzungen beteiligt sind, minimieren können. Zuletzt werden die Kanten gezeichnet, wobei ein neuer Zeichenstil mit Bézierkurven entwickelt wurde, der bei Rechengraphen bisher nicht eingesetzt wurde.

Wie man in den Beispielausgaben des letzten Kapitels und denen im Anhang sieht, funktioniert der Algorithmus gut. Obwohl nur ein kleiner Teil der Knoten und Kanten des Graphen gezeichnet wird, ist das Gewicht sehr hoch. Das heißt die Informationen, die der gegebene Rechengraph enthält, sind noch zu einem Großteil in der Zeichnung vorhanden.

Da an vielen Stellen des Algorithmus unterschiedliche Einstellungen gemacht werden können, wurde in ausführlichen Tests untersucht, welche davon am besten funktionieren. Dabei hat sich herausgestellt, dass der optimale Schrankenwert zum Entfernen von leichten Knoten und Kanten mit der Größe der Eingabe wächst. Außerdem hat sich gezeigt, dass das Umlegen von Gewichten einen positiven Einfluss auf das Kantengewicht hat, weshalb das Umlegen immer aktiviert werden sollte. Welcher Lagenalgorithmus benutzt werden sollte, lässt sich auch nach den Tests nicht eindeutig sagen, da es auch von der Eingabe abhängt, welcher am besten funktioniert. Insgesamt liefern aber alle Lagenalgorithmen ähnlich gute Ergebnisse. Des Weiteren hat sich bei den Tests gezeigt, dass die verwendete Definition der Knotenwichtigkeit keine Rolle spielt, da alle genauso gut funktionieren. Bei der Reduzierung des Kantenkreuzungen muss der Benutzer entscheiden, ob die Anzahl der Kreuzungen minimiert werden soll. In diesem Fall sollte die Mittelwert-Heuristik eingesetzt werden. Normalerweise ist es jedoch sinnvoller, die Adjacent-Exchange-Heuristik zu benutzen, weil mit dieser das Gewicht der Kreuzungen deutlich reduziert werden kann, wobei sich die Anzahl von Kreuzungen nur leicht erhöht. Der Benutzter muss außerdem entscheiden, welchen Kantenstil er bevorzugt. In den meisten Fällen ist der Stil mit Bézierkurven jedoch besser geeignet, weil es hier keine Knicke in den Kanten gibt, was vor allem dann von Vorteil ist, wenn im Graph viele Kanten enthalten sind.

In der Praxis sollten zum Erstellen einer guten Zeichnung zuerst verschiedene Einstellungen für die Schranke der leichten Knoten und Kanten sowie die verschiedenen

Lagenalgorithmen getestet werden, um herauszufinden, welche Einstellungen die Zeichnungen mit dem höchsten Gewicht produzieren. Anschließend sollten mit den besten Einstellungen einige Zeichnungen generiert werden, aus denen dann per Hand die Übersichtlichste ausgesucht wird. Damit ist sichergestellt, dass die Zeichnung einerseits viele Informationen enthält und andererseits gut lesbar ist.

Obwohl der Algorithmus schon gute Zeichnungen generiert, gibt es noch einige Erweiterungen, welche die Ergebnisse noch verbessern könnten. Sie konnten im Rahmen dieser Arbeit jedoch nicht mehr in den Algorithmus integriert werden. Dazu gehört zunächst einmal die Beschriftung von Kanten mit ihren Gewichten. Durch die unterschiedliche Breite kann man zwar jetzt schon erkennen, welche Kanten wichtig sind, trotzdem wäre es gut, hier genauere Informationen in der Zeichnung abzubilden. Im orthogonalen Stil macht es Sinn, die Beschriftungen an den horizontalen Segmenten der Kanten anzubringen, wobei auch darauf geachtet werden muss, wo eventuelle Kantenkreuzungen liegen. Beim Stil mit Bézierkurven ist das Beschriften etwas komplizierter, da es keine horizontalen Segmente gibt. Hier ist es vielleicht sinnvoll, die Beschriftungen abhängig von der Steigung der Kurve zu drehen.

Eine weitere Verbesserung des Algorithmus besteht darin, auch Kanten zu zeichnen, die von rechts nach links verlaufen. Dazu dürfen beim Entfernen von Kreisen die Kanten im Feedback-Set nicht entfernt werden, sondern müssen in umgekehrter Richtung im Graph enthalten bleiben. Außerdem muss festgelegt werden, wie diese Kanten gezeichnet werden sollen. Da eine Kante ihren Startknoten immer nach rechts verlassen soll und links an ihrem Zielknoten ankommen soll, bietet sich im orthogonalen Stil eine Darstellung wie in Abbildung 6.1 an. Für Bézierkurven ist es wahrscheinlich besser, die Rückwärtskanten wie normale Kanten zu zeichnen, wobei dann nur die Pfeilspitze die Richtung angibt.

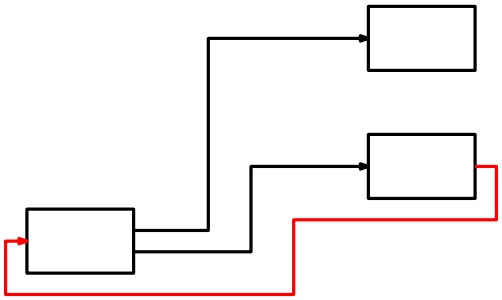
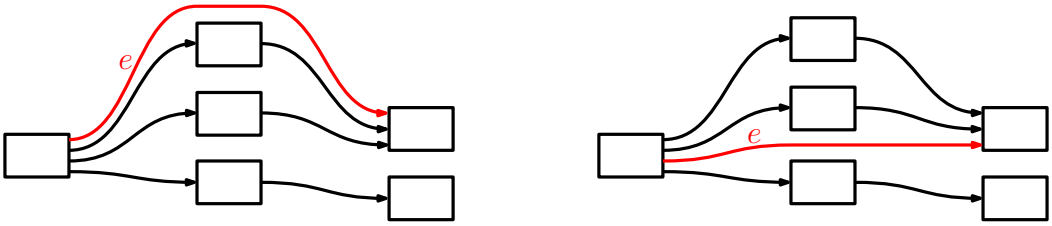


Abb. 6.1.: Darstellung einer Rückwärtskante im orthogonalen Stil

Um die Zeichnungen noch weiter zu verbessern, könnte außerdem die Kreuzungsminimierung so erweitert werden, dass sie möglichst kurze Kanten erzeugt. Denn manchmal kommt es zu Fällen wie in Abbildung 6.2a. Hier gibt es zwar keine Kreuzungen, aber die Zeichnung lässt sich trotzdem noch verbessern, indem der Dummyknoten der Kante e weiter unten gezeichnet wird, weil sich dadurch die Länge von e reduziert (Abb. 6.2b).

Des Weiteren könnten, ähnlich wie im Rechengraph in der Einleitung, unterschiedliche Farben verwendet werden, die richtige Zwischenergebnisse und fehlerhafte Rechenschritte kennzeichnen. Diese Informationen müssen dann allerdings in der Eingabe mitgeliefert



(a) Zeichnung ohne Kreuzungen **(b)** Bessere Zeichnung ohne Kreuzungen

Abb. 6.2.: Erzeugen von kurzen Kanten bei der Kreuzungsminimierung

werden. Darüber hinaus könnte auch der orthogonale Stil mit nur einem Ausgangsport pro Knoten näher untersucht werden. Vielleicht ist es hier möglich, das kräftebasierte Verfahren anzupassen und andere Erweiterungen einzubauen, um Zeichnungen in diesem Stil noch deutlich zu verbessern.

Literaturverzeichnis

[BETT98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia und Ioannis G. Tollis: *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, 1998.

[CG72] E. G. Coffman und R. L. Graham: Optimal Scheduling for Two-Processor Systems. *Acta Informatica*, 1(3):200–213, 1972.

[GJ79] Michael R. Garey und David S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Hen99] Martin Hennecke: *Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen*. Dissertation, Universität Hildesheim, 1999.

[Inc13] Gurobi Optimization Inc.: Gurobi Optimizer Reference Manual, 2013. `http://www.gurobi.com`.

[Kar72] Richard M. Karp: Reducibility among Combinatorial Problems. In: *Complexity of Computer Computations*, The IBM Research Symposia Series, Seiten 85–103. Springer US, 1972.

[KN09] Sven Oliver Krumke und Hartmut Noltemeier: *Graphentheoretische Konzepte und Algorithmen*. Vieweg+Teubner Verlag / GWV Fachverlage GmbH, 2009.

[San96] Georg Sander: A Fast Heuristic for Hierarchical Manhattan Layout. In: Franz J. Brandenburg (Herausgeber): *Proceedings of the Symposium on Graph Drawing (GD '95)*, Band 1027 der Reihe *Lecture Notes in Computer Science*, Seiten 447–458. Springer Berlin Heidelberg, 1996.

[SFHM10] Miro Spönemann, Hauke Fuhrmann, Reinhard Hanxleden und Petra Mutzel: Port Constraints in Hierarchical Layout of Data Flow Diagrams. In: David Eppstein und Emden R. Gansner (Herausgeber): *Proceedings of the 17th International Symposium (GD '09)*, Band 5849 der Reihe *Lecture Notes in Computer Science*, Seiten 135–146. Springer Berlin Heidelberg, 2010.

[STT81] Kozo Sugiyama, Shojiro Tagawa und Mitsuhiko Toda: Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981.

A. Zusätzliche Beispielausgaben

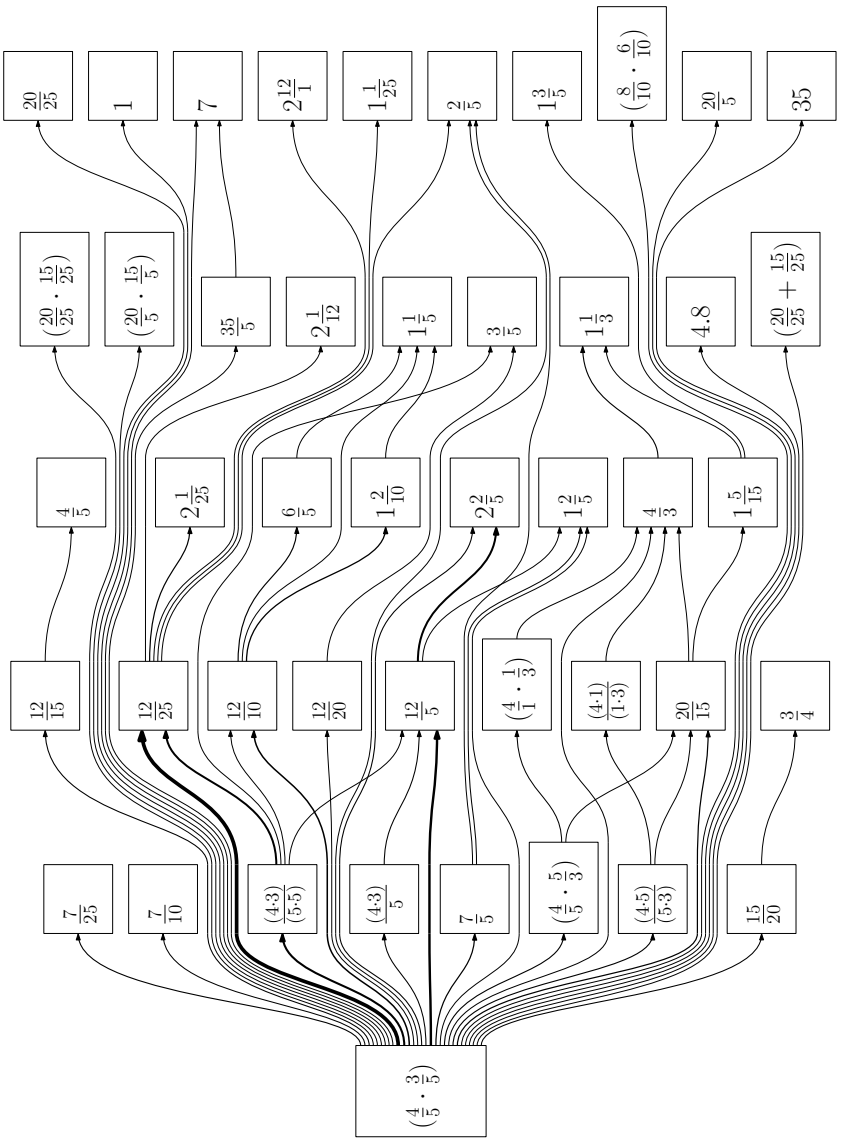


Abb. A.1.: Beispielzeichnung des Graphen mit 203 Knoten; enthaltene Knoten: 22,1 %; enthaltenes Knotengewicht: 95,2 %; enthaltene Kanten: 22,0 %; enthaltenes Kantengewicht: 94,0 %

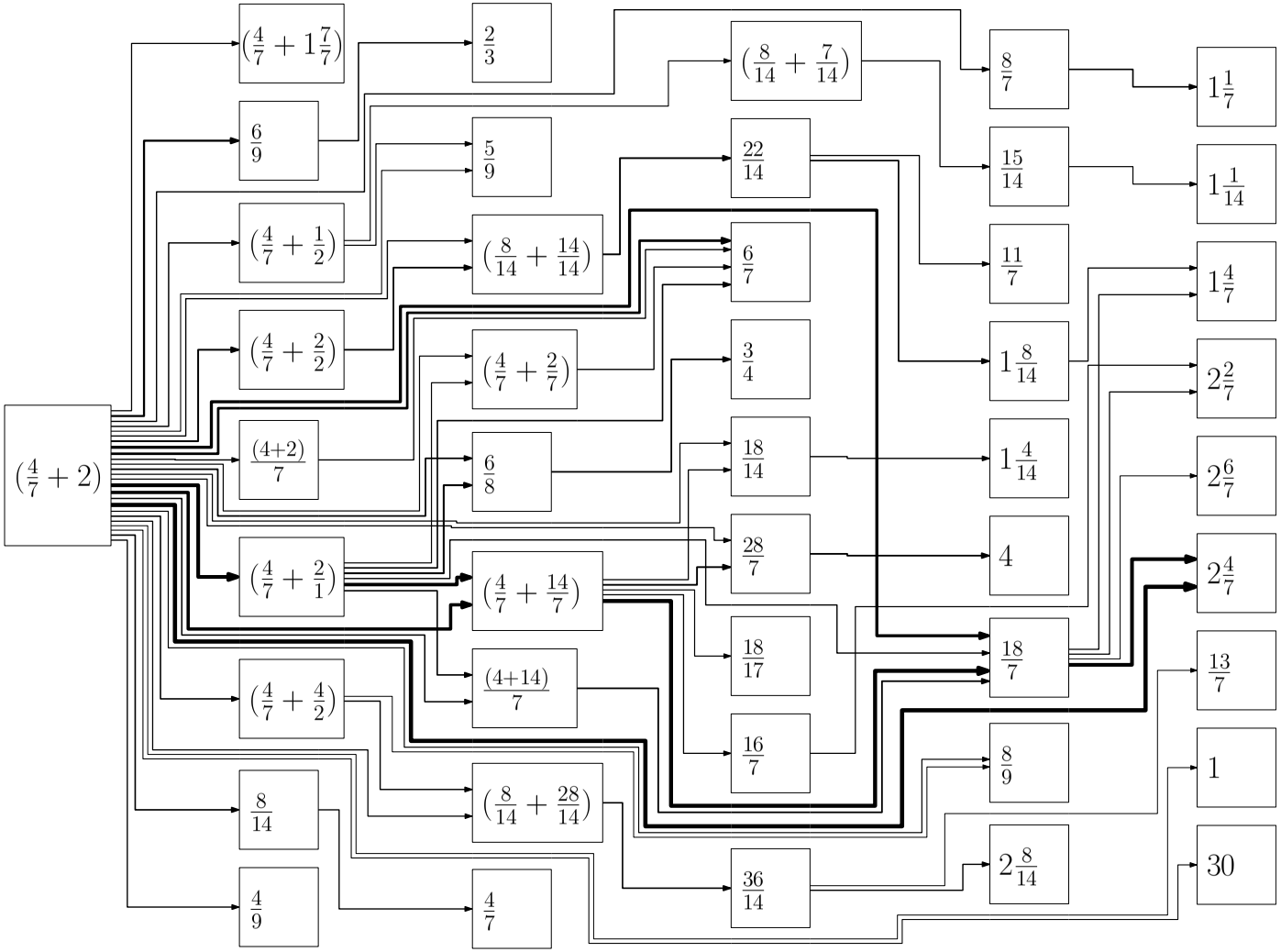


Abb. A.2.: Beispielzeichnung des Graphen mit 331 Knoten; enthaltene Knoten: 13,9 %; enthaltenes Knotengewicht: 92,9 %; enthaltene Kanten 11,4 %; enthaltenes Kantengewicht: 89,6 %

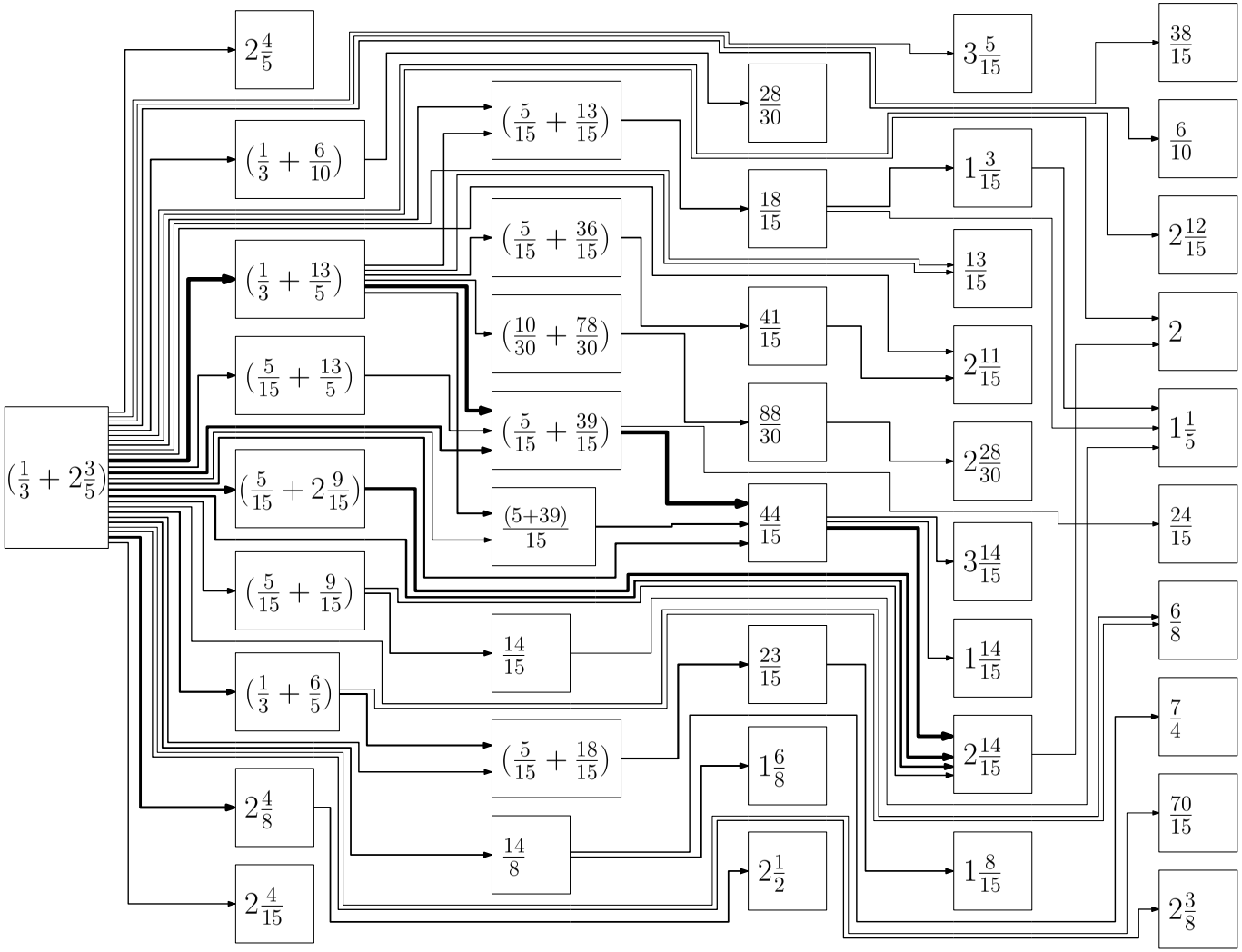


Abb. A.3.: Beispielzeichnung des Graphen mit 775 Knoten; enthaltene Knoten: 5,8 %; enthaltenes Knotengewicht: 80,6 %; enthaltene Kanten 4,6 %; enthaltenes Kantengewicht: 75,0 %

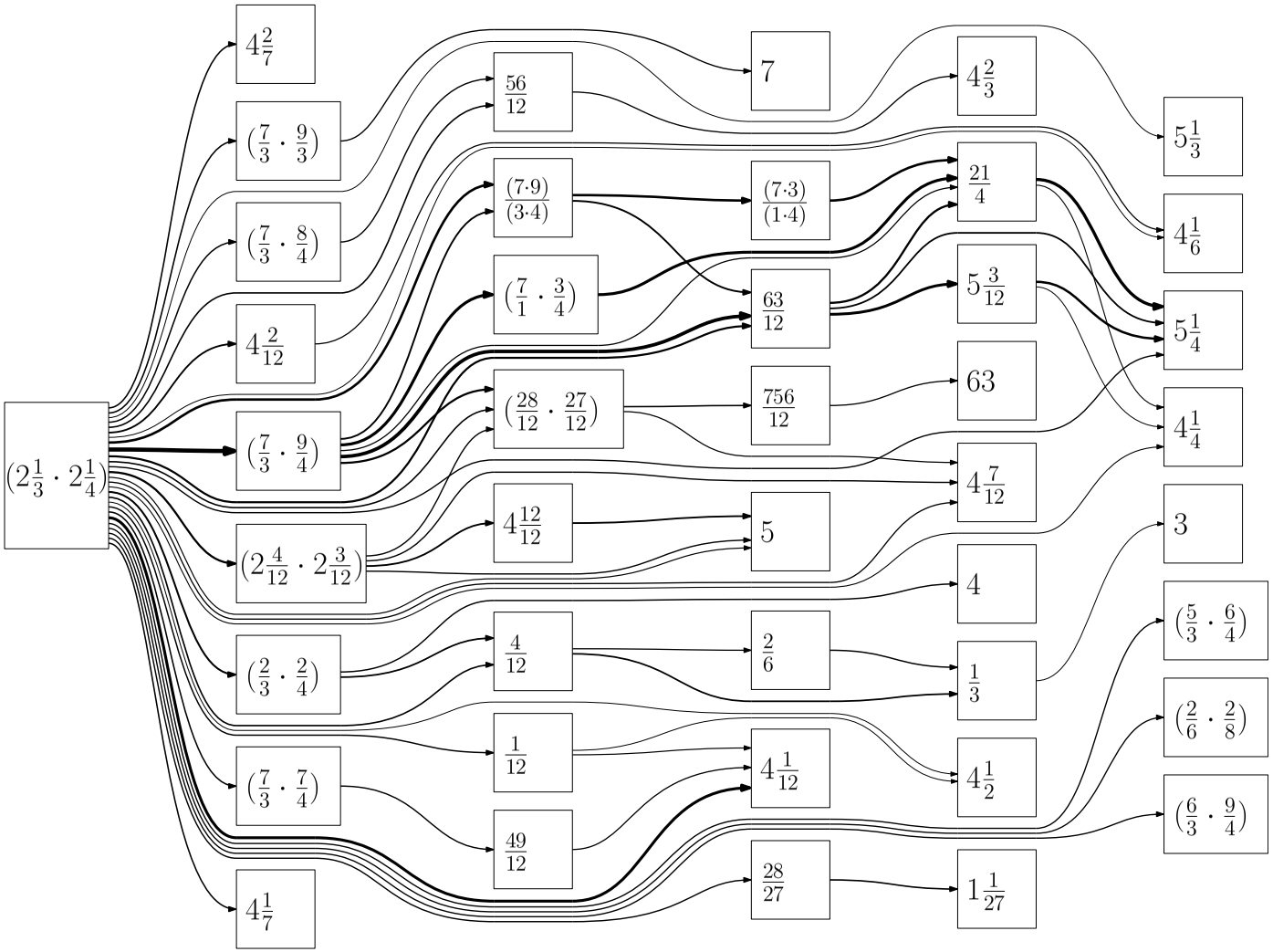


Abb. A.4.: Beispielzeichnung des Graphen mit 799 Knoten; enthaltene Knoten: 5,4 %; enthaltenes Knotengewicht: 81,7 %; enthaltene Kanten 5,3 %; enthaltenes Kantengewicht: 78,7 %

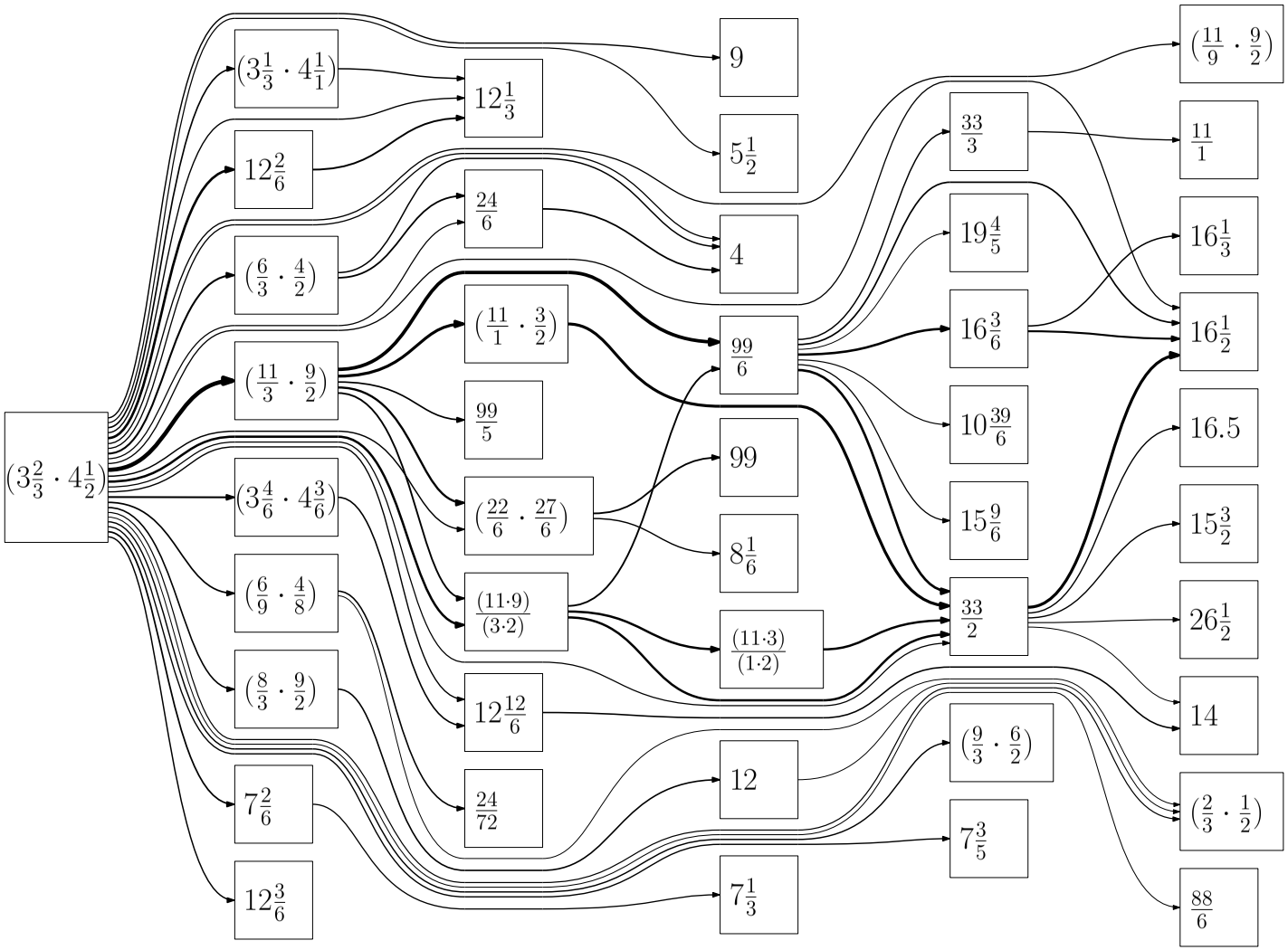


Abb. A.5.: Beispielzeichnung des Graphen mit 1031 Knoten; enthaltene Knoten: 4,7 %; enthaltenes Knotengewicht: 75,9 %; enthaltene Kanten 4,1 %; enthaltenes Kantengewicht: 74,1 %

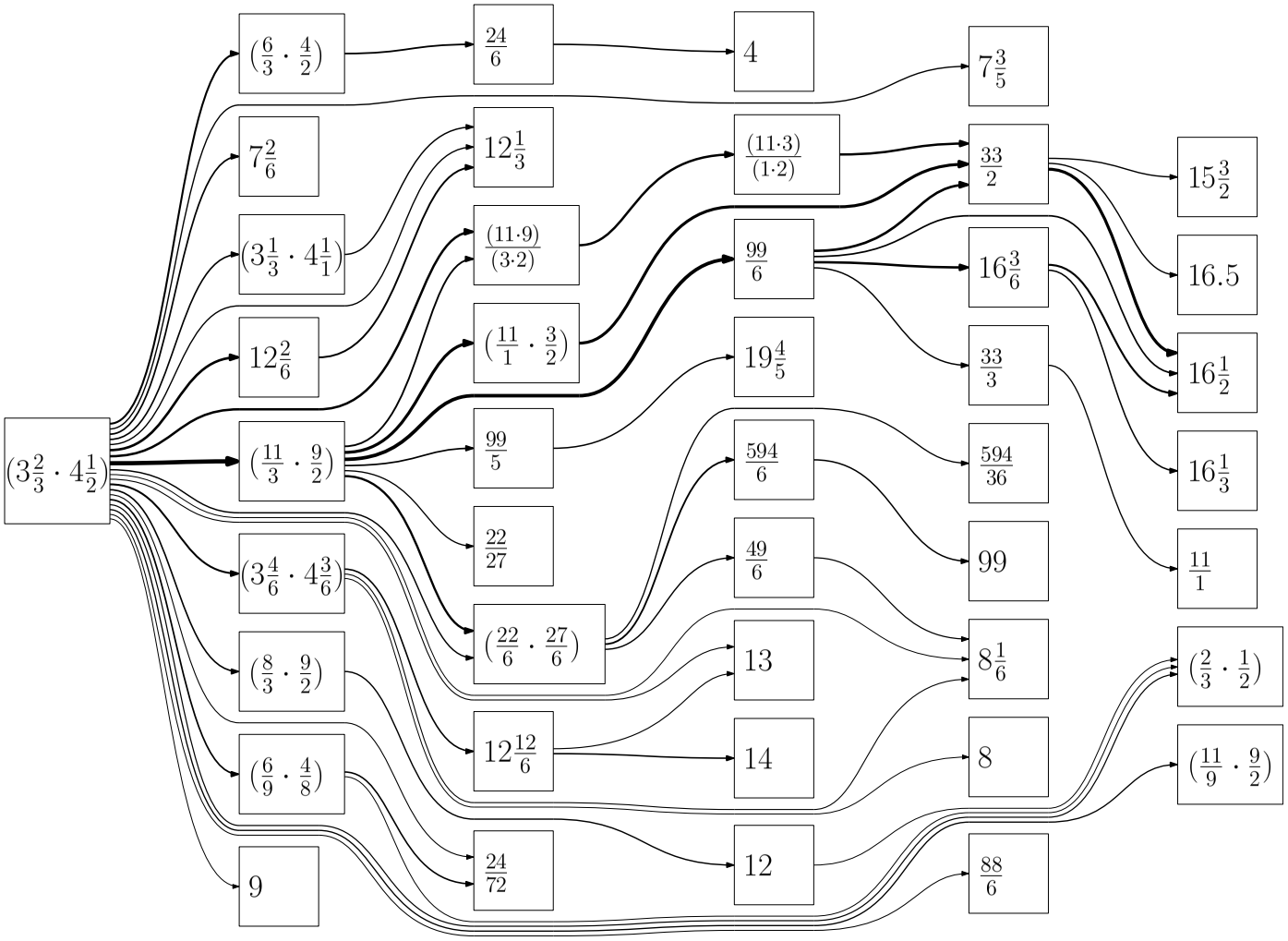


Abb. A.6.: Beispielzeichnung des Graphen mit 1031 Knoten ohne erlaubte Kantenkreuzungen; enthaltene Knoten: 4,3 %; enthaltenes Knotengewicht: 76,6 %; enthaltene Kanten 3,9 %; enthaltenes Kantengewicht: 70,2 %

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen Hilfsmittel und Quellen als die angegebenen benutzt habe. Weiterhin versichere ich, die Arbeit weder bisher noch gleichzeitig einer anderen Prüfungsbehörde vorgelegt zu haben.

Würzburg, den _____ , _____

(Julian Schuhmann)