Erweiterung und Implementierung

eines Knotenplatzierungsalgorithmus

Katja Petrat

Bachelorarbeit
eingereicht im Jahr 2014

Christian-Albrechts-Universität zu Kiel
Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme

Betreut durch: Christoph Daniel Schulze

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Überblick

Datenflussdiagramme sind eine wichtige Technik zur Modellierung der Funktionalität eines Systems. Die manuelle Erstellung ist bei kleinen Diagrammen mit wenigen Komponenten noch zu bewältigen, aber bei großen Systemen wird das schnell eine zu komplexe und zeitraubende Aufgabe. Die Automatisierung des Zeichnens solcher Diagramme ist daher von großem Interesse und kann die Entwicklungsprozesse eines Systems stark vereinfachen und beschleunigen.

KLay Layered ist ein Layoutalgorithmus für das automatische Zeichnen von solchen Datenflussdiagrammen, der den ebenenbasierten Ansatz von Sugiyama verwendet. Eine Phase des Algorithmus ist die Knotenplatzierung, welche die vertikale Position der Knoten berechnet. Gegenstand dieser Arbeit ist die Implementierung eines weiteren Knotenplatzierungsalsgorithmus, dem Algorithmus von Buchheim, Leipert und Jünger. Der Algorithmus wurde für KLay Layered so angepasst, dass er auch Ports und verschiedene Knotengrößen unterstützt.

Die Evaluation zeigt allerdings, dass der neu implementierte Algorithmus ähnliche Ergebnisse liefert wie der *Linear Segments Node Placer* und daher keine signifikanten Verbesserungen liefert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

In Technik und Wissenschaft gewinnt die automatische Informationsvisualisierung zunehmend an Bedeutung. Zum Verständnis der Struktur komplexer Abläufe in umfangreichen Softwaresystemen etwa ist die anschauliche Darstellung in Form von Diagrammen hilfreich. Kleine Diagramme lassen sich durchaus manuell übersichtlich konstruieren, aber mit zunehmender Anzahl der Relationen und Komponenten wird dies schnell zu einer so zeitraubenden Aufgabe, dass sie ohne algorithmische Unterstützung nicht mehr lösbar ist. Daher ist das automatisierte Zeichnen von Diagrammen ein vielbeachtetes Forschungsgebiet in der Informatik. Hier werden Algorithmen entworfen, welche die Elemente von Graphen nach ästhetischen Merkmalen geometrisch übersichtlich anordnen. Hierzu gehören minimale Kantenüberkreuzungen, wenige Kantenknicke oder die Größe der Zeichnung.

Diese Arbeit beschäftigt sich mit dem automatischen Layout von *Datenflussdiagrammen*. Dies sind gerichtete Graphen, deren Kanten den Fluss von Daten anzeigen, und deren Knoten die jeweiligen Quellen und Ziele beschreiben. Ein Mittel, bessere Lesbarkeit von Datenflussdiagrammen zu erzielen, ist die hierarchische Anordnung wie in Abbildung 1.1 dargestellt. Die Methode von Sugiyama [STT81] liefert bis heute die Grundlagen

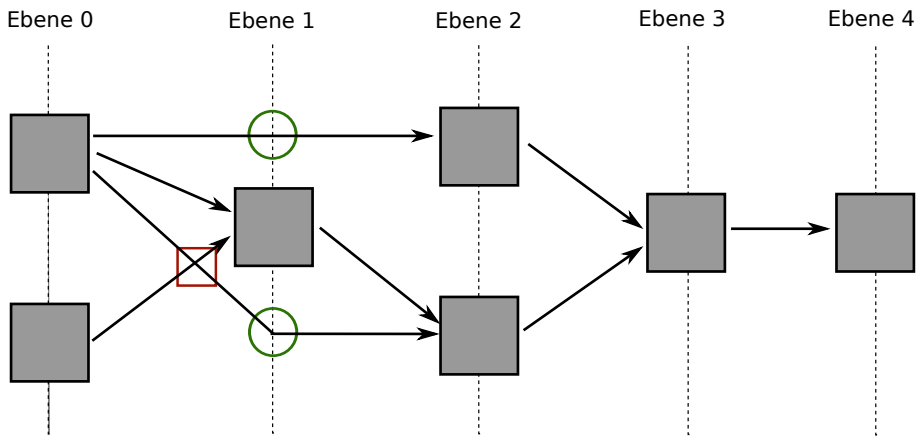


Abbildung 1.1. Hierarchische Darstellung eines Graphen. Graue Quadrate stellen Knoten dar. Diese sind auf die Ebenen 0 bis 4 verteilt. Pfeile verbinden die Knoten und zeigen alle von links nach rechts. Das entspricht der Datenflussrichtung dieses Graphen. Grüne Kreise kennzeichnen Stellen, an denen Dummyknoten eingefügt werden müssen, das rote Kästchen markiert eine Kantenüberkreuzung.

aller hierarchischer Zeichenverfahren. Die Berechnung des Layouts geschieht in drei aufeinander folgenden Phasen:

1. Die Knoten werden auf Ebenen verteilt, so dass alle Kanten möglichst in eine Richtung zeigen, etwa von oben nach unten oder von links nach rechts. Wo Kanten mehrere Ebenen überspannen, werden Dummyknoten eingefügt, so dass es nur noch direkte Verbindungen zwischen zwei aufeinanderfolgenden Ebenen gibt.
2. Die Knoten werden innerhalb der jeweiligen Ebenen derart sortiert, dass sich die Kantenüberkreuzungen minimieren.
3. Die Knoten bekommen ihre konkreten Koordinaten, um die Anzahl der Kantenknicke und die Größe der Zeichnung zu minimieren.

Es existiert eine Reihe von konkreten Implementierungen der Sugiyama-Methode. Dabei werden für die einzelnen Phasen immer bessere Algorithmen entwickelt. Auch der ebenenbasierte Algorithmus KIELER Layouters (KLay) Layered der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme der Christian-Albrechts-Universität zu Kiel optimiert und erweitert das Verfahren. KLay Layered besteht aus fünf Phasen, von denen die mittleren drei der Sugiyama-Methode folgen. In der vorgeschalteten ersten Phase des KLay Layered Algorithmus werden zunächst alle Kreise entfernt, weil alle späteren Schritte einen azyklischen Graphen voraussetzen. In der ergänzenden fünften Phase wird ein zusätzliches Kantenrouting durchgeführt, um das Layout gezielt zu verfeinern.

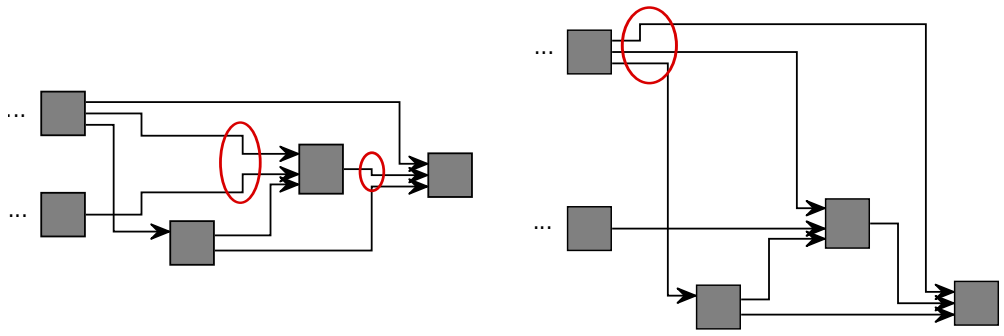
1.1 Ziele

Ziel dieser Arbeit ist es, einen zusätzlichen Algorithmus zur Knotenplatzierung zu implementieren. Es existieren bereits zwei Implementierungen: Der *Linear Segments Node Placer* [San96, Sch11] und der *BK Node Placer* [BK02, Car12]. Wie in der Abbildung 1.2 zu sehen ist, produziert der *Linear Segments Node Placer* noch unnötige Kantenknicke. Der *BK Node Placer* reduziert diese Kantenknicke, hat aber den Nachteil, dass sein Graph eine grössere Zeichnung erfordert. Das ist oft nicht zu vermeiden, wenn gerade Kanten priorisiert werden. Häufig lässt sich die Zeichnung nur verkleinern, indem Kanten um Knoten herumgeführt werden.

Der *BK Node Placer* produziert allerdings auch Kantenknicke, die ohne eine Vergrößerung des Layouts zu vermeiden wären. Der rot markierte Knick in Abbildung 1.2b resultiert etwa daraus, dass für jeden Knoten nur jeweils eine ausgehende Kante stets gerade gezeichnet wird. Solche Knicke können die Zeichnung im ungünstigsten Fall sogar vergrößern. Die Fragestellung ist nun, ob es möglich ist, mit Hilfe des Ansatzes von Buchheim, Leipert und Jünger Verbesserungen des Layouts in Bezug auf die Anzahl der Kantenknicke und die Größe zu erreichen [BJL01]. Da KLay Layered Ports und verschiedene Knotengrößen unterstützt, wird der Algorithmus in dieser Arbeit hierfür angepasst. Resultierende Graphen werden untersucht und evaluiert.

1.2 Verwandte Arbeiten

Gansner hat einen einfachen Knotenplatzierungsalgorithmus entwickelt, der eine Positionierung mittels Prioritätswerten vornimmt [GKN02]. Die Knoten werden anhand des



(a) *Linear Segments Node Placer* **(b)** *BK Node Placer*

Abbildung 1.2. Die Darstellung zeigt zweimal den gleichen Ausschnitt eines mit KLay Layered generierten Graphenlayouts. Bis auf die Knotenplatzierungen sind alle Einstellungen identisch. Die roten Markierungen kennzeichnen unnötige Kantenknicke.

Medians der mit ihnen verbundenen Knoten auf benachbarten Ebenen ausgerichtet. Damit wird insbesondere die Kantenlänge minimiert, jedoch nicht automatisch die Anzahl der Kantenknicke.

Sander liefert einen ersten Ansatz, um möglichst viele Kanten gerade zu zeichnen und damit die Anzahl der Kantenknicke zu reduzieren [San96]. Man fasst hierfür bestimmte Knotengruppen zu linearen Segmenten zusammen. Darin lassen sich alle Kanten gerade zeichnen. Diese linearen Segmente können entweder aus einzelnen Knoten bestehen oder aus miteinander verbundenen Dummyknoten. Die topologische Sortierung der Segmente und die Ausrichtung mittels eines kräftebasierten Verfahrens ergeben zusätzlich zu den reduzierten Kantenknicken ein balanciertes Layout. Ein Nachteil dieses Verfahrens ist es, dass ausschließlich Kanten zwischen Dummyknoten definitiv gerade gezeichnet werden. Mit Originalknoten verbundene Kanten werden hierbei vernachlässigt.

Die Idee von Brandes und Köpf beruht darauf, die Knoten zu Blöcken zu vereinen, deren Kanten gerade dargestellt werden können [BK02]. Ein Block enthält miteinander verbundene Knoten aus aufeinanderfolgenden Ebenen. Pro Ebene wird maximal ein Knoten dem Block hinzugefügt, es kann jedoch mehrere Kandidaten hierfür geben. Im Algorithmus werden deshalb verbundene Dummyknoten priorisiert und die Auswahl der Kandidaten je nach Laufrichtung durch den Graphen variiert. Die sich daraus ergebenen Layouts kombiniert man am Ende zu einem balancierten Ergebnis. Nachteilig ist, dass jeweils nur eine ausgehende Kante pro Knoten definitiv gerade gezeichnet wird.

Wie Sander teilen Buchheim, Leipert und Jünger den Graphen in lineare Segmente ein und fassen diese wiederum zu Klassen zusammen [BJL01]. Der Algorithmus schiebt die Knoten in mehreren Durchläufen immer weiter zusammen. Zuerst werden die Knoten innerhalb ihrer Klassen so dicht wie möglich angeordnet und dann die Klassen als Ganzes ausgerichtet. Auf diese Art wird ein kompaktes Layout generiert. Der wesentliche Unterschied zu den zwei vorher genannten Algorithmen ist, dass die Positionierung der Dummyknoten und der Originalknoten nacheinander ausgeführt werden. Im Vergleich zu Sander kann dies zu einer Verbesserung der Originalknotenplatzierung führen.

Zur Bewertung der Qualität von Graphenlayouts definieren Sugiyama, Tagawa und

Toda in ihrer Arbeit ästhetische Kriterien [STT81]. Helen Purchase diskutiert und evaluiert diesbezügliche Kriterien ausführlich, etwa anhand diverser Testreihen [Pur02].

1.3 Gliederung

Zum Abschluss dieser Einführung gebe ich einen kurzen Überblick zum Aufbau dieser Arbeit. Kapitel 2 beinhaltet die nötigen theoretischen Grundlagen eines ebenenbasierten Graphenlayouts und beschreibt den allgemeinen Aufbau von KLay Layered. Dabei wird insbesondere auf die einzelnen Phasen des Algorithmus eingegangen. Kapitel 3 bildet den Kern dieser Arbeit. Dort wird die Bedeutung der Knotenplatzierung innerhalb von Layoutalgorithmen erläutert und der Algorithmus von Buchheim, Leipert und Jünger detailliert vorgestellt. Da es an einigen Stellen nötig ist, den Algorithmus anzupassen, werden anschließend die vorgenommenen Modifikationen beschrieben. Die konkrete Umsetzung des Algorithmus findet am Ende des Kapitels ihren Platz, dort wird auch auf technische Details wie Datenstrukturen eingegangen. Kapitel 4 beschreibt die Ergebnisse der Evaluation, die in die abschließende Bewertung in Kapitel 5 mit einfließen.

Kapitel 2

Grundlagen

In diesem Kapitel sollen zunächst relevante Grundlagen der Graphentheorie erläutert werden. Im zweiten Abschnitt des Kapitels folgt eine Beschreibung der Sugiyama-Methode und wichtiger Aspekte des ebenenbasierten Graphenlayouts. Zum Abschluss wird der KIELER KLay Layered-Algorithmus eingeführt.

2.1 Terminologie und Definitionen

Im Kontext dieser Arbeit werden Datenflussdiagramme betrachtet. Diese basieren grundsätzlich auf gerichteten Graphen. Wir betrachten ohne Einschränkung der Allgemeingültigkeit hier die Berechnung eines von links nach rechts ausgerichteten Graphenlayouts.

2.1.1 Definition. Ein *gerichteter Graph* ist ein Paar $G = (V, E)$. V ist eine endliche Menge an *Knoten* und $E \subseteq VxV$ eine Relation auf V, welche die Menge der *Kanten* zwischen den Knoten beschreibt. Die gerichteten Kanten werden durch geordnete Knotenpaare (v, w) dargestellt, wobei $v \in V$ der Anfangsknoten der Kante ist und $w \in V$ der Endknoten. *Nachbarn* eines Knotens v sind alle Knoten, die direkt mit v verbunden sind : $\delta_G(u) = \{w \in V | (u, w) \in E\}$.

Kanten können mit Knoten über explizite Anknüpfungspunkte, den *Ports*, verbunden sein. Dann wird ein solcher Graph wie folgt definiert.

2.1.2 Definition. Ein *gerichteter, portbasierter Graph* ist ein Tupel $G = (V, E, P, \vartheta)$, in dem V eine endliche Menge an Knoten darstellt. P ist eine endliche Menge Ports und $\vartheta : P \to V$ eine Funktion, die Ports bestimmten Knoten zuordnet. $E \subseteq PxP$ ist eine Menge an Kanten, die Knoten über Ports verbinden. Kanten sind geordnete Paare (p, q), wobei $p \in P$ der Ausgangssport der Kante ist und $q \in P$ der Zielport.

Ein derartiger Graph kann darüber hinaus in Ebenen eingeteilt werden, welche die Menge der Knoten partitionieren. Die Definition lautet dann:

2.1.3 Definition. Ein *ebenenbasierter, gerichteter, portbasierter Graph* ist ein Tupel $G = (V, E, P, \lambda, \vartheta)$, wobei V, E, P wie in 2.1.2 definiert sind. Die zusätzliche Funktion $\lambda(v) : V \to \{1, \ldots, k\}$ ordnet jeden Knoten v einer Ebene von 1 bis k zu.

Für die Berechnung eines hierarchischen Layouts ist die Einfachheit eines Graphen eine weitere wichtige Eigenschaft.

2.1.4 Definition. Ein ebenenbasierter, gerichteter, portbasierter Graph heißt *einfach*, wenn alle Kanten nur Knoten zweier aufeinanderfolgender Ebenen verbinden, also wenn gilt $\lambda(w) - \lambda(v) = 1$ für alle $(v, w) \in E$.

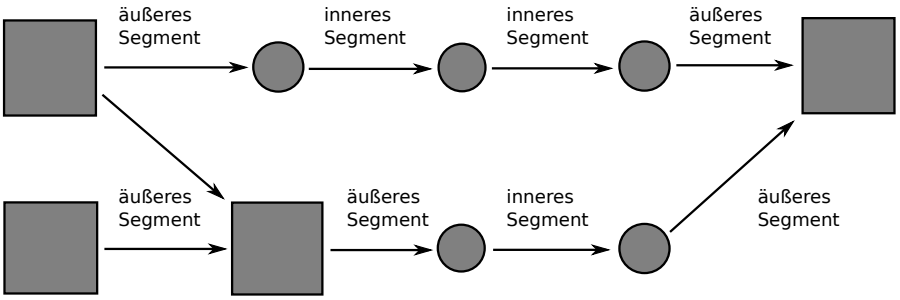


Abbildung 2.1. Äußere und innere Kantensegmente, Quadrate symbolisieren Originalknoten, Kreise virtuelle Knoten.

Ein Graph ist demnach nicht einfach, wenn es ebenenüberspannende Kanten $(v, w) \in E$ mit $\lambda(w) - \lambda(v) > 1$ gibt. Diese bezeichnen wir im weiteren Verlauf als *lange Kanten*.

Es ist möglich, einen Graphen durch Aufteilung der langen Kanten in kurze Segmente in einen einfachen Graphen zu überführen. Dort wo eine lange Kante eine Ebene schneidet, werden virtuelle Knoten $v \in \bar{V}$, sogenannte *Dummyknoten*, eingefügt (Abb. 1.1). $\bar{V}$ bezeichnet die Menge aller virtuellen Knoten, $\bar{E}$ die angepasste Kantenmenge. Kantenabschnitte, die zwei virtuelle Knoten verbinden, werden als *innere Segmente* bezeichnet und alle anderen als *äußere Segmente* (Abb.2.1). Dies führt zu dem neuen Graphen $G = (V \cup \bar{V}, \bar{E}, P, \lambda, p)$.

Um weitere Begriffe einführen zu können, wird ein geordneter Graph benötigt.

2.1.5 Definition. Ein Graph $G = (V \cup \bar{V}, \bar{E}, P, \lambda, p)$ heißt *geordnet*, wenn für jede Ebene $l \in \{1, \ldots, k\}$ eine Permutation seiner Knoten festgelegt ist. Die Position eines Knotens in der Ebene wird durch $pos(v)$ angegeben. Der *obere Bruder* $s_-(v)$ ist der Knoten an Position $pos(v) - 1$ innerhalb derselben Ebene. Existiert kein oberer Bruder, lautet die Notation $s_-(v) = *$. Der *untere Bruder* $s_+(v)$ befindet sich analog an $pos(v) + 1$ und hat ebenso den Wert $s_+(v) = *$, wenn er nicht existiert.

Nachdem nun wichtige Begriffe und Notationen eingeführt worden sind, folgt im nächsten Abschnitt eine Beschreibung der einzelnen Phasen der Sugiyama-Methode zum Zeichnen hierarchisch aufgebauter Graphen.

2.2 Ebenenbasiertes Graphenlayout

Ein bedeutendes Verfahren zum Zeichnen gerichteter Graphen ist die Methode von Sugiyama [STT81]. Sie hebt den hierarchischen Aufbau eines solchen Graphen durch Aufteilung der Knoten auf diskrete, parallele Ebenen und möglichst in dieselbe Richtung zeigende Kanten hervor. Die Popularität dieser Methode läßt sich daran ablesen, dass sie in vielen Frameworks zum Zeichnen von Graphen Verwendung findet. Prominente Beispiele sind etwa Graphlet, OGDF oder dot als Teil der GraphViz Bibliothek [Him97, CGJ+07, GKN02, GN00].

Die Sugiyama-Methode besteht aus drei Phasen, in denen schrittweise ästhetische Eigenschaften der Graphenzeichnung optimiert werden sollen. Die Methode lässt sich erweitern, so dass sie aus den fünf folgenden Phasen besteht:

1. Entfernen von Zyklen.

   Die Eingabe ist ein gerichteter Graph und die Ausgabe ein azyklischer, gerichteter Graph, bei dem möglichst alle Kanten in dieselbe Richtung zeigen. Kreisfreiheit ist zwingende Voraussetzung für die Ebenenzuweisung.

2. Unterteilung in Ebenen.

   Diese Phase bekommt als Eingabe einen gerichteten, kreisfreien Graphen und verteilt die Knoten möglichst gleichmäßig auf Ebenen. Lange Kanten werden in Segmente unterteilt und Dummyknoten eingefügt. Die Anzahl der Knoten pro Ebene beeinflusst die Höhe der Zeichnung und die Anzahl der Ebenen die Breite. Je mehr Dummyknoten eingefügt werden, umso aufwendiger werden alle folgenden Berechnungen. Die Ausgabe ist ein gerichteter, kreisfreier, ebenenbasierter, einfacher Graph.

3. Kreuzungsminimierung.

   Diese Phase minimiert die Kantenüberkreuzungen durch Bestimmung von Knotenpermutationen innerhalb der Ebenen. Dabei werden aus Effizienzgründen üblicherweise nur benachbarte Ebenen betrachtet. Dieses Problem ist bereits bei zwei Ebenen NP-schwer und wird deshalb in der Regel durch Heuristiken gelöst. Nach dieser Phase ist die Reihenfolge der Knoten in der Ebene fix.

4. Knotenplatzierung.

   Diese Phase positioniert die Knoten innerhalb der Ebenen, ohne deren Reihenfolge zu ändern. Die Abstände zwischen den Knoten einer Ebene bestimmen die Höhe der Zeichnung. Die Ausrichtung verbundener Knoten zueinander beeinflußt die einzelnen Kantenlängen und die Anzahl der benötigten Kantenknicke. Als Resultat stehen die y-Koordinaten fest.

5. Kantenrouting.

   Das Kantenrouting legt die x-Koordinaten der Knoten fest und berechnet die genauen Verläufe der Kanten.

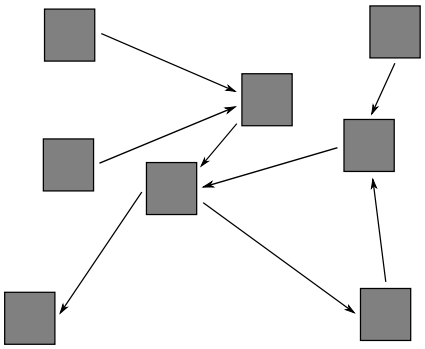
Abbildung 2.2 illustriert die beschriebenen fünf Phasen der erweiterten Sugiyama-Methode anhand eines Beispielgraphen. Jeder dieser Schritte kann separat betrachtet werden, und es gibt eine Vielzahl alternativer Lösungsansätze für jede der Phasen.

2.3 KLay Layered

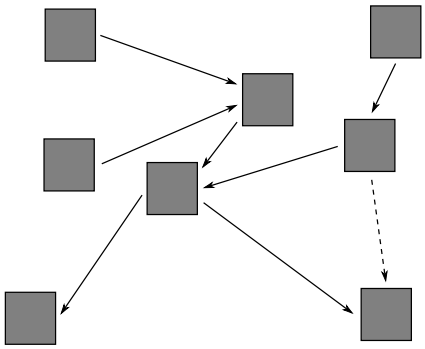
KLay Layered ist ein ebenenbasierter Layoutalgorithmus, der in das Projekt Kiel Integrated Evironment for Layout Eclipse RichClient, kurz KIELER, eingebettet ist. KLay Layered basiert auf der Sugiyama-Methode und erweitert sie um die bereits beschriebenen zwei zusätzlichen Phasen. Die in KLay Layered implementierten Algorithmen für die einzelnen Phasen werden nun beschrieben.

2.3.1 Die 5 Phasen von KLay Layered

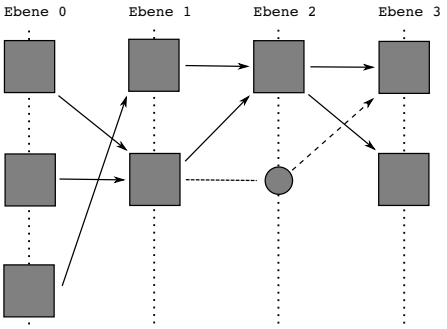
Phase 1: Entfernen von Zyklen Kreise werden entfernt, indem man einige Kanten wie in Abbildung 2.3 umdreht. Eine Menge von Kanten, deren Umkehrung einen



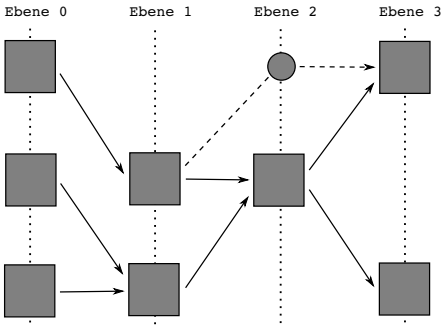
(a) Eingabe: Gerichteter, zyklischer Graph



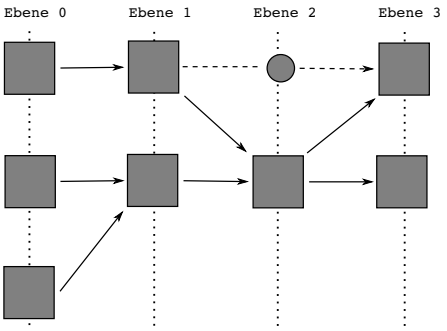
(b) Phase 1: Entfernen von Zyklen



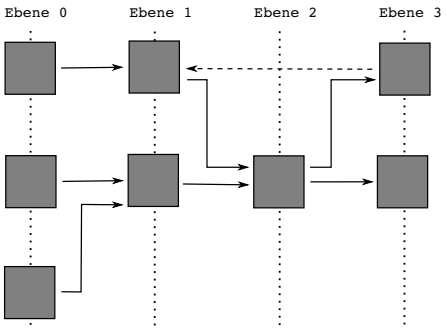
(c) Phase 2: Unterteilung in Ebenen.



(d) Phase 3: Kreuzungsminimierung.

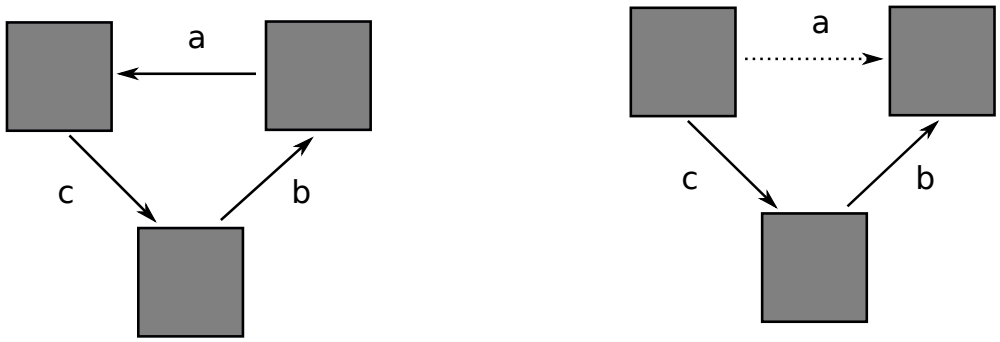


(e) Phase 4: Knotenplatzierung



(f) Phase 5: Orthogonales Kantenrouting.

Abbildung 2.2. Die fünf Phasen der erweiterten Sugiyama-Methode. Quadrate stellen die Originalknoten dar, Kreise symbolisieren die Dummyknoten



(a) Graph mit Kreis

(b) kreisfreier Graph, Kante a wurde umgedreht

Abbildung 2.3. Entfernen von Kreisen durch Umdrehen einer Kante, Feedback Set = {a}.

Graphen kreisfrei macht, nennen wir *Feedback Set*. Es ist einfach, ein beliebiges Feedback Set zu finden, aber NP-schwer, ein minimales zu ermitteln. Für die Qualität des Layouts ist dies aber wichtig, weil alle umgekehrten Kanten in der resultierenden Zeichnung gegen den Datenfluss zeigen.

In KLay Layered ist der *GreedyCykleBreaking*-Algorithmus nach Eades implementiert [ELS93], dem folgende Idee zugrunde liegt: Eine Kante ist niemals Teil eines Kreises, wenn sie mit einer Quelle oder Senke verbunden ist. Führt man eine eindeutige Ordnung der Knoten ein, so läßt sich die Kantenmenge leicht in zwei Teilmengen partitionieren. Die eine enthält alle Kanten, deren Quellknoten vor den Zielknoten auftreten. Die andere Teilmenge umfasst die Kanten, die entgegen der Ordnung gerichtet sind und bildet ein Feedback Set. Dreht man diese Kanten um, ist der resultierende Graph kreisfrei.

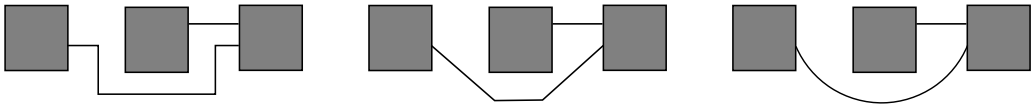
Phase 2: Unterteilung in Ebenen Eine einfache Möglichkeit zur Verteilung der Knoten auf Ebenen ist die Berechnung eines *Spannbaums*. Den ersten Knoten weist man einer beliebigen Ebene L_i zu. Alle seine Nachbarn werden je nach Kantenrichtung auf die Ebenen L_{i-1} oder L_{i+1} verteilt. Anschließend wiederholt man mit allen Nachbarn diesen Vorgang rekursiv.

In den meisten Fällen soll die Verteilung der Knoten aber bestimmten zusätzlichen Anforderungen genügen. Die Anzahl der Knoten pro Ebene, die Ebenenanzahl oder die Menge der benötigten Dummyknoten soll begrenzt sein. Dadurch wird das Problem weitaus komplexer.

In KLay Layered sind der *LongestPath*- und der *NetworkSimplexLayering*-Algorithmus implementiert [GKNV93].

Der *LongestPath* beruht auf dem topologischen Sortieren. Er hat den großen Vorteil, nur lineare Laufzeit zu benötigen und die minimale Höhe des Layouts zu finden. Er tendiert aber dafür zu breiten unteren Ebenen und großer Anzahl an Dummyknoten.

Der *NetworkSimplexLayering* ist komplexer und minimiert die Anzahl der Dummyknoten. Er basiert auf der Simplex-Methode, einem numerisch-iterativen Rechenverfahren. Die Grundidee solcher Verfahren besteht darin, so lange mögliche Lösungen zu ge-



(a) Orthogonal Edges **(b)** Polyline Edges **(c)** Spline Edges

Abbildung 2.4. Verschiedene Möglichkeiten des Kantenrouting.

nerieren, bis ein optimales Ergebnis gefunden ist. In unserem Fall können aus jedem Spannbaum des zugrundeliegenden Graphen verschiedene gültige Ebenenzuweisungen berechnet werden. Aus den Resultaten wählt der Algorithmus das Layout mit der günstigsten Anzahl an Dummyknoten aus.

Phase 3: Kreuzungsminimierung Das Konzept der gängigen Heuristiken ist das *Layer-by-Layer-Sweep*-Prinzip. Dabei wird der Graph Ebene für Ebene in beide Richtungen durchlaufen. In jedem Schritt werden immer nur je zwei benachbarte Ebenen betrachtet. Die Knotenreihenfolge auf der zuvor besuchten Schicht ist fix; die Reihenfolge auf der aktuellen Schicht wird so geändert, dass es danach weniger Kantenüberkreuzungen gibt. Die Permutation der Knoten wird in KLay Layered anhand des *Barycenter*-Wertes bestimmt. Der Barycenter-Wert und damit die neue Position eines Knotens ist der Mittelwert aller Nachfolger- oder Vorgängerpositionen. Die Topologie der Zeichnung ist nach dieser Phase festgelegt.

Phase 4: Knotenplatzierung Aktuell sind in KLay Layered zwei verschiedene Algortihmen implementiert: der *Linear Segments Node Placer* und der *BK Node Placer*.

Der *Linear Segments Node Placer* basiert auf dem Algorithmus von Sander [San96]. Er fasst Knoten zu linearen Segmenten zusammen, welche die gleiche y-Koordinate bekommen können. Segmente bestehen entweder aus einem Originalknoten oder allen Dummyknoten einer langen Kante. Diese linearen Segmente werden mit Hilfe der *Barycenter* Heuristik platziert.

Der *BK Node Placer* orientiert sich an dem Algorithmus von Brandes und Köpf [BK02]. Er vereinigt Knoten zu Blöcken. Alle Knoten eines Blockes werden so platziert, dass die zwischen ihnen verlaufenden Kanten ohne Knickpunkte gezeichnet werden können. Blöcke können aus Dummyknoten und Originalknoten bestehen, enthalten aber immer maximal einen Knoten pro Ebene. Der *BK Node Placer* zeichnet damit nicht nur Kanten zwischen Dummyknoten gerade, sondern bezieht auch die Originalknoten mit ein. Insofern produziert er mehr gerade Kanten als der *Linear Segments Node Placer*.

Phase 5: Kantenrouting In der letzten Phase werden die x-Koordinaten der Knoten und die genauen Routen der Kanten berechnet. Kanten können auf verschiedene Arten gezeichnet werden. KLay Layered unterstützt orthogonale Kantenführung, kurvige Kanten oder Kanten als Linienzüge wie in Abbildung 2.4. Der einfachste implementierte Algorithmus ist der *PolylineEdgeRouter*, der lediglich Kantenknicke an den Positionen der Dummyknoten einfügt. Der *SplineEdgeRouter* benutzt die Positionen der Dummyknoten zur Ausrichtung der splinebasierten Kurven. Der *OrthogonalEdgeRouter* orientiert sich am Algorithmus von Sander [San04] und berechnet eine gerade Kantenführung mit

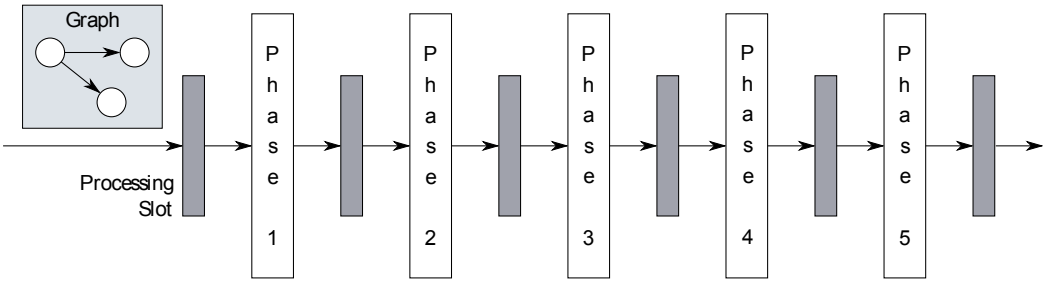


Abbildung 2.5. Die Darstellung zeigt die Struktur von KLay Layered mit den 5 Hauptphasen und den Slots für Intermediate Processors [Sch11]

ausschließlich rechtwinkeligen Kantenknicken. Am Ende dieser Phase ist die Breite der einzelnen Ebenen und der gesamten Zeichnung gesetzt.

2.3.2 Intermediate Processors

Eine Besonderheit von KLay Layered ist seine dynamische Architektur. Damit die fünf eigentlichen Phasen möglichst einfach bleiben, gibt es *Intermediate Processors*, die zusätzliche Aufgaben übernehmen können [Sch11]. Dies sind kleine Programme, die in sogenannten *Slots* vor, zwischen und nach den Hauptphasen ausgeführt werden können (Abb.2.5). Ein Intermediate Processor lässt sich potentiell in verschiedenen Slots ausführen, und in jedem Slot können sich mehrere dieser Prozessoren befinden.

Intermediate Processors erweitern KLay Layered für eine größere Bandbreite an Graphen. Es gibt den *Graph Transformer*, der KLay Layered auch das Berechnen von Layouts nach oben, unten oder links erlaubt.

Die Vermeidung von Programmcodedublikaten ist ein weiterer Vorteil. Der *Long Edge Splitter* etwa fügt die Dummyknoten ein, die für alle Phasen des Layoutalgorithmus benötigt werden. Ohne diesen zwischengeschalteten Prozessor müsste man die Berechnung für jede Phase neu implementieren.

Kapitel 3

Knotenplatzierung

Nach der Kreuzungsminimierung steht die Topologie der Zeichnung fest. Der Graph ist kreisfrei, die Knoten sind auf die einzelnen Ebenen verteilt und ihre Reihenfolge innerhalb der Ebenen ist berechnet. Funktion der Knotenplatzierung ist es, allen Knoten y-Koordinaten zuzuweisen. Die Bedeutung dieser Phase für die Qualität der Zeichnung wird anhand eines Beipiels demonstriert. Im ersten Abschnitt werden die Ziele umrissen, im zweiten der Algorithmus von Buchheim et al. beschrieben. Der dritte Abschnitt enthält die Anpassungen für KLay Layered und der letzte die technischen Details der Implementierung.

3.1 Problemstellung

Formal lässt sich eine gültige Knotenplatzierung wie folgt definieren:

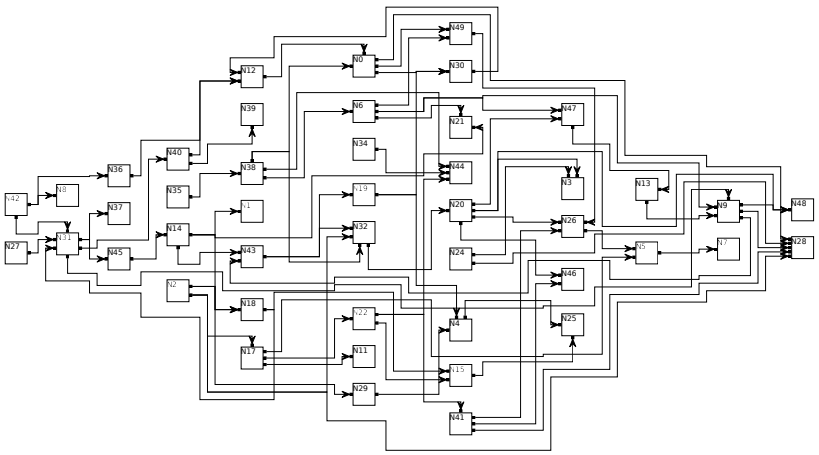
3.1.1 Definition. Sei $m(v_1, v_2)$ der Mindestabstand von v_1 zu v_2, wobei $v_1 = s_-(v_2)$. Die Funktion $y(v_i)$ liefert die y-Koordinate. Eine Platzierung der Knoten $v_i \in V$ ist dann gültig, wenn $y(v_i) - y(s_-(v_i)) \geqslant m(s_-(v_i), v_i)$ für alle $v_i \in V$ gilt.

Zum Einstieg betrachten wir ein Beispiel, das die Bedeutung der Knotenplatzierung innerhalb von ebenenbasierten Layoutalgorithmen veranschaulicht. Beide Layouts in Abbildung 3.1 wurden mit Klay Layered berechnet. Für die Erstellung der ersten Zeichnung wurde der *Simple Node Placer* angewendet, für die zweite der *BK Node Placer* nach Brandes und Köpf. Alle anderen Einstellungen sind für beide Layoutberechnungen identisch.

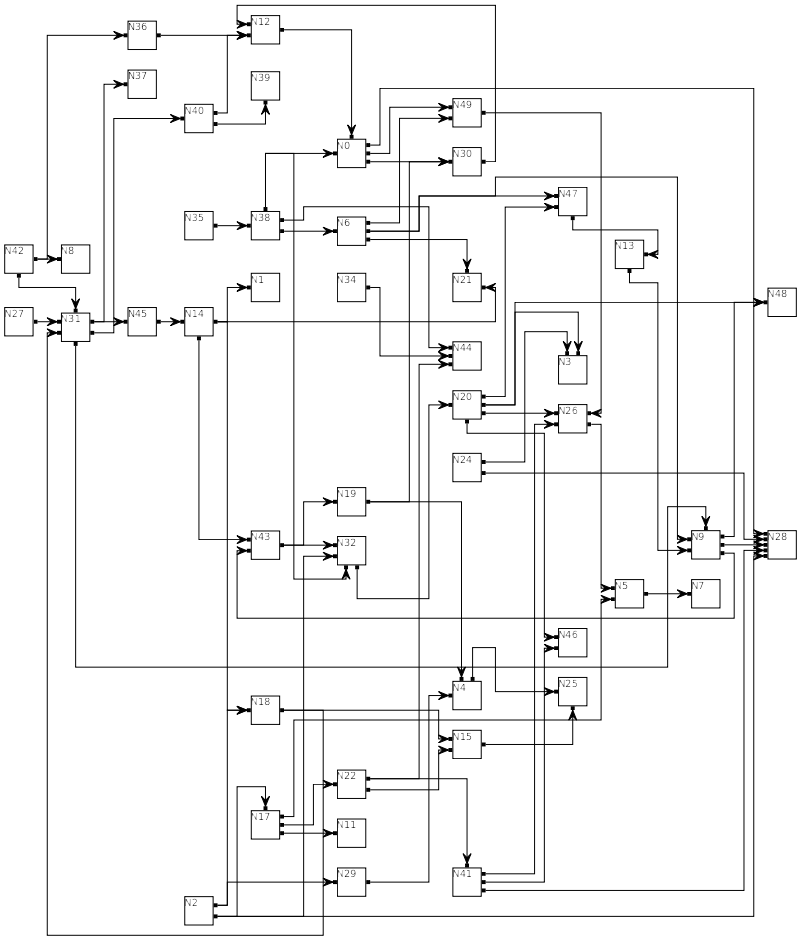
Man erkennt deutliche Qualitätsunterschiede. Allein durch weniger Knicke in den Kanten wird eine erhebliche Verbesserung der Lesbarkeit erreicht. Das menschliche Auge kann geraden Kanten leichter folgen und Datenflüsse besser identifizieren. Sugiyama et al. haben ästhetische Kriterien zur Qualitätsverbesserung evaluiert. Wichtige Kriterien, die durch geeignete Knotenpositionierung beeinflusst werden können, sind:

- Minimierung der Kantenknicke mittels gleicher y-Koordinaten für ausgewählte Knotengruppen
- Minimierung der Kantenlängen durch Ausrichten der Knoten an den Positionen ihrer Nachbarn
- Minimierung der Höhe einer Zeichnung durch möglichst geringe Abstände aller Knoten einer Ebene zueinander

Es ist eine komplexe Aufgabe, mehrere Kriterien gleichzeitig zu optimieren. In KLay Layered sind bereits zwei Knotenplatzierungsalgorithmen implementiert, die unterschiedliche Lösungsansätze bieten. Der *Linear Segments Node Placer* zeichnet lange Kanten gerade



(a) einfaches Layout, Spaghetti-Effekt deutlich zu erkennen



(b) verbessertes Layout, lange Kanten wurden begradigt

Abbildung 3.1. Einfluss der Knotenplatzierung auf die Qualität des Layouts

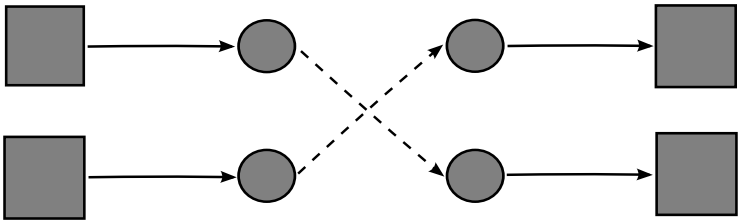


Abbildung 3.2. Kreuzung innerer Segmente. Die Kanten können nicht gerade gezeichnet werden, ohne Eigenschaft (**C**) zu verletzen.

und liefert ein sehr kompaktes Layout, produziert aber vermeidbare Kantenknicke. Der *BK Node Placer* reduziert die Kantenknicke, allerdings auf Kosten der Kompaktheit. Ziel dieser Arbeit ist es, den Algorithmus von Buchheim et al. zu implementieren und zu untersuchen. Es sollen alle drei Ansätze hinsichtlich der oben genannten ästhetischen Eigenschaften miteinander verglichen werden.

3.2 Algorithmus von Buchheim, Jünger und Leipert

In diesem Abschnitt wird nun der Algorithmus von Buchheim, Leipert und Jünger vorgestellt. Zuerst werden die Eigenschaften des erzeugten Layouts erläutert und dann die einzelnen Schritte des Algorithmus beschrieben.

3.2.1 Eigenschaften des Layouts

Der Algorithmus erzeugt ein Layout, dass die folgenden Eigenschaften garantiert:

(**A**) Die Reihenfolge der Knoten in den Ebenen bleibt erhalten und Mindestabstände werden beachtet.

(**B**) Alle Kantensegmente werden gerade gezeichnet.

(**C**) Innere Segmente werden horizontal gerade gezeichnet. Dadurch besitzt jede Kante maximal 4 Knicke und der sogenannte *Spaghetti- Effekt*, also viele sehr lange Kanten mit vielen Knicken, wird vermieden.

Wenn sich innere Segmente wie in Abbildung 3.2 überschneiden, können Eigenschaft (**A**) und (**C**) nicht gleichzeitig erfüllt werden; diese Ausnahme muss man gesondert behandeln. In dieser Arbeit wird das Problem gelöst, indem lange Kanten, die sich überkreuzen, in zwei lineare Segmente aufgeteilt werden. Diese können dann jeweils gerade gezeichnet werden. Bei der Berechnung der linearen Segmente des Graphen kann dies direkt mit eingebunden werden.

3.2.2 Beschreibung des Algorithmus

Für einen gerichteten, azyklischen Eingabegraphen soll eine Knotenplatzierung berechnet werden, welche die Eigenschaften (**A**), (**B**) und (**C**) erfüllt. Der Algorithmus besteht aus zwei Schritten:

Algorithmus 1 placeDummy

```
placeTop(v, c)
placeBottom(v, c)
for all v ∈ V ∪ V̄ do
    y(v) ← (y_top(v) + y_bottom(v))/2
end for
```

Algorithmus 2 computePositionTop

```
computeTopClasses(c)
for all i = 0 to the number of classes do
    for all v of c_i do
        if v is not placed yet then
            placeTop(v, c)
        end if
    end for
    adjustTopClass(i, c)
end for
```

1. Platzierung der Dummyknoten
2. Platzierung der regulären Knoten

Wir beziehen uns in der Beschreibung auf ein von links nach rechts gerichtetes Layout, weil KLay Layered mit dieser Orientierung arbeitet. Die Position der Knoten innerhalb der Ebenen wird entsprechend durch die y-Koordinate festgelegt.

Platzierung der Dummyknoten

Alle Knoten einer langen Kante erhalten die gleiche y-Koordinate, und der Knotenabstand in y-Richtung wird unter Einhaltung des Mindestabstandes m soweit wie möglich minimiert. Die Positionierung der Dummyknoten vollzieht sich in drei Schritten (Algorithmus 1, *placeDummy*):

1. Alle Knoten werden so weit wie möglich nach oben platziert und als oberes Layout $y_{top}(v_i)$ gespeichert (Algorithmus 2, *computePositionTop*).
2. Alle Knoten werden so weit wie möglich nach unten platziert und als unteres Layout $y_{bottom}(v_i)$ gespeichert (*computePositionBottom*). Dieser Algorithmus funktioniert analog zu *computePositionTop* und wird hier deshalb nicht explizit angegeben.
3. Aus oberem und unterem Layout wird das Durchschnittslayout $y(v_i)$ ermittelt:

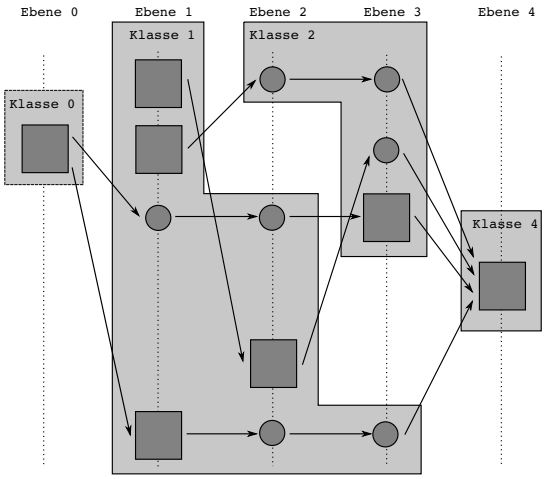
$$y(v_i) = \frac{y_{top}(v_i) + y_{bottom}(v_i)}{2}$$

Zur Berechnung der oberen und unteren Positionen wird der Graph zunächst zerlegt. Knoten werden zu linearen Segmenten zusammengefasst.

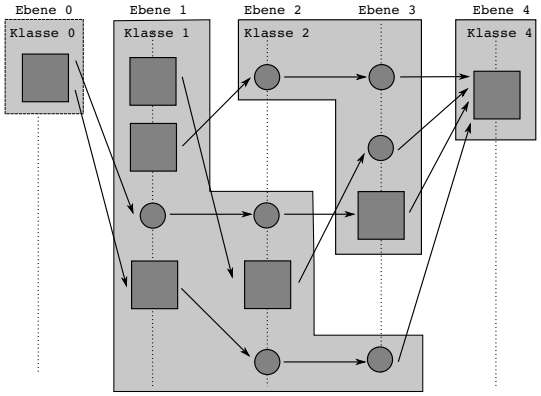
3.2.1 Definition. Für ein lineares Segment $L(v)$ eines Knotens v gilt:

$$L(v) = \begin{cases} \{v\}, \text{ für } v \in V \\ \{v' \in V\} |\ v, v' \text{ gehören zur selben Kante, für } v \in \bar{V} \end{cases}$$

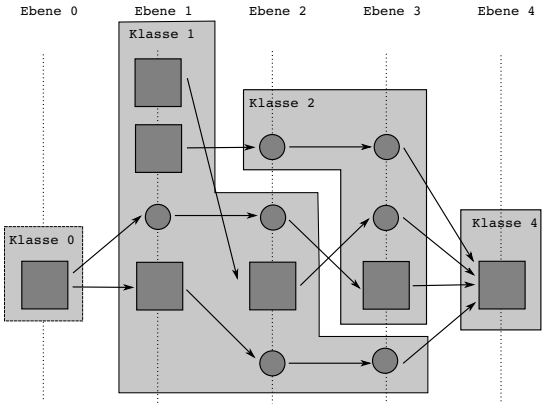
Der Algorithmus 3 fügt diese linearen Segmente anschließend zu Klassen zusammen wie in Abbildung 3.3 illustriert. Dafür werden die obersten Knoten v des Graphen von



(a) Aufteilung eines Diagramms in obere Klassen. Die Klassen werden nach der Ebene benannt, in der sich der oberste linke Knoten befindet. In einer Klasse werden alle Knoten eines linearen Segmentes und deren untere Brüder zusammengefasst, die noch keiner anderen Klasse angehören.



(b) Positionierung der Knoten soweit oben wie möglich, wobei immer alle Knoten eines linearen Segmentes die gleiche y-Koordinate erhalten.



(c) Kombination der Klassen. Eine Klasse wird jeweils anhand der bereits platzierten Klassen ausgerichtet.

Abbildung 3.3. Berechnung des oberen Layouts

Algorithmus 3 computeTopClasses

```
for all levels l = 0 to k do
    c' ← l
    for all v on level l traversed top down do
        if c(v) is not initialized yet then
            for all v' ∈ L(v) do
                c(v') ← c'
            end for
        else
            c' ← c(v)
        end if
    end for
end for
```

Algorithmus 4 placeTop(v, c)

```
p ← −∞
for all v' ∈ L(v) do
    if s_−(v') ≠ ∗ and c(s_−(v')) = c(v')) then
        if s_−(v') is not placed yet then
            placeTop(s_−(v'))
        end if
        p ← max{p, y_top()s_−(v')) + m(s_(v'), v')}
    end if
    if p = −∞ then
        p ← 0
    end if
    for all v' ∈ L(v) do
        y_top(v') ← p
    end for
end for
```

links nach rechts durchlaufen. Falls v noch zu keiner Klasse gehört, wird eine neue Klasse C erstellt, die v enthält. Rekursiv werden dann alle Knoten einer langen Kante und alle unteren Brüder eingefügt, wenn diese noch zu keiner Klasse gehören. Die Klasse erfüllt mindestens die folgenden Bedingungen:

1. $v \in C$

2. wenn $w \in C$, dann gilt auch $L(w) \subseteq C$.

3. wenn $w \in C$ und wenn der untere Bruder $s_+(w) \neq *$ noch keiner Klasse zugeordnet war, dann wird $s_+(w)$ der Klasse C hinzugefügt.

Zunächst betrachtet man alle Klassen separat und platziert die Knoten innerhalb ihrer jeweiligen Klasse. Dieses wird durch den Algorithmus 4, *placeTop*, realisiert. Alle Knoten eines linearen Segmentes bekommen die gleiche y-Koordinate, um Eigenschaft (**C**) zu genügen. Der oberste Knoten einer Klasse erhält die Position 0, alle anderen Knoten derselben Klasse werden so nah wie möglich herangeschoben, ohne wiederum Eigenschaft (**A**) zu verletzen. Sie bekommen also die folgende Position:

$$max\{y(w) + m(w, s_+(w)) \mid w \in W\},$$
$$W = C \cap \{s_-(w) \mid w \in L(w)\}$$

W ist die Menge aller oberen Brüder eines linearen Segmentes der gleichen Klasse. Ist diese Menge leer, werden alle Knoten des linearen Segments auf die Position 0 gesetzt.

Nachdem alle relativen Knotenpositionen innerhalb der Klassen berechnet wurden, kombiniert Algorithmus 5, *adjustTopClasses*, die Klassen miteinander.

Algorithmus 5 adjustTopClass(i, c)

```
d ← infty
for all v of class i do
    if s+(v') ≠ * and c(s+(v)) ≠ i then
        d ← min{d, y_top(s+(v)) − y_top(v) − m(v, s+(v))}
    end if
end for
if d = ∞ then
    let D be a heap;
    for all v of class i do
        for all w ∈ δ(v) do
            if c(w) < i then
                push y_top(w) − y_top(v) to D
            end if
        end for
    end for
    let d_1, ... d_s be the values in D
    if s = 0 then
        d ← 0;
    else
        d ← d_⌈s/2⌉
    end if
end if
for all v of class i do
    y_top(v) ← y_top(v) + d
end for
```

Die Klassen werden von oben nach unten platziert. Falls C untere Brüder in einer anderen Klasse hat, die bereits platziert sind, wird C so nah wie möglich um d von oben an diese Klasse herangeschoben, ohne die relativen Positionen innerhalb der Klasse zu verändern.

$$d = min\{(y(w) - y(s_-(w)) - m(s_-(w), w) \mid w \in W'\},$$
$$W' = ((V \cup \bar{V}) \backslash C) \cap \{s_+(w) \mid w \in C\}$$

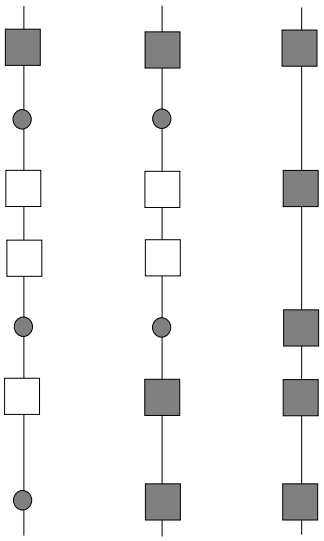
W' ist die Menge der unteren Brüder anderer Klassen, die bereits platziert worden sind. Falls keine solchen unteren Brüder existieren, wird die ganze Klasse C so verschoben, dass die Gesamtlänge aller Kantensegmente, also folgende Summe, minimiert wird:

$$\Sigma|y(v) - y(w)|$$

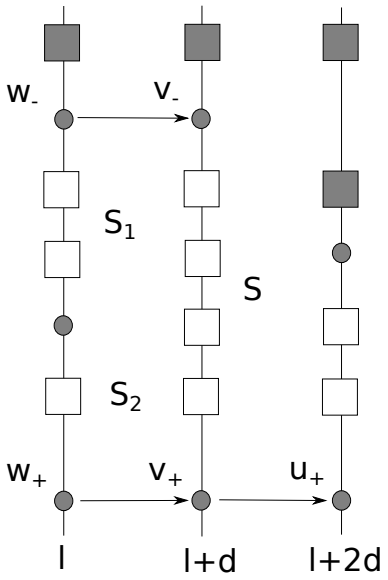
Das obere Layout für die Dummyknotenpositionen ist damit berechnet. Für das untere Layout geht man analog vor. Beide Knotenplatzierungen sind gültig. Es werden also alle Mindestabstände zwischen je zwei Knoten einer Ebene eingehalten. Daher ist auch die Knotenplatzierung aus den Durchschnittswerten der beiden Layouts ebenso gültig:

$$\begin{aligned} y(s_+(v_i)) &= \frac{y_{top}(s_+(v_i)) + y_{bottom}(s_+(v_i))}{2} \\ &\geqslant \frac{y_{top}(v_i) + m(v_i, s_+(v_i)) + y_{bottom}(v_i) + m(v_i, s_+(v_i))}{2} \\ &= \frac{y_{top}(v_i) + y_{bottom}(v_i)}{2} + \frac{2 * m(v_i, s_+(v_i))}{2} \\ &= y(v_i) + m(v_i, s_+(v_i)) \end{aligned}$$

Man sieht, dass in jedem Fall für jedes Knotenpaar der Mindestabstand in dem resultierenden Layout eingehalten wird.



(a) Innere und äußere Sequenzen regulärer Knoten (dunkelgraue Quadrate stellen äußere Sequenzen dar und helle innere Sequenzen)



(b) S_1 und S_2 sind Nachbarfolgen von Sequenz S und d ist die Laufrichtung. S ist eine innere Sequenz in Bezug auf Ebene l, da es virtuelle Nachbarknoten w_- und w_+ der Knoten v_- und v_+ gibt. In Bezug auf Ebene $l + 2d$ ist S eine äußere Sequenz, da kein u_- existiert.

Abbildung 3.4. Sequenzen

Platzierung der Originalknoten

Die Platzierung der Originalknoten dient der Minimierung der Länge und der Knicke äußerer Kantensegmente. Die Positionen der Dummyknoten sind bereits fixiert. Daher können Sequenzen von Originalknoten unabhängig voneinander platziert werden, wenn sie sich in derselben Ebene befinden und mindestens ein Dummyknoten dazwischen liegt. Eine *reguläre Sequenz* ist eine maximale Folge von nicht durch Dummyknoten unterbrochenen regulären Knoten derselben Ebene. Eine *innere Sequenz* wird oben und unten von einem Dummyknoten begrenzt, eine *äußere Sequenz* maximal von einem, wie in Abbildung 3.4a dargestellt.

Wir suchen ein Layout, das die Summe der Abstände aller regulären Knoten einer Sequenz $v_1, \ldots, v_r$ zu ihren Nachbarn $v \in \bar{\delta}$ minimiert:

$$\Sigma_{i=1}^{r} \Sigma_{v \in \bar{\delta}} |y(v) - y(v_i)|$$

Der Algorithmus durchläuft alle Ebenen des Graphen nacheinander; im ersten Schritt von links nach rechts und im zweiten in umgekehrter Richtung. Alle Sequenzen, deren begrenzende Dummyknoten bereits den Mindestabstand aufweisen, können zu diesem Zeitpunkt bereits als platziert angenommen werden. Alle regulären Knoten solcher Sequenzen haben bereits exakt den Mindestabstand zueinander. Alle anderen regulären

Algorithmus 6 placeRegular

```
let D be a heap
let P ∈ {true, false}^V̄
for all b_− ∈ V̄ do
    let b_+ be the next dummy node to the bottom of b_−
    if b_+ ≠ ∗ then
        D(b_−) ← 0
        if y(b_+) − y(b_−) = m(b_−, b_+) then
            P(b_−) ← true
        else
            P(b_−) ← false
        end if
    end if
end for
for all d = 1, −1 do
    for all levels l traversed by directions d do
        if level l contains a dummy node then
            let b_− be the topmost dummy node of level l
            let v_1, ..., v_r be the nodes to the top of b_−
        else
            b_− ← ∗
            let v_1, ..., v_r be all nodes of level l
        end if
        placeSequence(∗, b_−, d, v_1, ..., v_r)
        for i = 1 to r − 1 do
            m(v_i, v_{i+1}) ← y(v_{i+1}) − y(v_i)
        end for
        if b_− ≠ ∗ then
            m(v_r, b_−) ← y(b_−) − y(v_r)
        end if
        while b_− ≠ ∗ do
            let b_+ be the next dummy node to the bottom of b_−
            if b_+ = ∗ then
                let v_1, ..., v_r be the nodes to the bottom of b_−
                placeSequence(b_−, ∗, d, v_1, ..., v_r)
                for i = 1 to r − 1 do
                    m(v_i, v_{i+1}) ← y(v_{i+1}) − y(v_i)
                end for
                m(v_−, v_1) ← y(v_1) − y(b_−)
            else
                if D(b_−) = d then
                    let v_1, ..., v_r be the nodes between b_− and b_+
                    placeSequence(b_−, b_+, d, v_1, ..., v_r)
                    P(b_−) ← true
                end if
            end if
            b_− ← b_+
        end while
    end for
    adjustDirections(l, d, D, P)
end for
```

Algorithmus 7 adjustDirections(l, d, D, P)

```
v_- ← *
for all dummy nodes v_+ on level l + d traversed top down do
    if the neighbor w_+ of v_+ on level l is a dummy then
        if v_- ≠ * then
            p ← P(w_-)
            for all dummy nodes w between w_- and w_+ do
                p ← (p and P(w))
            end for
            if p then
                D(v_-) ← d
                for all dummy nodes v between v_- and v_+ do
                    D(v) ← d
                end for
            end if
        end if
        v_- ← v_+
        w_- ← w_+
    end if
end for
```

Sequenzen, die maximal von einem Dummyknoten begrenzt sind, werden nacheinander platziert. Äußere Sequenzen werden in beiden Durchläufen positioniert, wobei die berechneten Positionen aus dem ersten Durchlauf im zweiten Durchlauf als Abstände gelten, die nicht unterschritten werden dürfen. Innere Sequenzen werden nur einmal platziert.

In welcher Richtung dies geschieht, wird von Algorithmus 7, *adjustDirections*, berechnet. Für jede innere Sequenz der nächsten Ebene werden ihre Nachbarsequenzen der aktuellen Ebene betrachtet. Nachbarsequenzen sind wie folgt definiert und in Abbildung 3.4b dargestellt:

3.2.2 Definition. Sei $v_1, \ldots, v_n$ eine reguläre Sequenz auf der Ebene $l + d$ und $d \in \{-1, 1\}$ die Richtung, in der der Graph durchlaufen wird (die 1 bedeutet von links nach rechts und die -1 von rechts nach links). Sei v_- der nächste obere Bruder von v_1, der ein Dummyknoten ist und einen Dummyknoten w_- als Nachbarn auf der Ebene l hat, analog für v_+ und w_+. Alle regulären Sequenzen zwischen w_- und w_+ heißen dann Nachbarsequenzen von $v_1, \ldots, v_n$. Falls v_- oder v_+ existieren, heißt $v_1, \ldots, v_n$ eine innere Sequenz in Bezug auf die Ebene l, sonst äußere Sequenz.

Sind die Nachbarsequenzen bereits positioniert, wird die betrachtete innere Sequenz markiert. Wenn nun Algorithmus 6, *placeRegular*, bei der Berechnung der nächsten Ebene auf eine so markierte innere Sequenz stößt, wird diese platziert. Innere Sequenzen, die noch nicht markiert wurden, werden übersprungen und erst beim Durchlauf in die umgekehrte Richtung positioniert. Bleibt nun die Frage, ob auf diese Art tatsächlich alle Sequenzen positioniert werden.

Satz. Algorithmus 6, *placeRegular*, ruft Algorithmus 8, *placeSequence*, für jede reguläre Sequenz auf.

Beweis. siehe [BJL01] □

Zur Platzierung der Sequenzen $v_1, \ldots, v_n$ benutzen wir einen Divide & Conquer-Algorithmus. Dazu teilen wir die Folge in der Mitte t und platzieren beide Teilfolgen $v_1, \ldots, v_t$ und $v_{t+1}, \ldots, v_n$ rekursiv. Anschließend werden die Teilfolgen von Algorithmus 10, *combineSequences*, wieder kombiniert. Die Platzierung eines einzelnen Knotens nimmt Algorithmus 9, *placeSingle*, anhand des Medians der Nachbarknoten vor. Bei der Kombination muss berücksichtigt werden, dass sich Knoten überlappen können. Zur Lösung

Algorithmus 8 placeSequence$(b_-, b_+, d, v_1, \ldots, v_r)$

```
if r = 1 then
    placeSingle(b_-, b_+, d, v_1)
end if
if r > 1 then
    t ← ⌊r/2⌋
    placeSequence(b_-, b_+, d, v_1, ..., v_t)
    placeSequence(b_-, b_+, d, v_{t+1}, ..., v_r)
    combineSequences(b_-, b_+, d, v_1, ..., v_r))
end if
```

Algorithmus 9 placeSingle(b_-, b_+, d, v_1)

```
let w_1, ..., w_s be the neighbor nodes in δ̄(v_1, d) from above to bottom
if s ≠ 0 then
    y(v_1) ← y(w_{⌈s/2⌉})
    if b_- ≠ ∗ then
        y(v_1) ← max{y(v_1), y(b_-) + m(b_-, v_1)}
    end if
    if b_+ ≠ ∗ then
        y(v_1) ← min{y(v_1), y(b_+) − m(v_1, b_+)}
    end if
end if
```

des Problems schiebt man entweder v_t innerhalb der Ebene nach oben oder v_{t+1} nach unten. Obere und untere Brüder verschiebt man zur Einhaltung der Mindestabstände gegebenenfalls mit.

Es muss noch entschieden werden, ob es günstiger ist, v_t oder v_{t+1} zu verschieben. Dazu verwendet man die sogenannten Widerstände r_- und r_+. Widerstand r_- bezeichnet die Anzahl der Kanten, die sich verlängern, wenn man v_t nach oben verschiebt. Analog

Algorithmus 10 combineSequences$(b_-, b_+, d, v_1, \ldots, v_r)$

```
let R_- and R_+ be heaps
collectTopChanges(R_-)
collectTopChanges(R_+)
r_- ← 0
r_+ ← 0
while y(v_{t+1}) − y(v_t) < m(v_t, v_{t+1}) do
    if r_- < r_+ then
        if R_- = ∅ then
            y(v_t) ← y(v_{t+1}) − m(v_t, v_{t+1})
        else
            pop(c_-, y(v_t)) from R_-
            r_- ← r_- + c_-
            y(v_t) ← max{y(v_t), y(v_{t+1}) − m(v_t, v_{t+1}){
        end if
    else
        if R_+ = ∅ then
            y(v_{t+1}) ← y(v_{t+1}) + m(v_t, v_{t+1})
        else
            pop(c + −, y(v_{t+1})) from R_+
            r_+ ← r_+ + c_+
            y(v_{t+1}) = min{y(v_{t+1}), y(v_t) + m(v_t, v_{t+1}){
        end if
    end if
end while
for i = t − 1 down to 1 do
    y(v_i) ← min{y(v_i), y(v_t) − m(v_i, v_t)}
end for
for i = t + 2 to r do
    y(v_i) ← max{y(v_i), y(v_{t+1}) − m(v_{t+1}, v_i)}
end for
```

Algorithmus 11 collectTopChanges(R_-)

```
for i = 1 to t do
    c ← 0
    for all v ∈ δ̄(v_1, d) do
        if y(v) >= y(v_i) then
            c ← c + 1
        else
            c ← c − 1
            push(2, y(v) + m(v_i, v_t)) to R_−
        end if
    end for
    push(c, y(v_i) + m(v_i, v_t)) to R_−
end for
if b_− ≠ ∗ then
    push(∞, y(b_−)) + m(b_−, v_t) to R_−
end if
```

ist r_+ die Anzahl der Kanten, die sich verlängern, wenn v_{t+1} nach unten verschoben wird. Anhand der jeweiligen Werte von r_- und r_+ können die Teilfolgen dann wieder optimal kombiniert werden. Solange $r_- \leqslant r_+$ und $y(v_{t+1}) - y(v_t) < m(v_t, v_{t+1})$ gilt, schiebt man v_t nach oben. Wenn $r_- > r_+$ und $y(v_{t+1}) - y(v_t) < m(v_t, v_{t+1})$ gilt, wird v_{t+1} nach unten verschoben. Damit wird erreicht, dass sich möglichst wenige Kanten verlängern.

3.3 Integration von Ports und Knotengrößen

Der Algorithmus von Buchheim et al. muss nun für die Integration in KLay Layered angepasst werden. Man benötigt in der Regel bei der Modellierung realer Systeme verschiedene Knotengrößen und Platz zur Beschriftung der Knoten. KLay Layered unterstützt Graphen mit beliebigen Knotengrößen und Ports. Ausserdem wird in einem Rahmen um die Knoten Platz für Beschriftung vorgehalten.

Die Einführung verschiedener Knotengrössen und der Rahmen lässt sich relativ leicht realisieren, lediglich die Berechnung der Mindestabstände muss angepasst werden, wie in Abbildung 3.5 dargestellt.

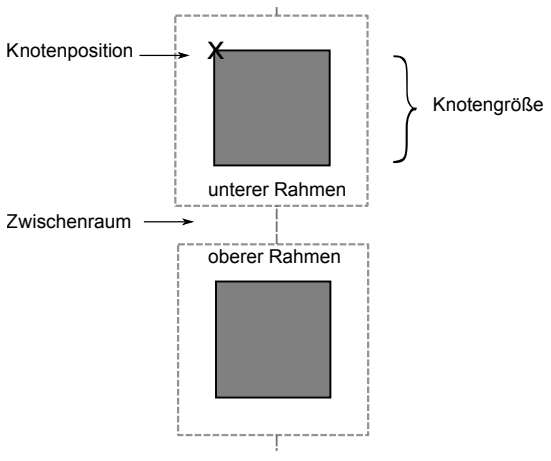
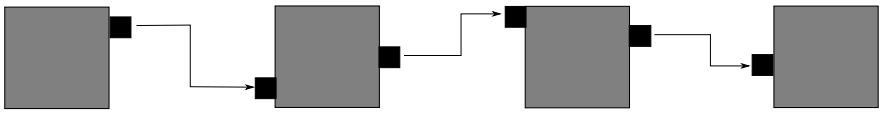
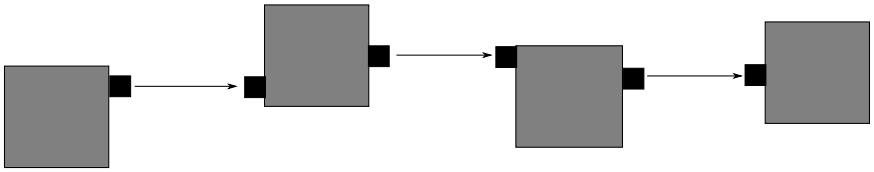


Abbildung 3.5. Der Mindestabstand der y-Koordinaten zweier Knoten zueinander berechnet sich, indem man Knotengröße, unteren Rahmen, Zwischenraum und oberen Rahmen addiert.



(a) unnötige Kantenknicke durch gleiche y-Koordinaten der Knoten.



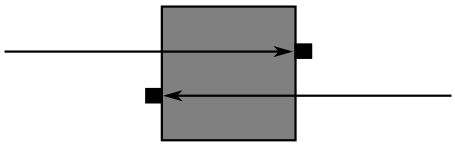
(b) Vermeidung von Kantenknicken durch gleiche y-Koordinaten der Ports.

Abbildung 3.6. Vermeidung von Kantenknicken durch Ausrichtung der Ports

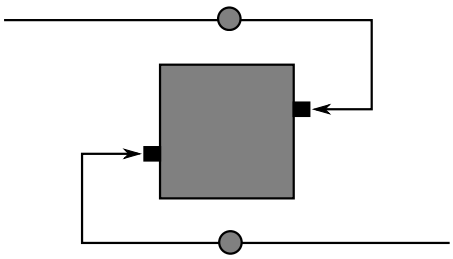
Außerdem ist zu beachten, dass die Abstände zwischen Kanten geringer sein können als zwischen regulären Knoten. Dafür werden unterschiedliche Werte für die Zwischenräume, das sogenannte *spacing*, berechnet.

Durch die Einführung von Ports reicht es nicht, Knoten an den Positionen ihrer Nachbarknoten auszurichten. Es müssen die verbundenen Ports der Knoten betrachtet werden. Gerade Kanten erhält man nur, wenn die Ports auf einer Höhe liegen, wie Abbildung 3.6 darstellt.

Eine weitere Besonderheit von KLay Layered sind die sogenannten *inverted ports*. Der Regelfall ist, dass sich eingehende Kanten auf der linken Seite eines Knotens befinden und ausgehende auf der rechten Seite. KLay Layered unterstützt aber auch den umgekehrten Fall. Wie in Abbildung 3.7 dargestellt, werden die Kanten um die Knoten dann



(a) *Inverted ports* ohne angepasstes Kantenrouting.



(b) *Inverted ports* mit angepasstem Kantenrouting durch zwei zusätzliche Dummyknoten.

Abbildung 3.7. *Inverted ports* mit unterschiedlichem Kantenrouting. Wenn die Kanten um die Knoten herumgeführt werden, sind zusätzliche Dummyknoten nötig und es entstehen Kanten innerhalb einer Ebene.

herumgeführt und hierfür Dummyknoten eingefügt. Dadurch entstehen Kanten, die Knoten derselben Ebene verbinden. In die Berechnung der Position eines Knotens anhand des Medians seiner Nachbarknoten dürfen solche Dummyknoten nicht mit einfließen.

Der Algorithmus von Buchheim et al. setzt voraus, dass es keine Überkreuzungen innerer Segmente gibt. Mit der Einführung von nördlichen und südlichen Ports in KLay Layered kann es aber vorkommen, dass sich innere Segmente überkreuzen, wie in Abbildung 3.8 dargestellt. Solche Überkreuzungen lassen sich erkennen, indem man die Reihenfolge der Segmente in zwei benachbarten Ebenen vergleicht. Ausführliche Erläuterungen hierzu sind in der Diplomarbeit von Schulze nachzulesen [Sch11].

In der aktuellen Implementierung wird dieses Problem direkt bei der Erstellung der linearen Segmente gelöst. Algorithmus 12, *computeLinearSegments*, zeigt das verwendete Verfahren.

KLay Layered unterstützt weitere Knotentypen. In dieser Implementierung ist die Unterscheidung zwischen regulären Knoten und übrigen Knoten ausreichend. Letztere können grundsätzlich wie Dummyknoten behandelt werden.

3.4 Implementierung

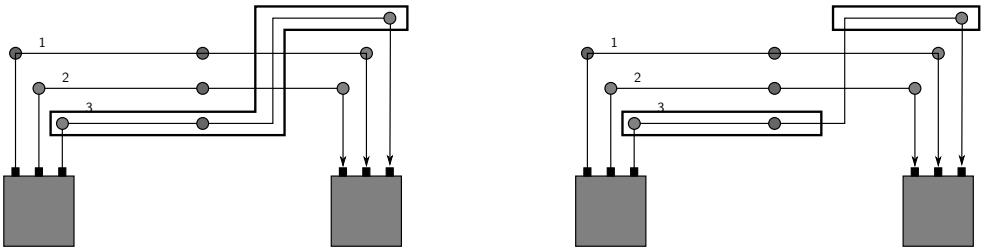
Integration in KLay Layered

KLay Layered ist in Java geschrieben. Durch die übersichtliche Struktur von KLay Layered kann der neue Knotenplatzierungsalgorithmus leicht eingebunden werden. Für jede der

Algorithmus 12 computeLinearSegments

```
let list_1 be an empty list
let list_2 be an empty list
let id ← 0 be a counter of the created segments
let s(v_i) be the id of the segment of v_i

for all level l do
    list2 ← list1
    for all v of level l do
        if v is not a dummy then
            s(v)
            id ← id + 1
        else
            let pred(v) be the neighbor of v in the previous layer
            if list_2 is not empty and pred(v) is dummy and l(pred(v)) < l(v) then
                let first(list_2) be the first element of list_2
                if first(list2) = s(pred(v)) then
                    s(v) ← s(pred(v))
                    remove first(list_2)
                else
                    remove first(list_2)
                    s(v) ← id
                    id ← id + 1
                end if
            else
                s(v) ← id
                id ← id + 1
            end if
        end if
        let succ(v) be the neighbor of vin the next layer
        if succ(v) is dummy and l(succ(v) > l(v)) then
            insert v in list_1
        end if
    end for
end for
```



(a) Konflikt: Nicht alle Knoten des linearen Segmentes können die gleiche y-Koordinate erhalten.

(b) Lösung: Aufteilung in zwei lineare Segmente. Alle Knoten eines linearen Segmentes erhalten wieder die gleiche y-Koordinate.

Abbildung 3.8. Überkreuzung innerer Segmente durch Nord-Süd-Ports. Durch die feste Reihenfolge der Ports benötigt mindestens eine lange Kante mehr als zwei Kantenknicke und kann deshalb nicht in einem einzigen linearen Segment zusammengefasst werden.

5 Phasen von KLay Layered ist jeweils ein Paket reserviert. Das Paket für die Phase 4 heißt folgerichtig:

```
de.cau.cs.kieler.klay.layered.p4nodes
```

Alle Knotenplatzierungsalgorithmen findet man an dieser Stelle. Die alternativen Algorithmen sind innerhalb des Pakets als Klassen repräsentiert. Der neue Knotenplatzierungsalgorithmus findet sich in der Klasse `BJLNodePlacer`.

Der `BJLNodePlacer` implementiert das Interface `ILayoutPhase`. Mit Hilfe dieses Interfaces lässt sich für jede der fünf Phasen von KLay Layered eine individuelle Konfiguration bestimmen. Dazu gehört auch die Spezifizierung der Intermediate Processors. Der neue *BJL Node Placer* wird über die Layoutoption `NODE_PLACER` zur Verfügung gestellt, um ihn für den Benutzer verwendbar zu machen.

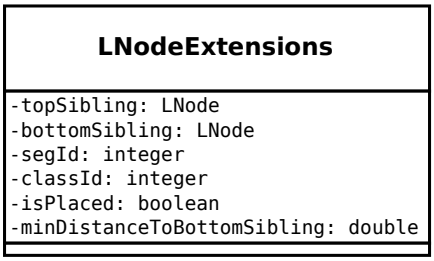
Datenstrukturen

Bei der Wahl der Datenstrukturen wird insbesondere auf schnelle Zugriffszeiten Wert gelegt, daher verwenden wir nach Möglichkeit Arrays.

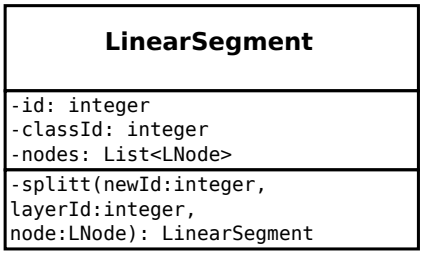
In einer Initialisierungsphase bekommen Knoten und Layer Indizes zugeordnet. Hierfür wird das frei belegbare Feld id verwendet, das allen Graphenelementen in KLay Layered zur Verfügung steht. Über diese Indizes lassen sich Knoten in Arrays einordnen. So kann auf jeden beliebigen Knoten mittels seiner id in konstanter Zeit zugegriffen werden.

Wir führen zudem eine innere Klasse `LNodeExtensions` ein. Der Algorithmus muss häufig auf Knoteninformationen zugreifen, die erst berechnet werden müssen. Dazu gehören etwa die Positionen direkter Brüder oder Abstände zu anderen Knoten. Diese Informationen werden, einmal berechnet, für jeden Knoten in einer eigenen `LNodeExtensions`-Instanz vorgehalten, wie im Klassendiagramm in Abbildung 3.9a dargestellt. Diese Erweiterungen werden für jeden Knoten anhand seiner id in dem Array `nodeExtensions` gespeichert.

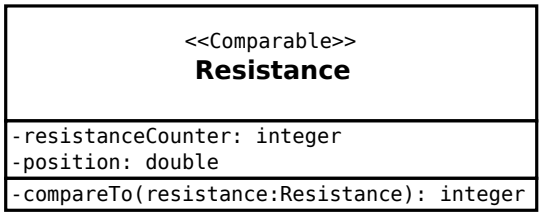
Wir benötigen Datenstrukturen zum Speichern der linearen Segmente und der Graphenklassen. Lineare Segmente werden im Programm als innere Klasse realisiert und enthalten eine Liste ihrer zugehörigen Knoten. Die Klasse wird in Abbildung 3.9b gezeigt.



(a) LNodeExtensions speichert erweiterte Informationen zu einem Knoten des aktuellen Graphen



(b) Ein lineares Segment besteht aus einem regulären Knoten oder allen Knoten einer langenKante



(c) Resistance speichert die Veränderung der Kantenlänge bei Variation der Knotenposition

Abbildung 3.9. Innere Klassen von `BJLNodePlacer`

Eine Graphenklasse besteht aus einer Liste von linearen Segmenten. Da generische Datentypen nicht in einem Array gespeichert werden können, müssen wir eine Liste verwenden. Man fügt die Klassen derart ein, dass ihre `id` dem Listenindex entspricht. Klassen sind nach dem Layer benannt, in welchem sich ihr oberster linker Knoten befindet. Deshalb ist die Nummerierung nicht zwangsläufig fortlaufend, und es ist zu beachten, dass es dadurch in der Liste Nulleinträge geben kann.

Dummyknoten erhalten ihre Position aus den Durchschnittswerten der berechneten oberen und unteren Layouts, welche daher zwischengespeichert werden müssen. Dies geschieht in den Arrays `nodePositionsTop` und `nodePositionsBottom`.

Im Zuge der Platzierung regulärer Knoten wird eine innere Klasse `Resistance` erstellt, wie in Abbildung 3.9c zu sehen. Die Klasse speichert die Widerstände gemeinsam mit den jeweiligen Knotenpositionen. Damit eine Sortierung anhand der Positionswerte möglich ist, implementiert sie das Interface `Comparable`.

Kapitel 4

Evaluation

In diesem Kapitel werden die Ergebnisse der Evaluation der vier in KLay Layered implementierten Node Placer dargestellt. Zunächst stellen wir die gemessenen Kriterien vor und erläutern die Auswahl der Testgraphen. Die Aufbereitung und Bewertung der Messergebnisse befindet sich im letzten Abschnitt.

4.1 Kriterien

Es ist keine leichte Aufgabe, objektive Kriterien zur Beurteilung von Graphenlayouts zu bestimmen. Jeder Mensch hat ein anderes Empfinden in Bezug auf Ästhetik oder Lesbarkeit einer Zeichnung. Dennoch gibt es messbare Kriterien, die eine objektive Qualitätsbeurteilung zulassen. Sugiyama et al. [STT81] und Di Battista [DETT99] haben eine Liste solcher messbaren Kriterien entwickelt. Die Bewertung des neu implementierten Algorithmus nach Buchheim et al. und der Vergleich mit den bereits implementierten Algorithmen wird anhand einiger ausgewählter Kriterien dieser Liste vorgenommen:

(**A**) Durchschnittliche Kantenlänge: Kürzere Kanten erhöhen die Nachvollziehbarkeit der Kantenverläufe.

(**B**) Anzahl der Kantenknicke: Weniger Kantenkicke erhöhen die Übersichtlichkeit der Kantenverläufe.

(**C**) Ausdehnung der Zeichnung: Eine kompakte Darstellung erhöht die Informationsmenge, die auf einem Bildschirm oder einem Ausdruck dargestellt werden kann. Die Knotenplatzierung beeinflusst allerdings nur die Höhe einer Zeichnung.

(**D**) Seitenverhältnis der Zeichnung: Ein gutes Layout nutzt die Ausmaße des Bildschirms oder eines Blattes Papier optimal aus.

Alle vier Kriterien können für jede Graphenzeichnung gemessen werden und ermöglichen eine objektive Bewertung der Zeichnung.

Als weiteres Qualitätskriterium betrachten wir die Performance des Algorithmus, also die Zeit für die Erstellung des Graphenlayouts.

4.2 Modelle

Die Auswahl der Testgraphen hat entscheidenden Einfluss auf die Qualität der Evaluation. Aussagekräftige Resultate erforden eine ausreichende Anzahl und Bandbreite verschiedener Graphen. Insgesamt werden 600 verschiedene Graphenlayouts evaluiert:

Tabelle 4.1. Modelle

	Zufallsgraphen	Ptolemy-Modelle
Diagramme	270	330
Anzahl der Knoten	10–50	10–450
Ausgehende Kanten pro Knoten	1,2	2,2
Varianz der Kanten	0,2	k.A.
Portpositionen	fixed	fixed

Tabelle 4.2. Ergebnisse mit Ptolemy-Modellen

	Simple	BJL	LS	BK
Fläche	705,57	763,26	725,06	877,45
Kantenlänge	214,56	199,78	181,47	202,53
Kantenknicke	92,73	58,56	57,53	40,59
Seitenverhältnis	3,05	2,44	2,64	2,18

Eine Auswahl an portbasierten Zufallsgraphen und eine Auswahl von Demonstrationsmodellen, die mit der an der UC Berkeley entwickelten Software Ptolemy[1] ausgeliefert werden. Die Größe der Zufallsgraphen variiert in der Knotenanzahl von 10 bis 50, die der Ptolemy-Modelle zwischen 10 und 450. Die wesentlichen Eigenschaften der Graphen können in Tabelle 4.1 nachvollzogen werden.

Zur Untersuchung der Performance wird ein Framework von Schulze und Spönemann verwendet. Das Framework erstellt Zufallsgraphen und misst die Zeit für deren Layoutberechnung. Die Einstellungen sind identisch mit denen der Zufallsgraphen zur Bewertung der ästhetischen Kriterien. Es werden je 5 verschiedene Graphen in 16 verschiedenen Größen erstellt. Der Test umfasst die Layoutberechnung von 80 Graphen, welche je fünfmal gemessen werden. Aus diesen fünf Testdurchläufen jedes Graphen wird die minimale Zeit übernommen, so dass Einflüsse von Caching und anderen laufenden Prozessen möglichst auszuschließen sind.

4.3 Ergebnisse

Wir untersuchen die vier verschiedenen Knotenplatzierungsalgorithmen, im Folgenden auch Simple, BJL, LS und BK genannt. Der *Simple Node Placer* gilt hier als Vergleichsalgorithmus, um anhand von Basiswerten die Relevanz des Node Placements generell zu veranschaulichen. Er geht aber nicht näher in die Analyse mit ein.

In Tabelle 4.2 sind die Ergebnisse für die Evaluation mit den Ptolemy-Modellen aufgeführt und in Tabelle 4.3 die Ergebnisse für die Zufallsgraphen. Für die Eigenschaften Knotenanzahl, Kantenlänge, Fläche und Seitenverhältnis sind die Durchschnittswerte der Messungen der vier Node Placer gegenübergestellt. Es fällt besonders auf, dass BK die wenigsten Kantenknicke produziert, während BJL und LS nahezu identische Werte liefern. Die Blockstruktur des BK mit der Einbeziehung regulärer Knoten erweist sich als

[1] `http://ptolemy.eecs.berkeley.edu/index.htm`

Tabelle 4.3. Ergebnisse mit Zufallsgraphen

	Simple	BJL	LS	BK
Fläche	234,34	271,27	256,22	371,72
Kantenlänge	160,87	147,89	132,09	154,02
Kantenknicke	89,06	58,15	57,37	42,48
Seitenverhältnis	1,48	1,19	1,19	1,04

effektiver bei der Minimierung von Kantenknicken. Demgegenüber führt die Aufteilung in lineare Segmente bei den anderen beiden Algorithmen zu einer höheren Kompaktheit.

Aus der Arbeit von Carstens ist bekannt, wie sich LS und BK zueinander verhalten, dazu gibt es auch hier keine überraschenden Ergebnisse [Car12]. Von primärem Interesse ist bei diesen Messungen die Einordnung des *BJL Node Placer* in diesem Kontext. Bezüglich der Kantenlängen und der Fläche liegen die Ergebnisse von BJL zwischen LS und BK. Sowohl bei den Ptolemy-Modellen wie bei den Zufallsgraphen kommt man zur gleichen Beurteilung der implementierten Algorithmen. In der Summe bringt der Einsatz von BJL in Bezug auf die gemessenen Kriterien also keinen Vorteil.

Die Grafiken 4.2, 4.3 und 4.4 zeigen die Beziehungen zwischen der Knotenanzahl und den verschiedenen Kriterien. Bei Kantenlängen und der Fläche liegen die Linien nah beieinander, nur in der Zahl der Kantenknicke unterscheiden sich die gemessenen Geradenverläufe deutlich, was bereits nach den ermittelten Durchschnittswerten zu erwarten war.

In den Diagrammen der Zufallsgraphen erkennt man eine in etwa lineare Abhängigkeit von Knotenanzahl und jeweiligem Kriterium. Die Ergebnisse der Ptolemy-Modelle weisen eine erhebliche Streuung der Messwerte auf, dennoch erkennt man auch hier in etwa eine Linearität. Alle vier Linien verlaufen nahezu parallel, auch wenn es erhebliche Schwankungen in den Messwerten gibt. Dies ist bemerkenswert deutlich in dem Diagramm 4.2 zu erkennen, das die Anzahl der Kantenknicke darstellt. Die Werte des BK liegen immer unter den Werten der anderen Placer, auch bei großen Schwankungen.

Beachtenswert sind die hohen Werte von BK für die Fläche bei sehr großen Graphen. Im Falle der Ptolemy-Modelle erkennt man in dem Diagramm 4.4a ab einer Anzahl von etwa 350 Knoten und bei den Zufallsgraphen in dem Diagramm 4.4b bei einer Knotenanzahl von 50, dass BK sehr viel größere Zeichnungen produziert als die anderen Node Placer. Das deutet darauf hin, dass es bei sehr großen Knotenzahlen durchaus sinnvoll sein kann, zugunsten der Kompaktheit mehr Kantenknicke zu tolerieren.

Wirklich überraschende Ergebnisse hat die Messung der Performance ergeben: Die Werte von BK und LS unterscheiden sich von früheren Messungen von Carstens erheblich, die Placer benötigen eine deutlich höhere Laufzeit. Bei 1000 Knoten lag die Ausführungszeit damals für LS bei etwa 50*ms* und bei BK sogar darunter [Car12]. Heute ergibt die gleiche Messung eine Zahl von 2754*ms* für LS und 575*ms* für BK. Die Ursachen hierfür sind zu untersuchen. Da die Algorithmen seit damals nicht verändert wurden, liegt das extreme Anwachsen der Laufzeit vermutlich nicht in den Knotenplatzierungsalgorithmen selbst begründet, sondern wurde durch Veränderungen an anderen Stellen von KLay Layered verursacht. Derzeit bleibt festzuhalten, dass der BJL erhebliche Vorteile in der Performance aufweist, er benötigt für 1000 Knoten im Test 73*ms*.

Im Ergebnis kann man feststellen, dass BJL insgesamt keine Qualitätsverbesserungen

Tabelle 4.4. Performancemessung der 4 Node Placer in *ms*

Knoten	Simple	BJL	LS	BK
10	0,03	0,79	0,59	1,08
100	0,05	3,02	6,03	7,26
398	0,53	11,88	114,34	83,52
1000	3,94	72,59	2754,15	575,07

des Layouts erreicht, hier gibt man sicher BK oder LS den Vorzug. Bei extrem großen Graphen ist es jedoch durchaus sinnvoll, BJL aufgrund der kürzeren Ausführungszeit einzusetzen.

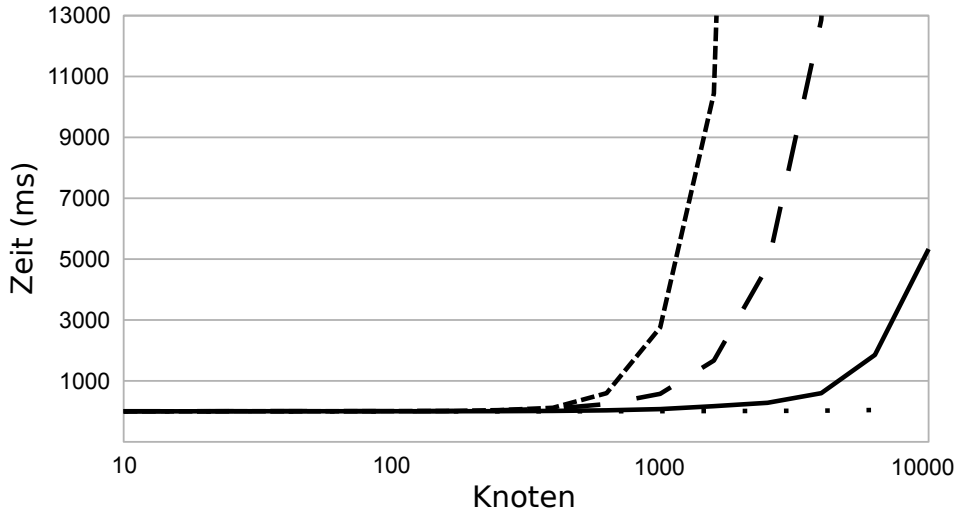
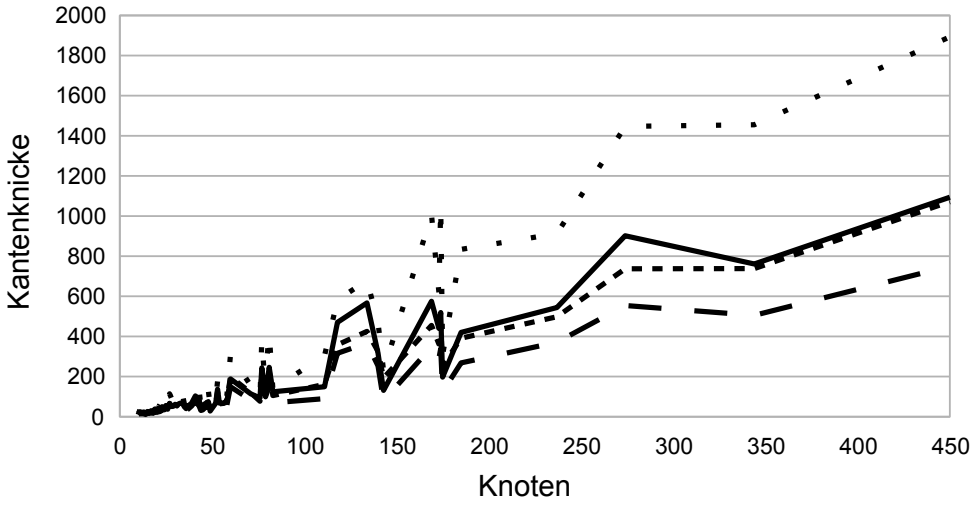
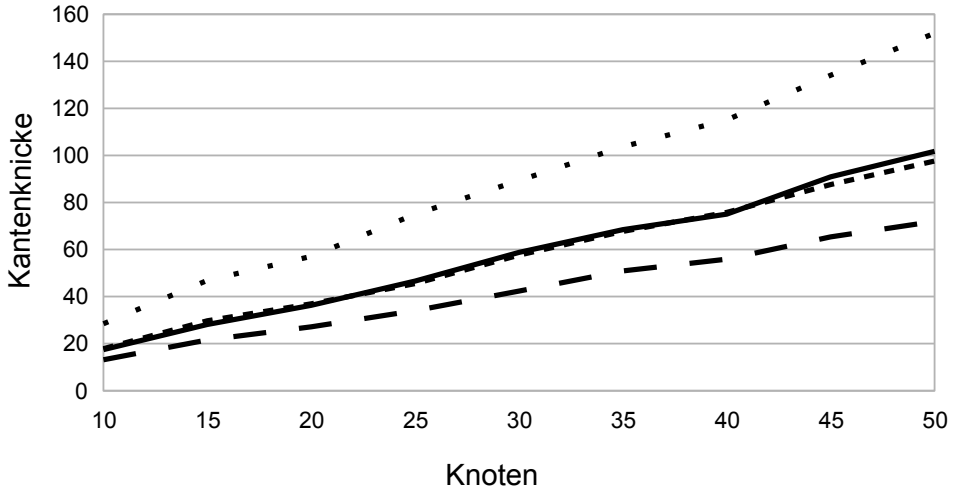


Abbildung 4.1. Performance der vier verschiedenen Node Placer. Die fein gestrichelte Linie repräsentiert LS, die grob gestrichelte Linie BK, die durchgehende Linie BJL und die gepunktete Linie Simple. Die x-Achse hat eine logarithmische Skalierung.

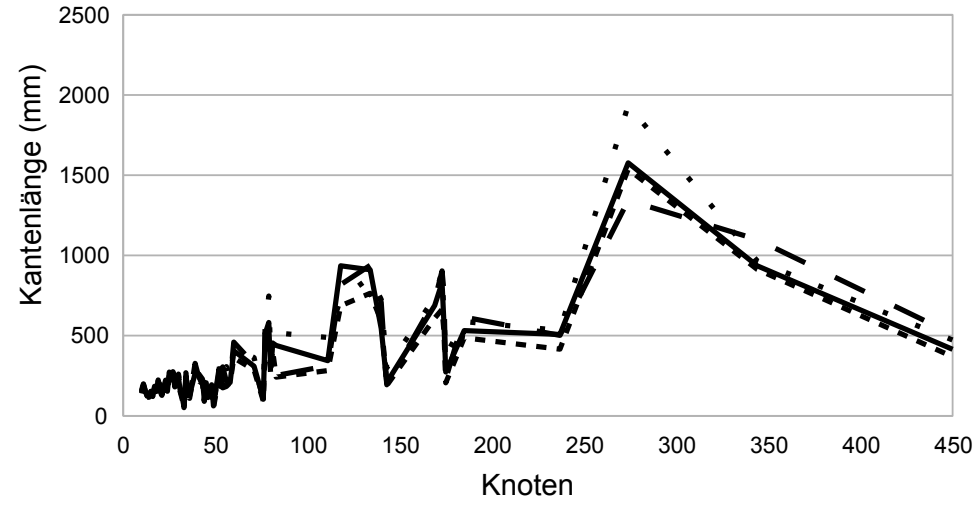


(a) Kantenknicke bei Ptolemy-Modellen

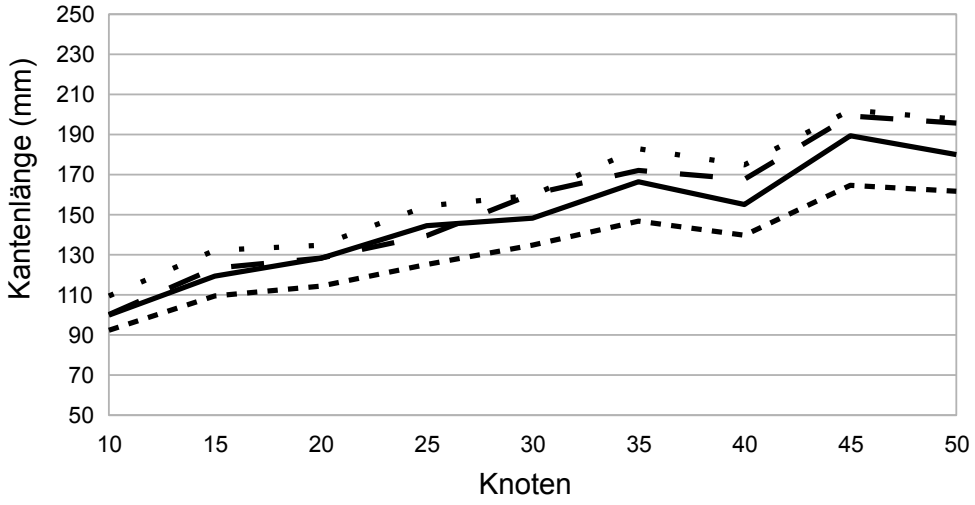


(b) Kantenknicke bei Zufallsgraphen

Abbildung 4.2. Gegenüberstellung der vier Node Placer in Bezug auf die Anzahl der Kantenknicke. Die fein gestrichelte Linie repräsentiert LS, die grob gestrichelte Linie BK, die durchgehende Linie BJL und die gepunktete Linie Simple

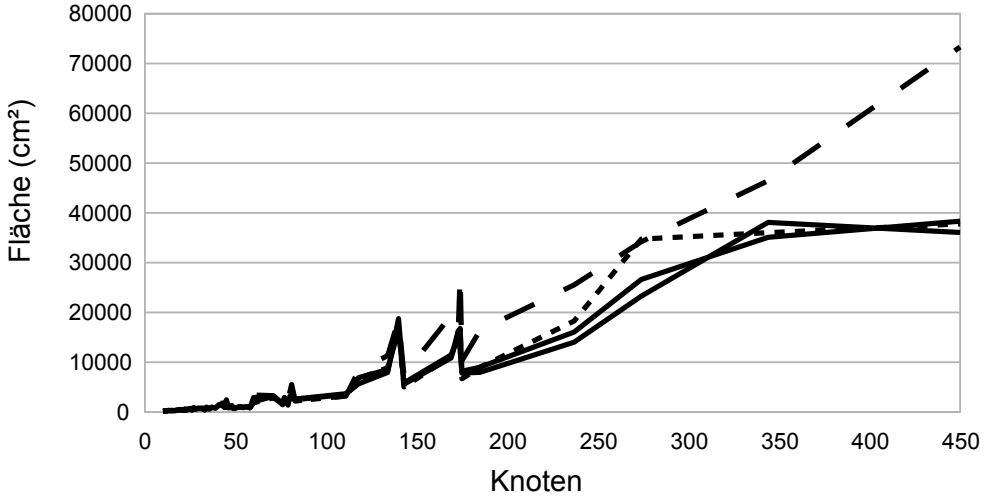


(a) Kantenlänge bei Ptolemy-Modellen

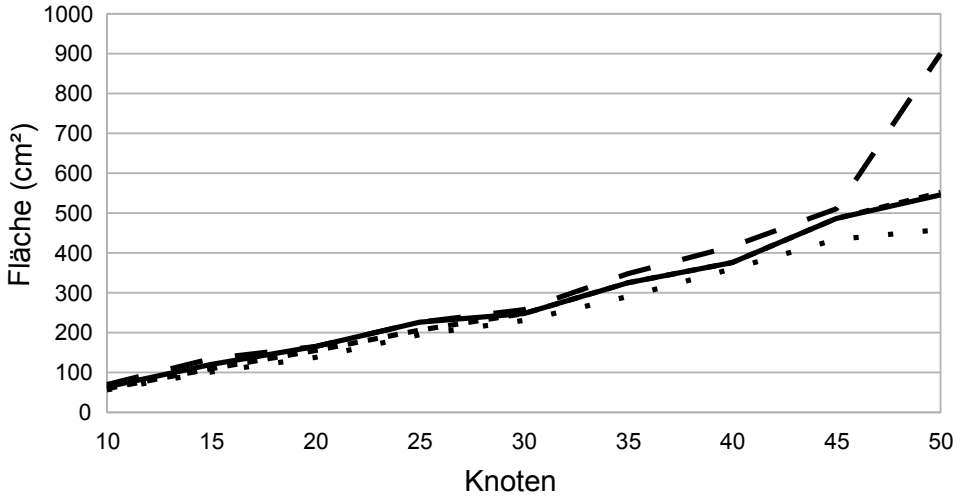


(b) Kantenlänge bei Zufallsgraphen

Abbildung 4.3. Gegenüberstellung der vier Node Placer in Bezug auf die Kantenlängen. Die fein gestrichelte Linie repräsentiert LS, die grob gestrichelte Linie BK, die durchgehende Linie BJL und die gepunktete Linie Simple



(a) Fläche bei Ptolemy-Modellen



(b) Fläche bei Zufallsgraphen

Abbildung 4.4. Gegenüberstellung der vier Node Placer in Bezug auf die Fläche. Die fein gestrichelte Linie repräsentiert LS, die grob gestrichelte Linie BK, die durchgehende Linie BJL und die gepunktete Linie Simple

Kapitel 5

Abschluss

5.1 Zusammenfassung

Ziel dieser Arbeit war es, den Knotenplatzierungsalgorithmus von Buchheim, Leipert und Jünger in KLay Layered zu implementieren. Zu diesem Zweck war es nötig, die speziellen Charakteristika von KLay Layered zu berücksichtigen. Der Algorithmus musste so modifiziert werden, dass er auch Ports und verschiedene Knotengrößen unterstützt. Bei der Berechnung von Mindestabständen zwischen zwei Knoten mussten diese Größen berücksichtigt werden.

Nord-Süd-Ports können zusätzliche Überkreuzungen innerer Segmente verursachen, die gesondert behandelt werden mussten. Inverted Ports erzeugen Kantenverbindungen innerhalb einer Ebene, weshalb sie bei der Ausrichtung eines Knotens anhand des Medians seiner Nachbarknoten nicht mit einbezogen werden durften.

Die allgemeine Fragestellung lautete, ob mit Hilfe dieses Verfahrens die Optimierung der Knotenplatzierungen zu realisieren sei. Insbesondere in Bezug auf die Minimierung von Kantenknicken sucht man für KLay Layered nach Möglichkeiten der Optimierung. Die Evaluation hat allerdings gezeigt, dass der neue *BJL Node Placer* hier keine Verbesserungen im Vergleich zu den bereits implementierten Lösungen bringen konnte.

5.2 Ausblick

Für die getesteten Graphen liefert der *BJL Node Placer* ähnliche Ergebnisse wie der *Linear Segments Node Placer*. Eine Möglichkeit, den *BJL Node Placer* zu optimieren, könnte das Einfügen einer Nachbereitungsphase innerhalb des Programms sein. Die Dummyknoten werden soweit wie möglich zusammen geschoben, um ein möglichst kompaktes Layout zu erhalten. Es könnten aber in einigen Fällen Knicke vermieden werden, wenn Kanten wieder nach außen verschoben würden.

Insgesamt zeigt die Evaluation jedoch die signifikanten Vorteile des *BK Node Placers* in Bezug auf die Anzahl der Kantenknicke. Es bietet sich daher an, diesen Platzierungsalgorithmus weiter zu entwickeln. Es existieren bereits Überlegungen zum *BK Node Placer*, um mehr Kanten pro Knoten gerade zu zeichnen. Bislang garantiert dieser immer nur eine gerade Kante pro Knoten.

Allerdings muss auch geklärt werden, wodurch die in den vorliegenden Messungen ermittelte nachteilige Entwicklung der Performance verursacht wird. Da die Werte ursprünglich erheblich besser waren, liegt das Problem vermutlich aber nicht in den Platzierungsalgorithmen selbst begründet.

Literaturverzeichnis

[BJL01] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. A Fast Layout Algorithm for k-Level Graphs. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 86–89. Springer Berlin / Heidelberg, 2001.

[BK02] Ulrik Brandes and Boris Köpf. Fast and Simple Horizontal Coordinate Assignment. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 33–36. Springer Berlin / Heidelberg, 2002.

[Car12] John Julian Carstens. Node and Label Placement in a Layered Layout Algorithm. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2012.

[CGJ+07] Markus Chimani, Carsten Gutwenger, Michael Jünger, Karsten Klein, Petra Mutzel, and Michael Schulz. The Open Graph Drawing Framework. Poster at the 15th International Symposium on Graph Drawing (GD07), 2007.

[DETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[ELS93] Peter Eades, Xuemin Lin, and William F Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[GKN02] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, February 2002.

[GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *Software Engineering*, 19(3):214–230, 1993.

[GN00] Emden R Gansner and Stephen C North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, 2000.

[Him97] Michael Himsolt. The graphlet system (system demonstration). In *Graph Drawing*, pages 233–240. Springer, 1997.

[Pur02] Helen C. Purchase. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages & Computing*, 13(5):501 – 516, 2002.

[San96] G. Sander. A fast heuristic for hierarchical Manhattan layout. In *Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 447–458. Springer Berlin / Heidelberg, 1996.

[San04] Georg Sander. Layout of Directed Hypergraphs with Orthogonal Hyperedges. In *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, volume 2912 of *LNCS*, pages 381–386. Springer, 2004.

[Sch11] Christoph Daniel Schulze. Optimizing Automatic Layout for Data Flow Diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2011.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.