Schöne Kurven

Ebenenbasiertes Kantenrouting mit Splines

Tibor Toepffer

Diplomarbeit
eingereicht im Jahr 2014

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme
Prof. Dr. Reinhard von Hanxleden

Betreut durch: Dipl.-Inf. 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,

Abstract

Automatisches Graphenlayout hat das Ziel, für einen gegeben Graphen ein Layout zu errechnen, das für den Benutzer verständlich und gut lesbar ist. Eine etablierte Herangehensweise hierfür ist das *ebenenbasierte Layout* nach Sugiyama et al. In dieser Arbeit wird eine kurvenbasierte Kantenführung für ein solches Layout entwickelt. Durch den Einsatz von Kurven werden wir ein angenehmes und harmonisches Layout erreichen. Eine Besonderheit ist, dass wir hierbei mit portbasierten Graphen arbeiten, wodurch der Freiheitsgrad beim Kantenrouting eingeschränkt wird. Ein weiterer Schwerpunkt liegt auf der Gestaltung und dem Routing von Selfloops. Auch hier werden wir gegebene Beschränkungen aufgrund von Ports berücksichtigen. Die Performance des entwickelten Routings ist vergleichbar mit bestehenden Routings.

Danksagung

Mein ganz besonderer Dank geht an meine Maus, ohne die ich niemals die Kraft und Motivation gefunden hätte, diese Arbeit zu erstellen. Meinem Betreuer Christoph Daniel Schulze und auch Ulf Rüegg danke ich für ihre stete und geduldige Unterstützung. Zuletzt soll auch Jan Foretník erwähnt werden, dessen NURBS-Onlinetool[1] ideal für das schnelle Ausprobieren neuer Ideen war.

[1] `http://geometrie.foretnik.net/files/NURBS-en.swf`

Inhaltsverzeichnis

Abbildungsverzeichnis

Definitionsverzeichnis

Abkürzungen

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KLay KIELER Layouters

KLay Layered KIELER Layouters Layered

OGDF Open Graph Drawing Framework

Graphviz Graph Visualisation

SCCharts Sequentially Constructive Charts

FAS Minimum Feedback Arc Set

MSAGL Microsoft Automatic Graph Layout

NURBS Non-Uniform Rational B-Spline

KLighD KIELER Lightweight Diagrams

GrAna Graph Analysis

Kapitel 1

Einleitung

Die visuelle Darstellung von Graphen hat sich aufgrund ihrer einfachen Verständlichkeit in einer Vielzahl von Anwendungsbereichen etabliert. Mit ihr ist es möglich, Strukturen und Zusammenhänge komplexer Daten auf einfache Art zu vermitteln. Gute Verständlichkeit bedingt jedoch ein gutes Design. Wird die Visualisierung manuell erstellt, so stehen einem meist guten Ergebnis zwei Nachteile entgegen. Neben der Schwierigkeit, ein einheitliches Designschema zu etablieren, ist der Prozess des manuellen Entwurfs sehr zeitaufwendig. Klauske and Dziobek berichten, dass zirka 30% der gesamten Modellierungszeit alleine auf die Platzierung der grafischen Elemente, das *Layout*, entfällt [KD10]. Mit dem automatischen Layout von Graphen (engl. *graph drawing*) wird das Ziel verfolgt, den Anwender von dieser Entwurfsarbeit zu entlasten.

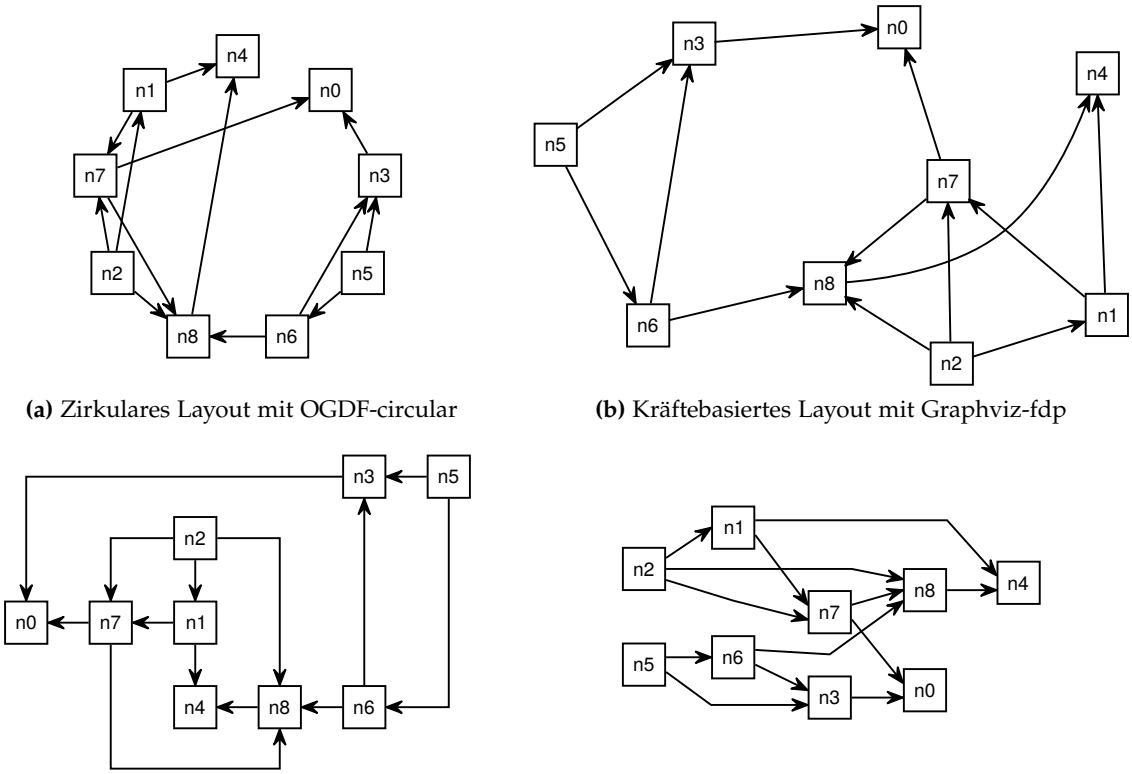
1.1. Graphen und ihr automatisches Layout

Automatisches Graphenlayout soll den Benutzer bei der zeitintensiven Entwurfsarbeit unterstützen und ihn idealerweise komplett davon befreien. Es gibt jedoch nicht das eine, optimale Layout. Abhängig von der Datenbasis und der Notwendigkeit, bestimmte Aspekte der Daten hervorzuheben, ändert sich die Wahl des besten Layoutansatzes. Auch die Darstellungstechnologie beeinflusst das Layout. In dieser Arbeit werden jedoch nur zweidimensionale Layouts betrachtet.

die hier betrachteten Graphen bestehen aus einer Menge von Objekten, *Knoten* genannt, und einer Menge von Verbindungen zwischen ihnen, *Kanten* genannt. Die Kanten können entweder gerichtet oder ungerichtet sein. Dieser Einteilung folgend spaltet sich das automatische Layout von Graphen in zwei Teilaspekte:

- *Das Layout der Knoten.* Die Aufgabe ist es, eine Platzierung der Knoten zu finden. Ziel kann beispielsweise eine gleichmäßige Verteilung auf der Darstellungsfläche oder eine Ordnung nach gegebenen Hierarchien sein.
- *Das Layout der Kanten.* Hier ist die Aufgabe, die Kanten zu *routen*. Dies bedeutet, für jede Kante einen Pfad vom Start- zum Zielknoten durch die bereits platzierten Knoten zu finden. Kriterien an ein gutes Routing sind beispielsweise die Gesamtlänge der Kanten und die Anzahl der Kantenkreuzungen.

Diese beiden Aspekte können nicht getrennt voneinander betrachtet werden, da sie sich gegenseitig beeinflussen. So hat beispielsweise die Platzierung der Knoten Einfluss auf



(a) Zirkulares Layout mit OGDF-circular

(b) Kräftebasiertes Layout mit Graphviz-fdp

(c) Planares Layout mit OGDF-planarization

(d) Ebenenbasiertes Layout mit KLay Layered

Abbildung 1.1. Vergleich verschiedener Layout Algorithmen eines identischen Graphen. Alle Graphen wurden mit KLay erstellt.

die Anzahl der Kantenkreuzungen.

Das automatische Layout von Graphen ist inzwischen gut erforscht. Einen Überblick über verschiedene Forschungsarbeiten bieten Di Battista et al. [DETT94]. Layoutalgorithmen können in folgende Kategorien eingeteilt werden:

- *Kreisbasierte Ansätze* ordnen die Knoten auf einer Kreisbahn an. Dieser einfache, geometrische Ansatz bietet sich beispielsweise für sehr dichte Graphen an, in denen nicht mehr die einzelne Kante, sondern die Gesamtmenge an Kanten eines Knotens von Interesse ist (Abbildung 1.1a).

- *Kräftebasierte Ansätze* gehen von einer randomisierten Ausgangsposition der Knoten aus. In dieses System werden nun Kräfte zwischen Knoten eingefügt: üblich ist beispielsweise eine abstoßende Kraft zwischen allen unverbundenen Knoten sowie eine anziehende Kraft zwischen mit Kanten verbundenen Knoten. Das Ziel ist es, einen Zustand möglichst geringer Energie für dieses System zu finden. Kräftebasierte Algorithmen führen zu einer Gruppenbildung von Komponenten, die viele Verbindungen

untereinander aufweisen (Abbildung 1.1b).

- ▻ *Planare Ansätze* haben das Ziel, die Kantenkreuzungen in der Darstellung zu minimieren. Ist ein Graph vollkommen ohne Kreuzungen darstellbar, so nennt man ihn auch *planar* (Abbildung 1.1c).
- ▻ *Ebenenbasierte Ansätze* berechnen eine Hierarchie der Knoten und gehen auf Kozo Sugiyama et al. zurück [STT81]. Die Hierarchie wird dabei so gewählt, dass der Graph eine „Flussrichtung“ erhält, in welcher alle Kanten verlaufen (Abbildung 1.1d).

Alle Beispiele in Abbildung 1.1 wurden im Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)-Projekt erstellt. Es wird am Lehrstuhl für Echtzeitsysteme der Instituts für Informatik an der Christian Albrechts Universität zu Kiel entwickelt. Für diese Arbeit ist vor allem das Unterprojekt KIELER Layouters (KLay) interessant, welches Teil vom KIELER ist. Einer der dort bereitgestellten Layoutalgorithmen ist KLay Layered, der eine erweiterte Implementierung des Ansatzes von Sugiyama et al. darstellt. In Unterkapitel 2.2 werden wir genauer auf KLay Layered eingehen. Abbildung 1.1d wurde mit diesem Algorithmus erzeugt. Er wird auch in dieser Arbeit genutzt werden.

1.2. Kanten

Eine Möglichkeit, die Qualität eines Layouts objektiv zu beurteilen, ist die Definition von *Metriken*. In Unterkapitel 3.1 werden wir sehen, dass sich viele der üblichen Metriken auf die Kanten beziehen. Es ist beispielsweise gängig, die Anzahl der Kantenkreuzungen (weniger ist besser) oder die Gesamtlänge aller Kanten (kürzer ist besser) zu messen. Das Kantenrouting hat also eine hohe Bedeutung bei der Erstellung eines guten Graphenlayouts.

In Abbildung 1.1 ist zu erkennen, dass es verschiedene Ansätze gibt, Kanten zu routen. Abbildung 1.1a zeigt ein Routing mit einfachen geraden Strecken. Dies ist auch in Abbildung 1.1b der Fall, jedoch mit einer Ausnahme: die Kante von $n8$ nach $n4$ wird durch eine *Kurve* dargestellt. Das planare Layout in Abbildung 1.1c weist ein *orthogonales* Kantenrouting auf, Abbildung 1.1d schließlich ein Routing mit *Polylinien*. Neben der Bestimmung des Kantenverlaufs, ist auch eine Variation der Kantenform und -farbe denkbar. Derlei Möglichkeiten werden in dieser Arbeit nicht betrachtet. Stattdessen werden wir näher auf eine Kantendarstellung durch Kurven eingehen.

1.2.1. Kurven

Wie bereits erwähnt ist in Abbildung 1.1b eine Kurve zu sehen. In Unterkapitel 3.1 werden wir auf die konkreten Vor- und Nachteile der Wahl von Kurven zur Kantendarstellung näher eingehen. In erster Linie ist diese Wahl jedoch eine Designentscheidung. Eine Kurve unterscheidet sich von einem Funktionsgraphen dahingehend, dass der Funktionsparameter nicht auf einer der Koordinatenachsen abgebildet wird. In Abbildung 1.2 ist

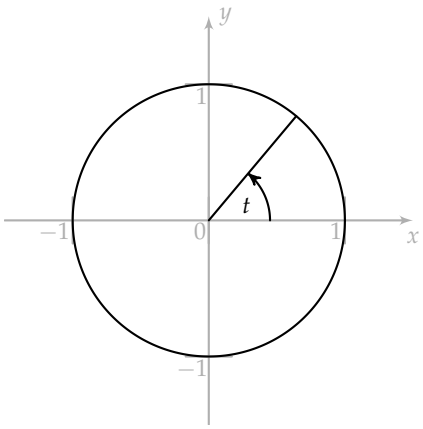


Abbildung 1.2. Der Einheitskreis als parametrischen Funktion $f(t) = \begin{pmatrix} cos(t) \\ sin(t) \end{pmatrix}$ mit $t \in [0, 2\pi[$.

beispielsweise der Einheitskreis als Ergebnis einer *parametrischen Funktion* dargestellt. Hier ist der Funktionsparameter t der Winkel im Koordinatenursprung. Kurven haben den Vorteil, dass einem x-Wert nun mehrere y-Werte zugeordnet werden können. So können mit Kurven beispielsweise auch senkrechte Kanten beschrieben werden.

Eine Definition der Kanten durch Kurven in Parameterdarstellung ist also denkbar. In Unterkapitel 2.3 werden wir sehen, dass dieser Ansatz verschiedene Nachteile hat. Im Bereich der Computergrafik hat sich eine andere Technik durchgesetzt: die Beschreibung von Kurven durch *Splines*. Splines basieren ebenfalls auf Funktionen und sind stückweise aus *Polynomen* zusammengesetzt. Ihr Vorteil liegt darin, dass sie nicht durch die Angabe einer komplexen Funktion definiert werden, sondern durch die Festlegung sogenannter *Stützstellen*, die in Abbildung 1.3 dargestellt sind.

Um einen glatten Verlauf zu gewährleisten, ist eine gewisse *Stetigkeit* an den Übergangspunkten der Teil-Polynome notwendig. Abbildung 1.3 zeigt einen weit verbreiteten Spline, einen *kubischen Bezier-Spline*, bestehend aus zwei stetig ineinander übergehenden *Bezierkurven*. Jede Kurve hat vier Kontrollpunkte, wobei die äußeren Kontrollpunkte den Start- beziehungsweise Zielpunkt bilden. Die zwei inneren Kontrollpunkte bestimmen den Kurvenverlauf. Der Verlauf von Bezierkurven lässt sich sehr leicht in beliebiger Genauigkeit berechnen, worauf wir in Unterkapitel 2.3 näher eingehen werden.

1.2.2. Kantenrouting in ebenbasierten Graphen

Layoutalgorithmen, die auf den ebenenbasierten Ansatz von Sugiyama et al. aufsetzen, ordnen die Knoten auf parallelen *Ebenen* (engl. *layer*) an. Ziel ist es, dass alle Kanten stets von einem Layer in den nächsthöheren Layer laufen. Um dies zu erreichen, werden im Verlauf des Algorithmus Kanten umgedreht und *Dummy-Knoten* eingefügt. Abbildung 1.4 zeigt dieses Vorgehen exemplarisch. Der Sugiyama-Algorithmus ist in mehrere Phasen eingeteilt, wobei das Kantenrouting die letzte Phase darstellt. Zu Beginn des

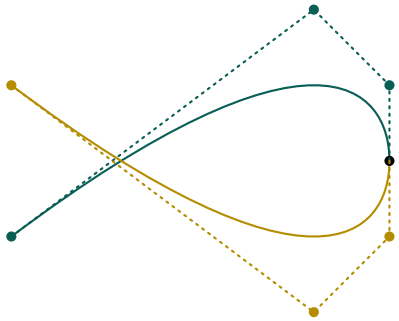
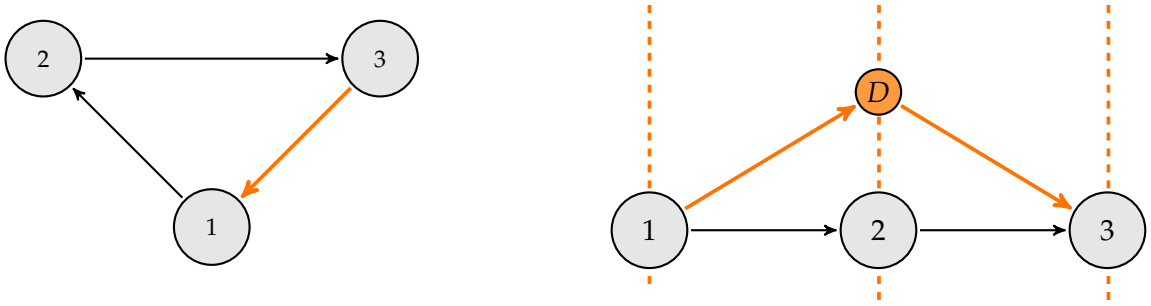


Abbildung 1.3. Ein kubischer Bezier-Spline, bestehend aus zwei kubischen Bezierkurven.



(a) Kante 3 → 1 ist umzudrehen

(b) Die Knoten sind verteilt, ein Dummy eingefügt

Abbildung 1.4. Der Sugiyama-Algorithmus verteilt die Knoten auf Layer. Layerübergreifende Kanten werden durch Dummy Knoten (D) unterbrochen. Kanten werden umgedreht ($3 \rightarrow 1$), so dass alle Kanten zum jeweils nächsthöheren Layer laufen.

Kantenroutings ist der Grobverlauf der Kanten bereits definiert. Alle Kanten liegen genau zwischen zwei benachbarten Layern und weisen in dieselbe Richtung. Diese Ausgangsbedingungen vereinfachen das Layout der Kanten erheblich. Eine genaue Betrachtung des Gesamtalgorithmus wird in Unterkapitel 2.2 erfolgen.

1.3. Ziele dieser Arbeit

Ziel dieser Arbeit ist die Entwicklung eines Kantenroutings auf Basis von Spline-Kurven. Das Routing soll in KLay Layered integriert werden.

Ein besonderes Augenmerk liegt hierbei auf Selfloops, also Kanten, die direkt an ihren Startknoten zurücklaufen. Diese sind zur Zeit nur unbefriedigend in KLay Layered implementiert. Wenn möglich soll die Implementierung der Selfloops später auch für andere Kantenroutings, wie beispielsweise das Polyline-Routing, eingesetzt werden.

Weiterhin soll eine sinnvolle Platzierung der Beschriftungen an allen Splinekanten gefunden werden. Beschriftungen müssen beim Routen der Kanten berücksichtigt werden,

damit sie frei von Kantenüberschneidungen bleiben.

Sinnvolle Kriterien für ein ästhetisches Kantenrouting müssen gefunden, und das entwickelte Routing bezüglich dieser Kriterien bewertet werden. Neben ästhetischen Kriterien soll auch die Performance untersucht und bewertet werden.

Ein wichtiger Verwendungszweck des zu entwickelnden Kantenroutings sind die ebenfalls am Lehrstuhl für Echtzeitsysteme entwickelten Sequentially Constructive Charts (SCCharts). Hierbei handelt es sich um eine visuelle, synchrone Programmiersprache, die zur Spezifizierung sicherheitskritischer, reaktiver Systeme entwickelt wird. Zur Zeit werden die Kanten in diesen SCCharts mit dem dot-Algorithmus von Graph Visualisation (Graphviz) dargestellt. Dieser soll ersetzt werden.

Es ist nicht das Ziel dieser Arbeit einen neuen Portplazierungsalgorithmus zu entwerfen. Wir werden an verschiedenen Stellen zu dem Ergebnis kommen, das die Anpassung dieses Algorithmus auch für die hier gestellte Aufgabe von Vorteil wäre. Es würde jedoch den zur Verfügung stehenden Rahmen sprengen auch diese Aufgabe anzunehmen.

1.4. Verwandte Arbeiten

Splines zur Darstellung von Kanten einzusetzen wurde schon vor dieser Arbeit realisiert. In diesem Abschnitt wird ein Überblick über einige dieser Arbeiten gegeben.

Graphviz[1] ist eine Open Source-Graphenvisualisierungssoftware. Ihre Entwicklung wird maßgeblich in den *AT&T Bell Laboratories* vorangetrieben. Der enthaltene *dot*-Algorithmus ist ein ebenenbasierter Graphenlayoutalgorithmus [GKNV93, GKN02], der Splines zur Kantendarstellung einsetzen kann. Die Graphviz-Bibliothek ist in KIELER eingebunden, so dass auch der dot-Algorithmus in dieser Umgebung getestet werden kann. Zudem steht mit *GVEdit* eine grafische Oberfläche zur Verfügung, welche jedoch zum im KIELER eingebunden Algorithmus abweichende Resultate liefert.

Das Routing der Kanten im dot-Algorithmus unterscheidet sich von dem in dieser Arbeit vorgestellten. Es ist zweigeteilt: Zunächst wird der für eine Kante nutzbare Raum bestimmt, um dann eine Bezierkurve durch den Raum zu legen. Der nutzbare Raum wird durch eine Reihe von Rechtecken bestimmt. Verlässt die Kurve an einer Stelle den definierten Raum, so wird zunächst versucht die Kurve durch Variation der Kontrollpunkte zu korrigieren. Ist auch dies nicht möglich, so wird die Kante an der Stelle des Defektes zweigeteilt und durch zwei Splines dargestellt.

Der Algorithmus liefert gute Ergebnisse. Durch die Nutzung des „freien Raumes“ werden unnötige Knicke in den Kanten vermieden. Die resultierenden Kanten weisen damit eine Krümmung auf, die das notwendige Maß nicht überschreitet.

Der Nachteil besteht im zusätzlichen Rechenaufwand. Bei geschickter Einbindung in den des Sugiyama-Algorithmus muss der Raum nicht extra berechnet werden. Der Algorithmus liefert unbefriedigende Ergebnisse bei Selfloops, oder wenn der Kantenpfad

[1] http://www.graphviz.org/

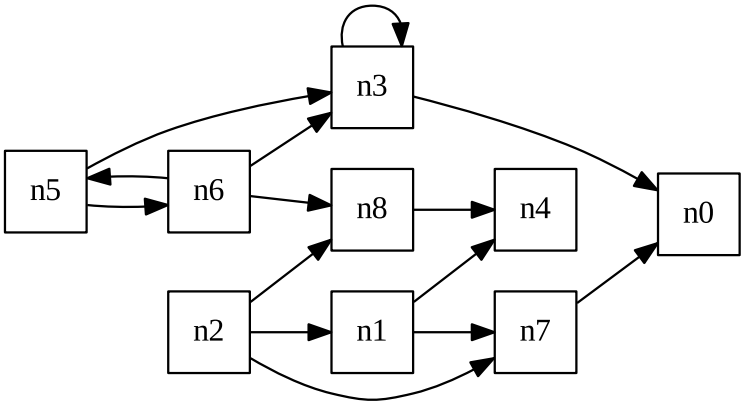


Abbildung 1.5. Beispielgraph. Layout durch den dot-Algorithmus von Graphviz, dargestellt durch GVEdit.

durch Randbedingungen wie *Ports* eingeschränkt wird. Gleiches gilt für Kanten, die zwei Knoten im selben Layer verbinden [DGKN97].

Selfloops werden stiefmütterlich behandelt: Liegen mehrere Loops an einem Knoten, so werden sie einfach an einer Seite des Knotens gestapelt. Beim Layout in GVEdit werden zusätzliche Selfloops am selben Knoten sogar ignoriert.

Dobkin et al. stellten eine Verbesserung des dot-Algorithmus vor [DGKN97]. Sie verallgemeinerten den dot-Algorithmus, so dass er nun auch für andere Anwendungssituationen als nur ebenenbasierte Graphen nutzbar ist. Die oben beschriebenen Probleme mit Randbedingungen an den Pfadverlauf werden somit vermieden, da der Algorithmus nicht auf die strenge Einhaltung des ebenbasierten Layouts angewiesen ist. Der „freie Raum“ wird nun nicht mehr durch eine Reihe von Rechtecken, sondern durch *nicht überschlagende Polygone* (engl. *simple polygon*) definiert. Sie enthalten Löcher, ebenfalls nicht überschlagende Polygone, welche die Knoten darstellen.

Auch hier sind die Ergebnisse sehr zufriedenstellend. Die resultierenden Splines stellen eine gute Mischung aus den Anforderungen „Nähe zum kürzesten Pfad“ und „geringe Krümmung“ dar. Die Autoren haben jedoch eine entscheidende Einschränkung vorgenommen: die Kanten werden ohne Berücksichtigung anderer Kanten berechnet. Somit stellt dieser Ansatz zwar eine interessante Grundlage, aber in seiner vorgestellten Form eine unbefriedigende Lösung für unser Kantenroutingproblem dar, da Kanten sich überlappen können.

Georg Sander entwickelte das VCG-Tool und verwendet ebenfalls Splines zum Kantenrouting [San94]. Gemein mit der Arbeit von Gansner ist dieser, dass das Kantenrouting umgebungsabhängig ist. Es wird also während des Routings auf mögliche Kollisionen mit anderen Objekten geprüft und der Kantenverlauf gegebenenfalls angepasst. Sanders Ansatz ist zur Erreichung einer guten Performance einfacher gefasst. Der Hauptaugenmerk der Arbeit lag auf Graphen mit Knoten stark unterschiedlicher Größe.

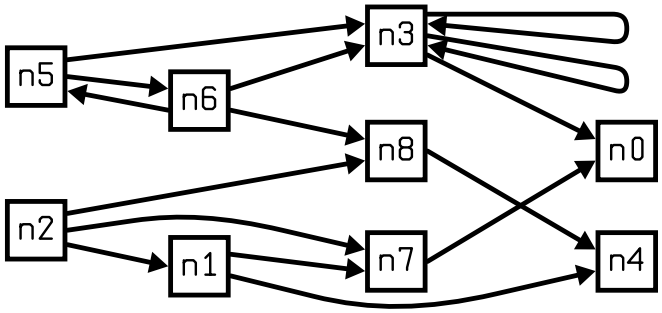


Abbildung 1.6. Beispielgraph. Layout durch das VCG-Tool, Darstellung durch AISee mit aktivierter Spline-Funktion.

Der Pfad, der sich aus den Dummy-Knoten ergibt, wird zunächst durch Polylinien nachvollzogen. Dann erfolgt die Kollisionskontrolle. Durchschneidet eine Kante auf dem Weg von einer Ebene zur nächsten einen Knoten, so wird zunächst ein Knickpunkt eingefügt, so dass sich ein gerades Segment bis auf Höhe des kollidierten Knotens ergibt. Ergibt sich nun eine neue Kreuzung mit einer anderen Kante, so wird auch diese Kante geknickt.

Die Firma AbsInt hat auf dem VCG Tool basierend die Visualisierungssoftware *AISee*[2] entwickelt. Wie man in Abbildung 1.6 sehen kann, werden Selfloops auch von diesem Algorithmus nur unbefriedigend behandelt.

Lev Nachmanson et al. entwickelten die Software GLEE [NRL08]. Sie ist inzwischen in das Microsoft Automatic Graph Layout (MSAGL)-Projekt aufgegangen. Das Kantenrouting von GLEE baut auf die im Sugiyama-Algorithmus gefundenen Pfade auf. Es wird eine Polylinie entlang des Pfades gelegt und auf Kollisionen mit anderen Objekten überprüft. Treten solche Kollisionen auf, so wird die Linie entsprechend angepasst. Danach wird die Polylinie nach einer Heuristik begradigt und ein Spline entlang ihres Verlaufs gelegt. Selfloops behandeln Nachmanson et al. in ihrer Arbeit nicht.

Da GLEE Teil des MSAGL ist, gibt es im Gegensatz zu den bisher vorgestellten Programmen keine Grammatik, um Graphen textuell zu beschreiben. Stattdessen bietet die Bibliothek eine C♯-Schnittstelle an.

Bereits 1988 stellten Gansner et al. mit *DAG* ein Programm vor, das einen sehr ähnlichen Ansatz wie diese Arbeit verfolgte [GNV88]. Die Pfade der Kanten werden zunächst durch die Knotenplatzierung festgelegt. Anschließend wird ein *B-Spline* entlang dieses Pfades gelegt. Die Ergebnisse sind jedoch, gerade bei größeren Graphen, recht unbefriedigend. Die Kanten weisen teilweise unnötig starke Krümmungen auf. Die Positionierung von Beschriftungen ist nicht ausgereift. Selfloops scheinen auf dieselbe Art wie im dot-Algorithmus berechnet zu werden.

[2] www.absint.com/aisee/

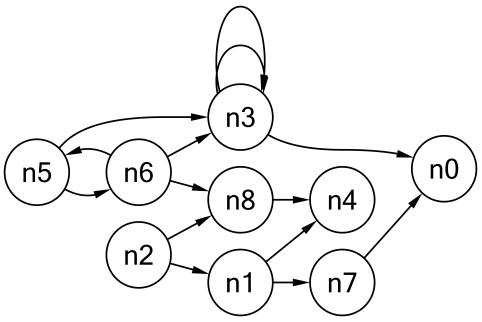


Abbildung 1.7. Beispielgraph. Layout und Darstellung durch GLEE.

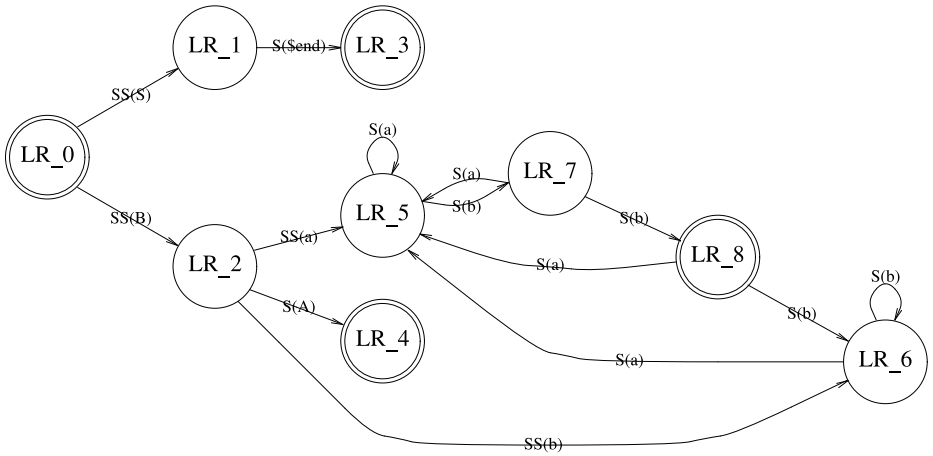


Abbildung 1.8. Ein Beispiellayout, erstellt von DAG. Entnommen aus [GNV88].

1.5. Gliederung

Nachdem in der Einleitung ein Überblick über das Umfeld der vorliegenden Arbeit gegeben wurde, kommen wir nun zur Ausarbeitung der Themen.

Im folgenden Kapitel 2 wird das notwendige Basiswissen für die restlichen Kapitel vermittelt. Zunächst wird die in dieser Arbeit verwendete Notation eingeführt. Neben einer Vorstellung und Diskussion verschiedener Arten von Splines wird auch das KIELER-Framework und speziell der KLay Layered-Algorithmus sowie der Ansatz von Sugiyama, auf dem er beruht, eingehend betrachtet.

In Kapitel 3 wird die Bewertung eines Graphenlayouts auf Grundlage verschiedener Ästhetikkriterien vorgestellt. Sinnvolle Kriterien für ein gutes Layout des zu entwerfenden Routings werden herausgearbeitet. Es wird aufgezeigt welche Spline-Technologie für die gestellte Aufgabe warum am besten geeignet ist. Auf Basis dieser Technologie werden dann die theoretischen Lösungsansätze für die verschiedenen Teilaufgaben entwickelt.

In Kapitel 4 werden diese theoretischen Lösungsansätze in Java umgesetzt. Hier liegt das spezielle Augenmerk auf programmiertechnischen Problemstellungen.

In Kapitel 5 wird die ästhetische Qualität des Kantenroutings betrachtet. Es wird ebenfalls ein Vergleich mit anderen Splines verwendenden Layouts vorgenommen. Weiterhin wird die Gesamtlaufzeit des kompletten KLay Layered-Algorithmus betrachtet.

Kapitel 6 zeigt dann abschließend nicht oder nur unbefriedigend gelöste Teilaufgaben auf gibt einen Ausblick auf zukünftige Verbesserungsansätze.

Kapitel 2

Grundlagen und Notationen

In diesem Kapitel sollen die wichtigsten Grundlagen vermittelt werden. Einige der Konzepte werden erst in den folgenden Kapiteln vorgestellt. Trotzdem wird die Notation bereits hier aufgezeigt, mit dem Ziel, das Nachschlagen einer Definition zu vereinfachen.

2.1. Mathematische Grundlagen

2.1.1. Graphen

1. Definition (Vektor, Gerade)**.** Gegeben seien zwei Punkte $v_0, v_1 \in \mathbb{R}^2$. Mit $\overrightarrow{v_0 v_1}$ bezeichnen wir dann den *Vektor* von v_0 nach v_1, mit $\overline{v_0 v_1}$ die *Gerade* durch die beiden Punkte.

2. Definition (Knoten, Kanten, (gerichteter) Graph)**.** Ein *(gerichteter) Graph* ist ein Tupel $G = (V, E)$ bestehend aus einer Menge von *Knoten* V (engl. *vertices*) und einer Menge von *Kanten* $E \subseteq V \times V$ (engl. *edges*). Ein gerichteter Graph wird auch *Digraph* genannt.

3. Definition (Pfad, Zykel)**.** Sei $G = (V, E)$ ein gerichteter Graph. Eine Folge von Knoten $(v_0, \ldots, v_n)$ mit $(v_i, v_{i+1}) \in E$ für alle $i \in \{0, \ldots, n-1\}$, heißt *Pfad*. Ein Pfad $(v_0, \ldots, v_n)$ mit $v_0 = v_n$ heißt *Zykel*. Ein Graph heißt *zyklenfrei* oder *azyklisch*, wenn er keine Zykel enthält.

4. Definition (eingehende Kanten, Startknoten, Zielknoten)**.** Sei $G = (V, E)$ ein gerichteter Graph. Die Menge der *eingehenden Kanten* eines Knotens $v \in V$ bezeichnen wir mit $E_i(v) := \{(x, y) \in E : y = v\}$, die Menge der ausgehenden Kanten mit $E_o(v) := \{(x, y) \in E : x = v\}$. Die Gesamtheit der mit v verbundenen Kanten wird mit $E(v) := E_i(v) \vee E_o(v)$ bezeichnet. Der *Startknoten* (engl. *source*) einer Kante $e = (u, v) \in E$ wird mit $v_s(e) := u$, der *Zielknoten* (engl. *target*) mit $v_t(e) := v$ bezeichnet.

5. Definition (Port)**.** Ein *portbasierter, gerichteter Graphen* $G = (V, E, P, f)$ besteht aus einer Menge von Knoten V, einer Menge von *Ports* P, einer Menge von Kanten $E \subseteq P \times P$ und einer totalen Funktion $f : P \to V$, die jeden Port genau einem Knoten zuweist. Die Mengen der Ports eines Knotens $v \in V$ wird mit $P(v) := f^{-1}(v)$ bezeichnet. Um in unserer bisherigen Notation zu bleiben, schreiben wir auch $v(p)$ für den Knoten an dem der Port p liegt. Der Startport einer Kante $e \in E$ wird mit $p_s(e) \in P(v_s(e))$, der Zielport mit $p_t(e) \in P(v_t(e))$ benannt. Die mit einem Port p verbundenen Kanten werden mit $E(p)$

bezeichnet, die mit ihm verbundenen ein- beziehungsweise ausgehenden Kanten mit $E_i(p)$ und $E_o(p)$.

2.1.2. Ebenenbasierte Graphen

6. Definition ((strenge) Ebenzuweisung)**.** Sei $G = (V, E)$ ein gerichteter, azyklischer Graph. Die totale Funktion $l : V \to \mathbb{N}$ heißt *Ebenenzuweisung* (engl. *layering*), wenn $l(v_0) < l(v_1)$ für alle $(v_0, v_1) \in E$ gilt. Einen Graphen $G = (V, E)$ mit einer Ebenenzuweisung l nennen wir *ebenenbasiert* und schreiben ihn als $G = (V, E, l)$. Entsprechend schreiben wir einen port- und ebenbasierten Graphen als $G = (V, E, P, f, l)$. Eine Ebenenzuweisung l heißt *streng*, wenn $l(v_0) + 1 = l(v_1)$ für alle $(v_0, v_1) \in E$ gilt. Für einen ebenbasierten Graphen bezeichnen wir mit L_n die Menge aller Knoten auf dem n-ten Layer. In einem port- und ebenbasierten Graphen mit einem Port p führen wir die vereinfachende Schreibweise $l(p) = l(v(p))$ ein.

7. Definition (kurze/lange Kanten)**.** Wir bezeichnen eine Kante (v_0, v_1) als *kurz*, falls $l(v_0) + 1 = l(v_1)$ gilt. Ansonsten bezeichnen wir die Kante als *lang*.

8. Definition (reguläre Knoten, Dummy-Knoten)**.** Die Menge der Knoten eines ebenenbasierten Graphen kann unterschieden werden in die Menge der *regulären Knoten* und die Menge der *Dummy-Knoten*. Ein Dummy-Knoten kann einem Graphen während des Layoutprozesses hinzugefügt werden. Dummy-Knoten werden nach Fertigstellung des Layouts wieder entfernt.

Eine nicht strenge Ebenenzuweisung kann stets zu einer strengen Ebenenzuweisung gemacht werden. Hierzu fügt man in alle langen Kanten Dummy-Knoten auf den passierten Layern ein. Im Verlauf dieser Arbeit wird sich zeigen, dass dieses Einfügen im von uns betrachteten Umfeld standardmäßig erfolgen wird. Daher kommen wir zu folgender Konvention.

2.1. Konvention. Wir bezeichnen im Folgenden eine Kante nur dann als kurz, falls sie zwei normale Knoten auf benachbarten Ebenen verbindet. Die einzelnen Abschnitte einer durch Dummy-Knoten unterteilten langen Kante werden wir nicht als kurze Kanten, sondern als *Kantenabschnitte* oder *Kantensegmente* bezeichnen.

Ordnet man die Ebenen einer Ebenenzuweisung in aufsteigender Reihenfolge an, so erhält man eine Hauptrichtung aller Kanten, wie in Abbildung 1.4b zu sehen ist. In der Literatur zum automatischen Graphenlayout ist diese Hauptrichtung eines ebenenbasierten Graphens üblicherweise von oben nach unten. Davon abweichend wird im folgenden davon ausgegangen, dass die Hauptrichtung eines Graphen von links nach rechts ist. Für Datenflussdiagramme, dem Anwendungsfall von KLay Layered, ist dies die übliche Orientierung. Demnach liegen die Eingangsports eines Knotens standardmäßig auf seiner linken, die Ausgangsports standardmäßig auf seiner rechten Seite.

2.1.3. Kurven

In der Einleitung haben wir bereits die Notwendigkeit von Kurven herausgearbeitet. Hier soll noch einmal eine genaue Definition erfolgen. In dieser Arbeit werden alle Kurven in Parameterdarstellung angegeben. Unser Zielbereich ist stets $\mathbb{R}^2$, da wir Kanten in der Ebene darstellen wollen. Zudem einigen wir uns auf einen Definitionsbereich von $[0,1]$. Dann lautet die Definition einer Kurve wie folgt:

9. Definition (Kurve in Parameterdarstellung, Parametrisierung)**.** Eine Abbildung

$$Q : [0,1] \to \mathbb{R}^2,\ Q(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

heißt *Kurve* in *Parameterdarstellung* oder auch *Parametrisierung* der Kurve.

Dabei ist t der Funktionsparameter. Man kann ihn sich vorstellen als eine Zeitachse, die Kurve ist dann die Position auf einem Pfad im Verlauf der Zeit. $x(t)$ definiert die Position auf der x-Achse zum Zeitpunkt t, $y(t)$ dementsprechend die Position auf der y-Achse.

10. Definition (reguläre Kurve)**.** Eine Kurve Q heißt *regulär*, falls für alle $t \in [0,1]$ gilt $\|Q'(t)\| \neq 0$, wobei $\|x\|$ die euklidische Norm von x ist.

Eine Kurve ist also genau dann regulär, wenn die Länge ihres Bewegungsvektors nie Null wird. Bleiben wir im oben eingeführten Bild des Zeitverlaufs, so dürfen wir auf der Kurve nie „stehenbleiben“.

11. Definition (äquvalente Parametrisierung, Reparametrisierung)**.** Es seien $Q(t)$, $t \in [0,1]$ und $R(u)$, $u \in [0,1]$ zwei reguläre, n-fach stetig differenzierbare Kurven. Ihre Parametrisierungen heißen *äquivalent*, wenn eine streng monoton wachsende, n-fach stetig differenzierbare Funktion $f : [0,1] \to [0,1]$ existiert, mit

$$f(0) = 0 \qquad \wedge \qquad f(1) = 1 \qquad \wedge \qquad \forall t \in [0,1] : Q(t) = R(f(t)).$$

$R(u)$ heißt dann auch *Reparametrisierung* von $Q(t)$ [BD89].

12. Definition (Krümmung, Krümmungsradius)**.** Die *Krümmung* $\kappa(t)$ sowie der *Krümmungsradius* $\rho(t)$ einer Kurve Q an der Stelle t ist definiert als

$$\kappa(t) = \frac{Q'(t) \times Q''(t)}{\|Q'(t)\|^3}, \qquad \rho(t) = \frac{1}{\kappa(t)} = \frac{\|Q'(t)\|^3}{Q'(t) \times Q''(t)}.$$

2.1.4. Stetigkeit

Eine Möglichkeit Stetigkeit für reelle Funktionen zu definieren, ist das *Epsilon-Delta-Kriterium*:

13. Definition (stetige Funktion, stetig differenzierbar)**.** Eine Funktion $f\colon \mathbb{R} \to \mathbb{R}$ heißt *stetig im Punkt* $x \in \mathbb{R}$, falls zu jedem $\epsilon > 0$ ein $\delta > 0$ existiert, so dass für alle $x' \in \mathbb{R}$ mit $|x - x'| < \epsilon$ gilt $|f(x) - f(x')| < \delta$. Die Funktion heißt *stetig* (oder *stetig auf* $\mathbb{R}$), wenn sie in jedem Punkt $x \in [0,1]$ stetig ist. Die Funktion heißt *n-mal stetig differenzierbar*, wenn sie n-mal differenzierbar ist und die n-te Ableitung stetig ist.

Diese Definition lässt sich auch auf Kurven anwenden.

14. Definition (parametrische Stetigkeit)**.** Eine Kurve, die im Punkt $t \in [0,1]$ das ϵ-Kriterium erfüllt, heißt *parametrisch stetig* (C^0-stetig) im Punkt t. Die Kurve heißt insgesamt parametrisch stetig, wenn sie in jedem Punkt $t \in [0,1]$ parametrisch stetig ist. Die Kurve heißt *parametrisch stetig n-ten Grades* (C^n-stetig), wenn ihre n-te Ableitung existiert, und diese parametrisch stetig ist.

Bei Kurven gibt es jedoch noch eine zweite, schwächere Form der Stetigkeit: die *geometrische Stetigkeit*. Nach der nun folgenden mathematischen Definition werden wir diese neue Form der Stetigkeit noch einmal mit der parametrischen Stetigkeit vergleichen.

15. Definition (geometrische Stetigkeit)**.** Eine Kurve Q heißt *geometrisch stetig n-ten Grades* (G^n-stetig), falls ihre Reparametrisierung in Bogenmaß C^n-stetig ist.

Zwei weitere Definitionen für geometrische Stetigkeit liefern Barsky und DeRose in ihrer Arbeit [BD89]. In dieser Arbeit werden wir uns nicht mit Stetigkeiten vom Grad > 2 befassen. Daher genügt für das Verständnis dieser Arbeit, folgende griffigere Definition der geometrischen Stetigkeit bis zum zweiten Grad.

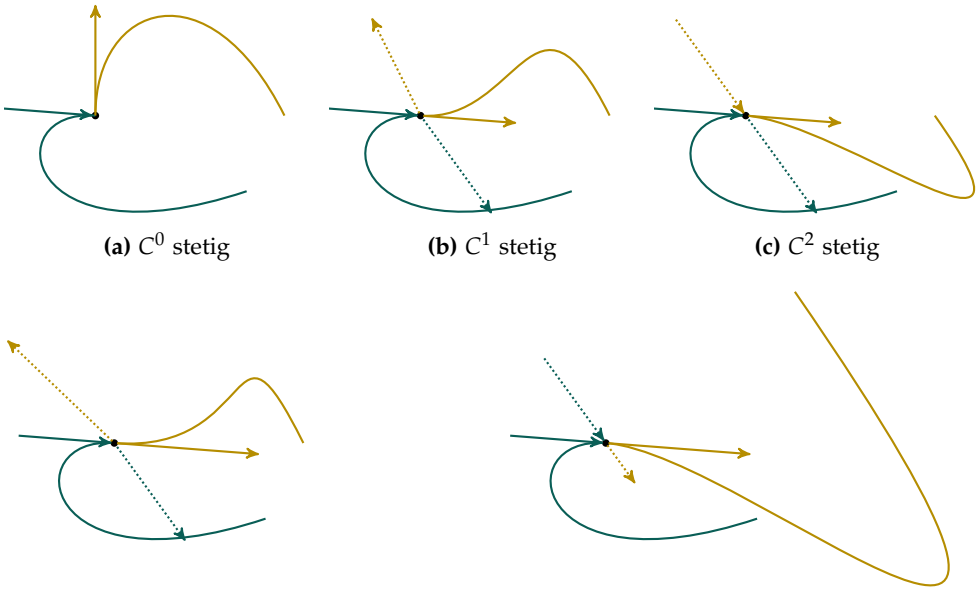
2.2. Korollar. *Für ein* $t \in [0,1]$ *seien die Kurven*

$$\begin{aligned} &Q\colon [0,1] \to \mathbb{R}^2, \\ &Q_0\colon [0,t] \to \mathbb{R}^2,\ Q_0(t') = Q(t') \textit{ für alle } t' \in [0,t], \\ &Q_1\colon [t,1] \to \mathbb{R}^2,\ Q_1(t') = Q(t') \textit{ für alle } t' \in [t,1] \end{aligned}$$

gegeben. Die geometrische Stetigkeit von Q im Punkt t ist dann bis zum zweiten Grad wie folgt definiert.

- *($G^0 \equiv C^0$) Die geometrische Stetigkeit nullten Grades ist äquivalent zur parametrischen Stetigkeit nullten Grades.*
- *(G^1) Q ist in t geometrisch stetig ersten Grades, wenn die Teilkurven Q_0 und Q_1 in t dieselbe* Normalentangente *haben, ihre Tangenten also in die selbe Richtung weisen.*
- *(G^2) Q ist in t geometrisch stetig zweiten Grades, wenn die Teilkurven Q_0 und Q_1 in t dieselbe Krümmung haben. Da Q in diesem Punkt auch G^1-stetig sein muss ist dies äquivalent dazu, dass die* Krümmungskreise *der Teilkurven in t den selben Mittelpunkt haben.*

16. Definition (stetige Fortsetzung von Kurven)**.** Gehen zwei Kurven Q_0 und Q_1 ineinander über, so haben sie einen *Übergangspunkt*. Wir sagen, dass Q_0 von Q_1 C^n- beziehungsweise *G^n-stetig fortgesetzt* wird, wenn die Kurven im Übergangspunkt die entsprechende Stetigkeit aufweisen.



(a) C^0 stetig **(b)** C^1 stetig **(c)** C^2 stetig

(d) G^1 stetig **(e)** G^2 stetig

Abbildung 2.1. Vergleich der verschiedenen Stetigkeiten beim Übergang von zwei Kurven ineinander. Die durchgezogen Vektoren sind die Bewegungsvektoren, die gepunkteten Vektoren sind die Krümmungsvektoren.

Da die C^n-Stetigkeit strenger ist, mag man erwarten, dass aus einer C^n-Stetigkeit auch immer eine G^n-Stetigkeit folgt. Dies stimmt aufgrund der Definition der G^1-Stetigkeit jedoch nicht. Ist die Kurve nicht regulär, so wird ihre erste Ableitung null. Somit besitzt der Bewegungsvektor keine Richtung, kann also auch nicht die selbe Richtung wie der Bewegungsvektor der Fortsetzung haben. Falls eine Kurve regulär ist, so folgt aus einer C^n-Stetigkeit auch stets die G^n-Stetigkeit.

Vergleich parametrischer und geometrischer Stetigkeit

Der Begriff der stetigen Fortsetzung ist gut geeignet, um sich eine bildliche Vorstellung der verschiedenen Stetigkeiten zu machen. Für unseren Anwendungsfall sind nur die Stetigkeiten bis zum Grad zwei interessant. Es seien Q_0 eine Kurve und Q_1 ihre Fortsetzung. Die verschiedenen Stetigkeiten wirken sich dann wie folgt aus:

C^0 Es gilt $Q_0(1) = Q_1(0)$. Die Kurven berühren sich im Übergangspunkt. (Abbildung 2.1a)

C^1 Es gilt C^0 und $Q_0'(1) = Q_1'(0)$. Die Bewegungsvektoren der beiden Kurven sind im Übergangspunkt identisch. (Abbildung 2.1b)

C^2 Es gilt C^1 und $Q_0''(1) = Q_1''(0)$. Die Bewegungsvektoren und die Krümmungsvektoren der beiden Kurven sind im Übergangspunkt identisch. (Abbildung 2.1c)

G^0 Identisch zu C^0. (Abbildung 2.1a)

G^1 Die Bewegungsvektoren der beiden Kurven haben in t die gleiche Richtung, können aber unterschiedliche Beträge aufweisen. (Abbildung 2.1d)

G^2 Die Bewegungsvektoren der beiden Kurven haben in t die gleiche Richtung, und die Krümmungsradien in t sind identisch. (Abbildung 2.1e)

2.2. Ebenenbasiertes Layout

Kozo Sugiyama et al. lieferten in ihrer Arbeit einen neuen Ansatz zum Layout gerichteter Graphen [STT81]. Der KLay Layered-Algorithmus ist eine erweiterte Implementierung seines Ansatzes. In diesem Kapitel wird nun der Sugiyama-Algorithmus vorgestellt. Gleichzeitig werden Besonderheiten bei der Implementierung durch KLay Layered erörtert und abschließend einige Erweiterungen von KLay Layered über den Algorithmus von Sugiyama et al. hinaus aufgezeigt. Hierzu sei auf die Arbeit von Christoph Daniel Schulze verwiesen, aus der ein Großteil der Informationen zu KLay Layered entnommen sind [Sch11]. Der Sugiyama-Algorithmus besteht aus fünf Phasen:

1. Auflösen von Zyklen.
2. Verteilen der Knoten auf den Ebenen.
3. Minimierung von Kantenkreuzungen.
4. Bestimmung der Knotenpositionen auf den Ebenen.
5. Routing der Kanten.

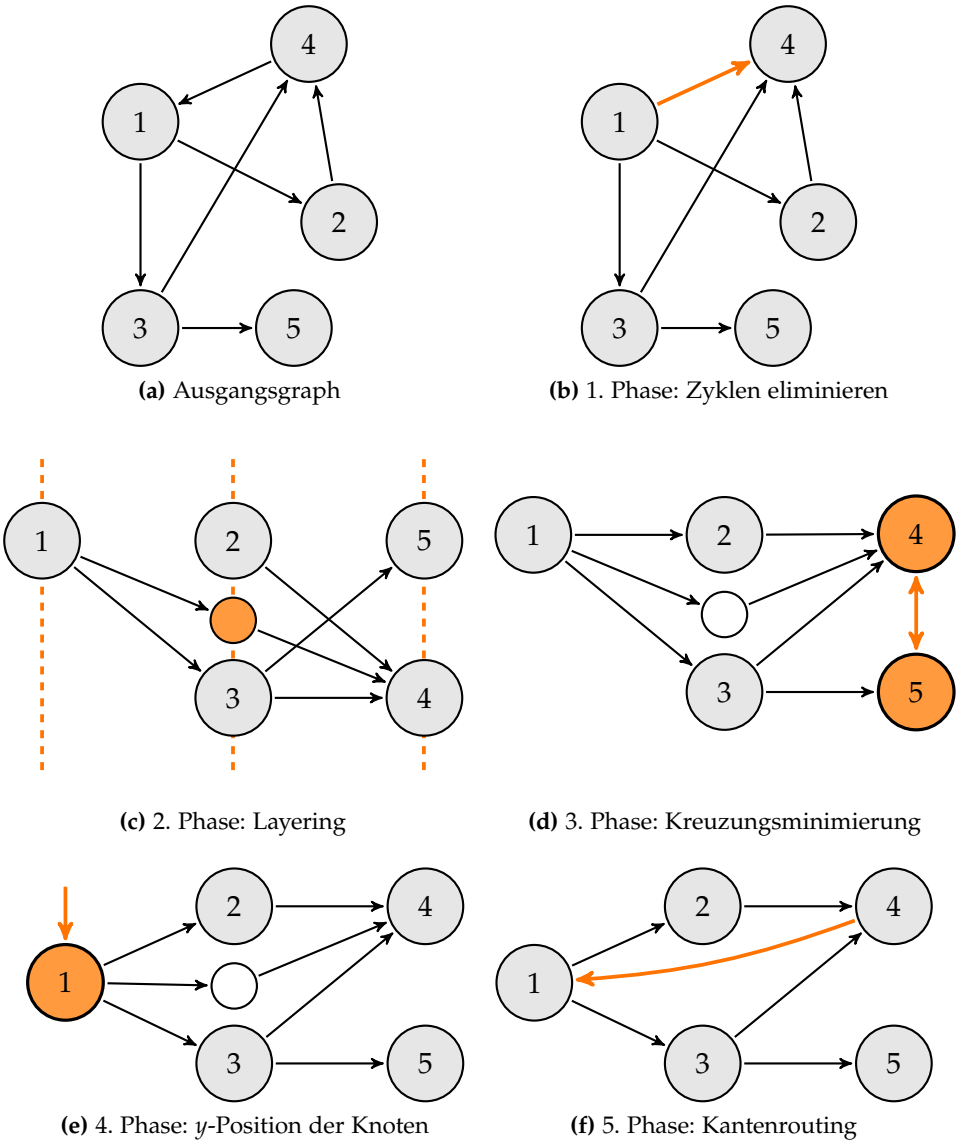
Um die fünf Phasen zu erklären, werden wir einen Beispielgraphen nach diesem Algorithmus layouten. Also sei $G = (V, E)$ ein gerichteter Graph mit (siehe Abbildung 2.2a)

$$V = \{1, 2, 3, 4, 5\},$$
$$E = \{(1, 2), (1, 3), (2, 4), (3, 4), (3, 5), (4, 1)\}.$$

1. Phase: Auflösen von Zyklen

Zusammenfassung: Ein beliebiger, gerichteter Graph wird zu einem azyklischen, gerichteten Graphen umgewandelt.

Beschreibung: (siehe Abbildung 2.2b) In dieser Phase werden alle Zyklen aus dem Graphen eliminiert. Dies geschieht durch Umdrehen von Kanten. Das Problem, eine minimale Menge von Kanten zu finden, die genügen, um den Graphen azyklisch zu machen, nennt man auch *Minimum Feedback Arc Set (FAS)*. Dieses Problem ist für gerichtete Graphen NP-vollständig [Kar72]. KLay Layered verwendet daher eine Heuristik zum



(a) Ausgangsgraph

(b) 1. Phase: Zyklen eliminieren

(c) 2. Phase: Layering

(d) 3. Phase: Kreuzungsminimierung

(e) 4. Phase: y-Position der Knoten

(f) 5. Phase: Kantenrouting

Abbildung 2.2. Die fünf Phasen des Algorithmus von Sugiyama et al. Die Änderungen in den einzelnen Phasen sind farblich hervorgehoben.

Berechnen einer geeigneten Kantenmenge. Es wird ein gieriger Algorithmus basierend auf Di Battista verwendet [DETT98].

2. Phase: Verteilen der Knoten auf die Ebenen

Zusammenfassung: Zu einem azyklischen, gerichteten Graphen wird eine strenge Ebenenzuweisung gefunden.

Beschreibung: (siehe Abbildung 2.2c) In der zweiten Phase werden die Ebenen eingeführt. Das Ziel ist es, eine strenge Ebenenzuweisung zu finden. Hierzu wird zunächst eine einfache Ebenenzuweisung gesucht. In KLay Layered sind hierzu verschiedene Algorithmen implementiert, die verschiedenen Ästhetikkriterien genügen. Ist eine einfache Ebenenzuweisung gefunden, so werden für alle langen Kanten Dummy-Knoten auf den von ihnen passierten Ebenen eingefügt. Nun sind alle Kanten kurze Kanten, und eine strenge Ebenenzuweisung ist gefunden.

3. Phase: Minimierung von Kantenkreuzungen

Zusammenfassung: Zu einem Graphen mit strenger Ebenenzuweisung wird die Reihenfolge der Knoten in den Layern festgelegt.

Beschreibung: (siehe Abbildung 2.2d) In dieser Phase wird die Anzahl der Kantenkreuzungen minimiert. Dies geschieht durch Variieren der Reihenfolge der Knoten in jeder Ebene. Da dieses Problem bereits für zwei Ebenen NP-vollständig ist [GJ83], muss das Problem auch hier wieder mittels einer Heuristik gelöst werden. Gegenwärtig gibt es in KLay Layered nur eine Implementierung einer solchen Heuristik, den *LayerSweepCrossingMinimizer*. In mehreren Layer-Durchläufen (engl. *Layer-Sweeps*) minimiert dieser die Anzahl der Kantenkreuzungen zwischen zwei benachbarten Ebenen mittels einer Schwerpunkt (engl. *barycenter*)-Heuristik.

4. Phase: Bestimmung der vertikalen Knotenposition

Zusammenfassung: Zu einem Graphen mit strenger Ebenenzuweisung und fester Knotenreihenfolge wird die y-Position der Knoten bestimmt. Vor dieser Phase muss zudem die Portreihenfolge an den Knoten feststehen.

Beschreibung: (siehe Abbildung 2.2e) In der vierten Phase werden nun die genauen y-Positionen der Knoten bestimmt. Durch ihre Positionierung wird die Anzahl der Kantenknicke beeinflusst. Aus der vertikalen Positionierung der Knoten folgt auch die Höhe der einzelnen Ebenen und des Graphens. Auch für diese Phase stehen in KLay Layered mehrere Algorithmen zur Verfügung. Als Standard wird zur Zeit eine Implementierung eines Algorithmus von Brandes und Köpf eingesetzt [BK02]. Die Höhe des Graphen entspricht gerade der maximalen Layerhöhe.

5. Phase: Routing der Kanten

Zusammenfassung: Zu einem Graphen mit strenger Ebenenzuweisung und fester y-Position der Ports werden die Biegepunkte der Kanten bestimmt.

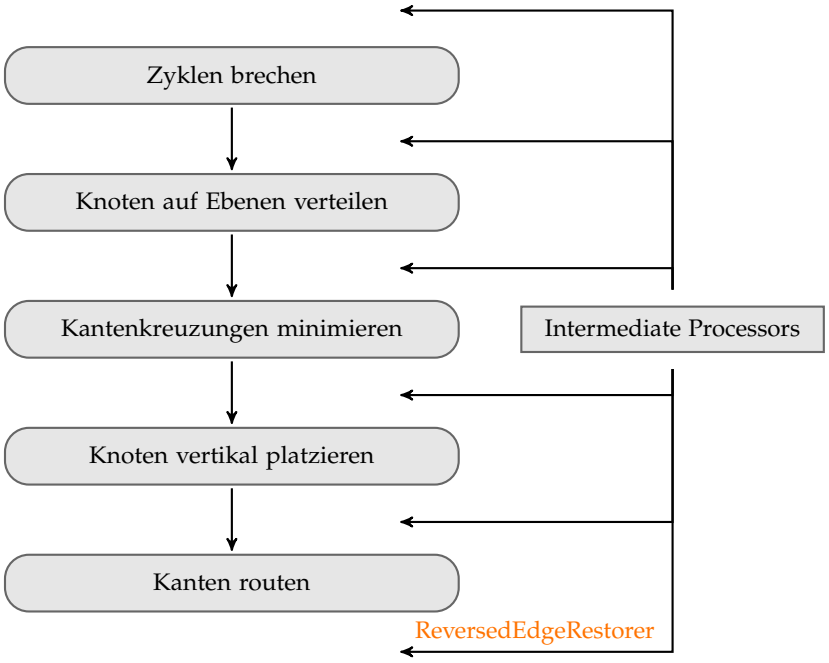


Abbildung 2.3. Vor, zwischen und nach den einzelnen Phasen des Algorithmus von Sugiyama et al. können in KLay Layered Zwischenprozessoren ausgeführt werden. Der ReversedEdgeRestorer ist einer von ihnen. Er wird nach der fünften Phase ausgeführt.

Beschreibung: (siehe Abbildung 2.2f) In der letzten Phase werden die Kanten geroutet. Hier stellen sich potentiell verschiedene Aufgaben. Je nach Art des Kantenroutings müssen jedoch nicht alle dieser Teilaufgaben vollständig erfüllt werden.

1. Bestimmung der x-Koordinaten-Bereiche der Ebenen. Alle Knoten einer Ebene müssen in diesem Bereich liegen.
2. Bestimmung der x-Koordinaten der Knoten.
3. Routing von Self-Loops.
4. Routing von Kanten, deren Ports unübliche Positionierungen (zum Beispiel Nord- oder Südseite des Knotens) aufweisen.
5. Bestimmung von Knickpunkten der Kanten.

2.2.1. Erweiterungen in KLay Layered

Gegenüber dem Vorgehen von Sugiyama et al. weist KLay Layered einige Erweiterungen auf. Die zwei wichtigsten Erweiterungen, die auch für diese Arbeit von Relevanz sind, werden im Folgenden erläutert.

Zwischenprozessoren Betrachtet man Abbildung 2.2f, so fällt auf, dass hier noch eine weitere Veränderung gegenüber Abbildung 2.2e vorgenommen wurde, die in der Beschreibung der fünften Phase nicht erwähnt wurde: Die Kante von 4 nach 1, welche wir in der ersten Phase zur Auflösung von Zyklen umdrehten, wurde wieder in ihre ursprüngliche Richtung gedreht. Dies wird im KLay Layered-Algorithmus in einem sogenannte *Zwischenprozessor* (engl. *Intermediate Processor*) umgesetzt. Diese Zwischenprozessoren, von denen es eine Vielzahl gibt, können vor oder nach jeder Phase eingesetzt werden. In diesem konkreten Fall wird das Wiederherstellen der ursprünglichen Kantenrichtung von dem *ReversedEdgeRestorer* nach Phase 5 realisiert. Siehe hierzu auch Abbildung 2.3.

Ports In KLay Layered sind die Kanten nicht direkt an die Knoten gebunden, sondern sie verlaufen immer zwischen zwei Ports. Diese Ports liegen wiederum an Knoten. Der Sugiyama-Algorithmus behandelt in seiner ursprünglichen Form keine Ports, es handelt sich hierbei also um eine KLay Layered-eigene Erweiterung. Gegenwärtig unterstützt KLay Layered keine Ports, die sowohl aus- als auch eingehende Kanten haben. Somit kann jedem Port ein *PortType* zugeordnet werden, der zwischen Ports mit eingehenden und ausgehenden Kanten differenziert.

Liegen an einem Port mehrere Kanten, so sprechen wir von einer *Hyperkante*. Da die Kanten an diesem Port sehr eng gedrängt liegen ist sinnvoll für diese Hyperkanten ein spezielles Routing zu entwickeln, was wir in Unterkapitel 3.5 auch tun werden. Auch für Hyperkanten gilt, dass jeder Port entweder ein- oder ausgehende Kanten hat.

17. Definition (Portseite)**.** Jeder Port hat eine *Portseite* (engl. *port side*), an der er am Knoten liegt. Wir unterscheiden in Nord (N), Ost (E), Süd (S) und West (W). Zudem kann die Portseite undefiniert (U) sein. Für einen Port p definieren wir $ps(p)$ als die Portseite von p.

In KLay Layered kann die Portseite bereits in der Graphendefinition festgelegt werden. Spätestens nach dem *PortSideProcessor*, einem Zwischenprozessor, gilt $ps(p) \neq u$. Der PortSideProcessor kann entweder vor Phase 1 oder vor Phase 3 ausgeführt werden.

2.3. Splines

Um die in der Einleitung gestellte Aufgabe der glatten Kurven zu lösen bieten sich verschiedene Ansätze. Man könnte die Knickpunkte des orthogonalen Kantenroutings beispielsweise mit Radien versehen. Wir wollen jedoch noch einen Schritt weiter gehen. In Definition 14 wurde die parametrische Stetigkeit vorgestellt. Von besonderem Interesse ist hier die C^2-Stetigkeit. Eine Kurve, welche diese Eigenschaft erfüllt, würde zu ästhetisch ansprechenden Ergebnissen führen: sie würde nicht schlagartig ihre Krümmung ändern, sondern allmählich. Doch wie erreichen wir, dass unsere Kanten in ihrem Verlauf C^2-stetig sind? Das wollen wir in diesem Kapitel erörtern. Dazu wird zunächst das Konstrukt der *Splines* motiviert, und anschließend verbreitete Formen der Splines vorgestellt.

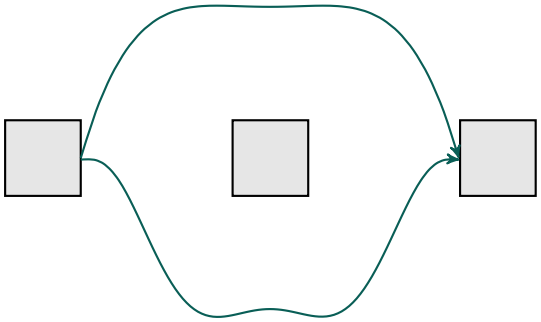


Abbildung 2.4. Runges Phänomen in Graphen. Obere Kurve mit fünf, untere Kurve mit neun definierten Koordinaten.

2.3.1. Das Problem der parametrischen Stetigkeit

Betrachten wir unsere Aufgabe einmal mathematisch. Wir wollen in unserem ebenenbasierten Graphen eine Kante zwischen zwei Knoten ziehen. Dies eventuell über mehrere Ebenen, wobei uns die Dummy-Knoten den Pfad vorgeben. Diese Kante soll C^2-stetig verlaufen. Also sind eine Anzahl paarweise verschiedener Punkte $(x_0, y_0), \ldots, (x_n, y_n)$ in der Ebene durch eine C^2-stetige Kurve miteinander zu verbinden. Der naive Ansatz ist, eine polynomielle Kurve $Q(t)$ vom Grad n zu nutzen.

$$Q(t) = \begin{pmatrix} f_x(t) \\ f_y(t) \end{pmatrix} = \begin{pmatrix} a_0^x \\ a_0^y \end{pmatrix} + \begin{pmatrix} a_1^x \\ a_1^y \end{pmatrix} t + \begin{pmatrix} a_2^x \\ a_2^y \end{pmatrix} t^2 + \ldots + \begin{pmatrix} a_n^x \\ a_n^y \end{pmatrix} t^n \quad \text{mit}$$

$$\forall i \in \{0, \ldots, n\} : Q(t_i) = \begin{pmatrix} f_x(t_i) \\ f_y(t_i) \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

Wir können die beiden Funktionen $f_x(t)$ und $f_y(t)$ separat betrachten. In beiden Funktionen haben wir $n+1$ Variablen und $n+1$ bekannte Koordinaten. Das Polynom ist somit lösbar. Polynome hohen Grades können jedoch starke Ausschläge aufweisen. Abbildung 2.4 zeigt einen Versuch, zwei Knoten um einen dritten Knoten herum mit einer Kante zu verbinden. Der erste Ansatz mit fünf festgelegten Koordinaten führt zu einem akzeptablen Ergebnis. Wollten wir jedoch erreichen, dass unsere Kanten in einem rechten Winkel die äußeren Knoten berühren, so müssten wir zwei weitere Koordinaten einfügen. Man erkennt, dass die Kurve nun deutlich stärkere Ausschläge aufweist: sie *überschwingt*.

Dies ist auch unter dem Namen *Runges Phänomen* nach Carl David Tolmé Runge bekannt [Run01]. Versucht man, die *Runge-Funktion* $\left(\frac{1}{1+x^2}\right)$ mithilfe von Polynomen zu interpolieren, so führt eine Erhöhung der Anzahl von Stützpunkten nicht zu einer Verbesserung der Interpolation, sondern das Polynom beginnt stark zu oszillieren, (siehe Abbildung 2.5).

Wir benötigen also einen anderen Ansatz, um C^2-stetige Kanten zu erzeugen. Greifen wir auf eine bewährte Strategie zurück, um solche Aufgabenstellungen zu lösen: *Divide and Conquer*.

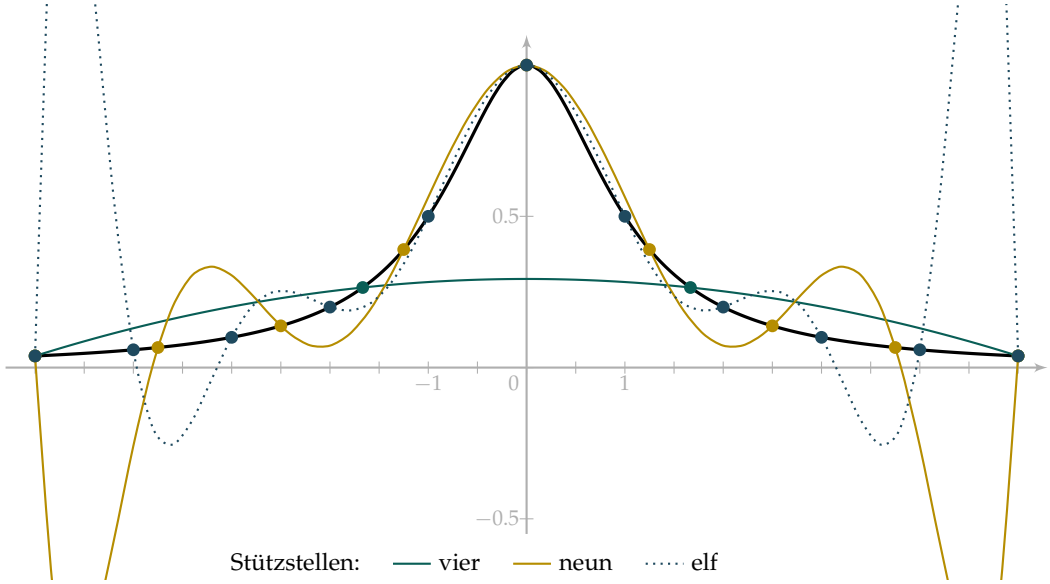


Abbildung 2.5. Runges-Phänomen. Versuch, die Runge-Funktion $\left(\frac{1}{1+x^2}\right)$ mithilfe von vier, neun und elf Stützstellen zu interpolieren.

2.3.2. Teile und herrsche

Wir teilen die Aufgabe, die $n+1$ Punkte zu verbinden, in n einfache Aufgaben, jeweils zwei benachbarte Punkte mit einem Polynom zu verbinden. Zusätzlich fordern wir an den Verbindungsstellen einen angenehmen Kurvenverlauf, also C^2-Stetigkeit. Ein derart zusammengesetztes Polynom nennt man auch *Spline*.

18. Definition (Spline)**.** Ein *Spline n-ten Grades* ist eine stückweise aus Polynomen höchstens n-ten Grades zusammengesetzte Funktion. Seine Einzelstücke sind also beliebig oft stetig differenzierbar, sie sind somit C^∞-stetig. Ein Spline n-ten Grades heißt C^n-, beziehungsweise G^n-stetig, falls er auch an den Übergangsstellen der Polynome die entsprechende Stetigkeit aufweist.

19. Definition (linearer, quadratischer, kubischer Spline)**.** Einen Spline ersten Grades nennt man auch *linear*. Einen Spline zweiten Graden nennt man auch *quadratisch*, einen dritten Grades *kubisch*.

Ursprünglich stammt der Begriff Spline aus dem Handwerk. Im Schiffbau wurden sowohl beim Bau als auch beim Entwurf schlanke, quadratische Holzleisten verwendet. Biegt man eine solche Leiste um mehrere Fixierungspunke, auch *Stützstellen* genannt, so nimmt sie stets die Form der geringsten Biegeenergie an, was einen harmonischen Verlauf ergibt [BBB87]. Diese Holzleisten wurden im Deutschen als *Straklatte*, im Englischen als Spline bezeichnet. Isaak Jakob Schoenberg führte den Begriff des Splines erstmals in der Mathematik ein [Sch46].

2.3.3. Kubische Splines

Im vorhergehenden Abschnitt haben wir festgestellt, dass Polynome hohen Grades für unsere Anwendung nicht geeignet sind. In diesem Abschnitt wollen wir nun die Frage beantworten, welcher Dimension ein Spline mindestens sein muss, um für die gestellte Aufgabe geeignet zu sein.

Wir haben $n+1$ Stützstellen $v_0 = (x_0, y_0) \dots v_n = (x_n, y_n)$ gegeben, und suchen nun einen Spline möglichst geringes Grades, bestehend aus n Kurven $Q_0 \dots Q_{n-1}$, die mindestens zwei mal stetig differenzierbar sind. Folgende Bedingungen müssen erfüllt werden:

- C^0: Jedes Polynom Q_i soll durch die Punkte v_i und v_{i+1} laufen. Also muss gelten $Q_i(0) = v_i$ und $Q_i(1) = v_{i+1}$.
- C^1: An allen Verbindungsstellen $v_i \in \{v_1, \dots, v_{n-1}\}$ sollen die Bewegungsvektoren der benachbarten Kurven Q_{i-1} und Q_i identisch sein, also $Q'_{i-1}(1) = Q'_i(0)$.
- C^2: An allen Verbindungsstellen $v_i \in \{v_1, \dots, v_{n-1}\}$ sollen die Krümmungsvektoren der benachbarten Polynome Q_{i-1} und Q_i identisch sein. Es gilt also $Q''_{i-1}(1) = Q''_i(0)$.

Für einen Spline aus n Kurven ergeben sich also $2n$ Bedingungen aus C^0, $n-1$ Bedingungen aus C^1 und $n-1$ Bedingungen aus C^2. Insgesamt sind $4n-2$ Bedingungen gegeben. Es werden also Kurven mindestens dritten Grades benötigt, da wir dann $4n$ Freiheitsgrade zur Verfügung haben. Die zwei verblieben Freiheitsgrade lassen sich verwenden, um beispielsweise eine Steigung im Start- und Endpunkt zu definieren.

Ein so zusammengesetzter Spline wird auch als *natürlicher kubischer Spline* bezeichnet. Dieses mathematische Modell entspricht recht genau den physikalischen Zusammenhängen der oben erwähnten Straklatte, wie Thomas Sonar ausführt [Son01]. Der natürliche Spline ist eine Lösung für unsere Aufgabe, besitzt jedoch verschiedene Nachteile:

- Die natürlichen Splines weisen keinerlei *Lokalität* auf. Das bedeutet eine Veränderung an einem der Stützstellen wirkt sich auf den gesamten Spline aus. Dies führt zu einem schwer kontrollierbaren Verhalten.
- Aus dieser Gesamtabhängigkeit folgt auch eine schlechte Parallelisierbarkeit der Berechnung.
- Das einzige Mittel zur Beeinflussung des Kantenverlaufs ist die Positionierung der Stützstellen. Die Steigungen und Krümmungen an den Stützstellen ergeben sich aus dem Gesamtverlauf des Splines. Es wäre wünschenswert, diese festsetzen zu können.

Wir werden in den folgenden Kapiteln weitere Splines vorgestellt, die sich in der Praxis durchgesetzt haben, und die erwähnten Nachteile der natürlichen Splines nicht aufweisen.

2.4. Bezier-Splines

Die *Bezier-Splines* gehören zu den bekanntesten und verbreitetsten Splines. Jedes moderne Grafikframework bietet eine Möglichkeit, sie darzustellen. Sie zeichnen sich unter anderem durch eine sehr einfache Berechenbarkeit in beliebiger Genauigkeit aus. Die Bezier-Splines wurden parallel und unabhängig von Pierre Bézier und Paul de Casteljau entwickelt. Bevor wir zu den Splines kommen, werden im folgenden Abschnitt zunächst ihre Komponenten, die Bezierkurven, behandelt.

2.4.1. Bezierkurve

20. Definition (Bernstein-Polynome)**.** Die *Bernstein-Polynome* vom Grad n sind für $t \in [0,1]$ und $i \in \{0,\dots,n\}$ definiert als

$$\mathcal{B}_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$

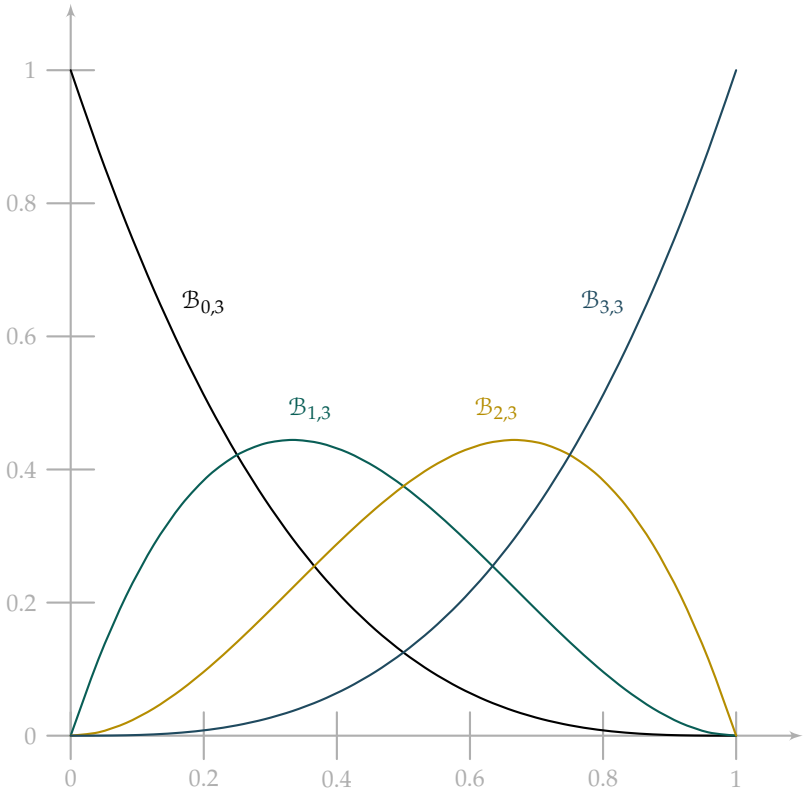
21. Definition (Bezierkurve)**.** Eine *Bezierkurve* n-ten Grades wird durch $n+1$ Kontrollpunkte (*Bezier Punkte*) definiert, welche das *Kontrollpolygon* bilden. Eine Bezierkurve Q mit Kontrollpunkten $v_0,\dots,v_n$ ist für $t \in [0,1]$ definiert als

$$Q(t) = \sum_{i=0}^{n} \mathcal{B}_{i,n}(t) v_i.$$

Die Bernstein-Polynome bestimmen, welchen Einfluss die einzelnen Kontrollpunkte auf die Kurve im Verlauf von t haben. Dabei legt $\mathcal{B}_{i,n}$ den Einfluss von v_i fest. In Abbildung 2.6 sind sie für den kubischen Fall, also für $n = 3$, dargestellt. Betrachtet man den Verlauf dieser Polynome, so bekommt man schon eine gute Vorstellung vom Verlauf der Kurve. Sie beginnt in v_0, krümmt sich im Verlauf der Reihe nach in Richtung der Punkte v_1 bis v_{n-1}, um schließlich in v_n anzukommen. Dieser Verlauf lässt sich gut am Beispiel der Abbildung 2.7 nachvollziehen.

Es gibt einige weitere Eigenschaften der Bezierkurven, welche für die Arbeit mit ihnen wichtig sind:

- Die Tatsache, dass die Kurve durch ihre Endpunkte verläuft, nennt man auch *Endpunktinterpolation*.
- Für $t \in]0,1[$ gilt stets $\mathcal{B}_{i,n}(t) > 0$. Dies hat zur Folge, dass jeder Kontrollpunkt einen Einfluss auf die gesamte Kurve hat, sein Einfluss also *global* ist. Das Gegenteil wäre ein *lokaler* Einfluss.
- Für $t \in [0,1]$ ist die Summe der Bernstein-Polynome gleich eins. Man sagt, die Bernstein-Polynome sind eine *Zerlegung der Eins*. Dies hat zur Folge, dass die Kurve in der *konvexen Hülle* des Kontrollpolygons liegt. Diese Eigenschaft kann man gut zur Kollisionsabfrage nutzen.



$$\mathcal{B}_{0,3}(t) = (1-t)^3$$
$$\mathcal{B}_{1,3}(t) = 3t(1-t)^2$$
$$\mathcal{B}_{2,3}(t) = 3t^2(1-t)^1$$
$$\mathcal{B}_{3,3}(t) = t^3$$

Abbildung 2.6. Die Bernstein-Polynome für $n = 3$

- Die Kurve kann an einem beliebigen Punkt in zwei Bezierkurven gleicher Ordnung geteilt werden, die zusammengesetzt deckungsgleich mit der Ausgangskurve sind. Auf diese Eigenschaft werden wir später noch genauer zurückkommen.
- Man kann jede Bezierkurve vom Grad n und den Kontrollpunkten $v_0, \ldots, v_n$ durch eine Bezierkurve vom Grad $n+1$ und den Kontrollpunkten $v'_0, \ldots, v'_{n+1}$ darstellen. Die neuen Kontrollpunkte lauten dann

$$v'_i = \begin{cases} \frac{i}{n+1} v_{i-1} + \left(1 - \frac{i}{n+1}\right) v_i & \text{für } i \in \{1, \ldots, n\} \\ v_0 & \text{für } i = 0 \\ v_n & \text{für } i = n+1. \end{cases}$$

- Die Kurve bildet einer gerade Linie genau dann, wenn alle Kontrollpunkte *kollinear* sind, sie also auf einer Geraden liegen.
- Eine *affine Transformation* (Verschiebung, Skalierung, Rotation, Scherung) der Kurve erreicht man durch Anwenden der Transformation auf das Kontrollpolygon.
- Der Tangentenvektor im Start- und Zielpunkt der Kurve entspricht gerade $3\overrightarrow{v_0 v_1}$ bezie-

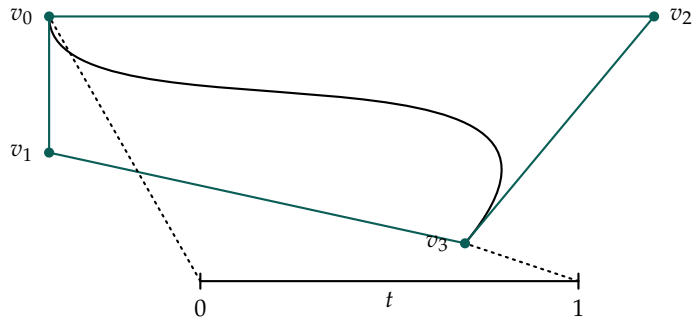


Abbildung 2.7. Eine kubische Bezier Kurve und ihr Kontrollpolygon.

hungsweise $3\overrightarrow{v_{n-1}v_n}$.

2.4.2. Der De-Casteljau-Algorithmus

In der Computergrafik werden Kurven durch Geraden, genauer durch einen Polynomenzug, dargestellt. Unterteilt man eine Kurve in hinreichend viele Segmente, so kann eine optisch glatte Kurve aus einer entsprechenden Anzahl Geraden zusammengesetzt werden.

In der Aufzählung wichtiger Eigenschaften der Bezierkurven, haben wir die Teilbarkeit in Unterkurven erwähnt. Ein Algorithmus, der diese Unterteilung auf effiziente Art vornimmt, ist nach dem zweiten Entwickler der Bezierkurven benannt, Paul de Casteljau. Es gibt effizientere Methoden, um mehrere Punkte auf einer Bezierkurve zu berechnen [PD00]. Der De-Casteljau-Algorithmus ist jedoch leicht verständlich und *numerisch stabil*. Dies bedeutet, dass beispielsweise Rundungsfehler bei der Berechnung sich nicht übermäßig stark auf die Berechnung auswirken. Die Berechnung eines Punktes auf der Bezierkurve durch Auswertung der Bernstein-Polynome ist hingegen numerisch instabil. Eine Rundungsungenauigkeit wirkt sich bei größeren Koordinaten der Kontrollpunkte entsprechend stärker auf das Ergebnis aus.

22. Definition (De-Casteljau-Algorithmus)**.** Gegeben sei eine Bezierkurve $Q(t)$, $t \in [0,1]$, vom Grad n und ihre Kontrollpunkte seien $v_0^0, \ldots, v_n^0$. Für $u \in]0,1[$ lässt sich nun $Q(u)$ wie folgt rekursiv berechnen.

$$\begin{aligned} Q(u) &= v_0^n && \text{mit} \\ v_i^{k+1} &= (1-u)v_i^k + u \cdot v_{i+1}^k && \text{mit } k \in \mathbb{N} \end{aligned}$$

Die Punkte $v_0^0, \ldots, v_0^n$ und $v_0^n, \ldots, v_n^0$ bilden zwei Unterkurven, welche wiederum Bezierkurven vom Grad n sind, und die Kurve $Q(t)$ überdecken.

Abbildung 2.8 zeigt den Algorithmus exemplarisch für eine kubische Bezierkurve. Da auch die Unterkurven wieder Bezierkurven sind, gilt auch für sie die Eigenschaft der Endpunktinterpolation. So lässt sich für ein beliebiges t der entsprechende Punkt auf

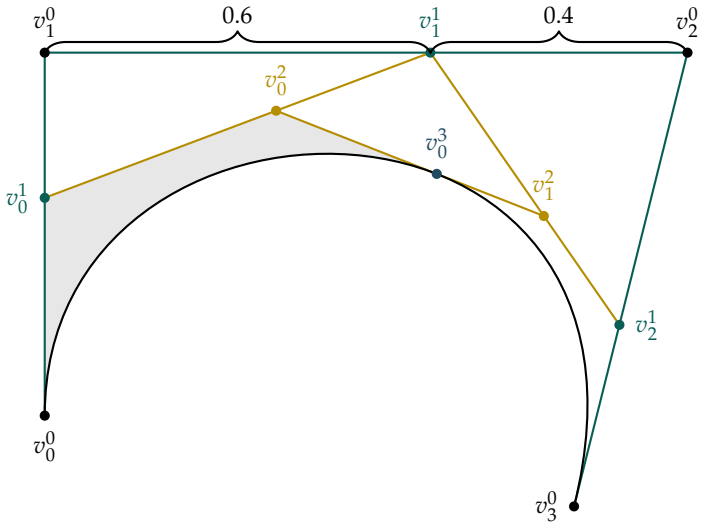


Abbildung 2.8. Der De-Casteljau-Algorithmus für $u = 0,6$. Der ausgefüllte Bereich stellt einen überdeckenden Bezier-Spline von v_0^0 nach v_0^3 mit den inneren Kontrollpunkten v_0^1 und v_0^2 dar.

der Kurve leicht berechnen. Für die Unterteilung der Bezierkurve in zwei Unterkurven werden $\frac{n(n+1)}{2}$ Additionen und $n(n+1)$ Multiplikationen benötigt.

Bevor wir die Kurven zu Splines zusammenfügen, wollen wir uns kurz mit der Ableitung von Bezierkurven beschäftigen.

2.4.3. Ableitung von Bezierkurven

2.3. Satz. *Es sei $Q(t)$ eine Bezierkurve vom Grad n und $v_0, \dots, v_n$ ihre Kontrollpunkte. Die erste Ableitung von Q lautet dann*

$$Q'(t) = \sum_{i=0}^{n-1} \mathcal{B}_{n-1,i}(t) w_i \quad \textit{mit}$$
$$w_i = n(v_{i+1} - v_i) \qquad i \in \{0, \dots, n-1\} \tag{2.4}$$

Q' ist dann eine Bezierkurve vom Grad $n-1$ und Kontrollpunkten $w_0, \dots, w_{n-1}$.

Beweis. Q hat nach Definition 21 folgende Form:

$$Q(t) = \sum_{i=0}^{n} \mathcal{B}_{n,i}(t) v_i \quad \text{mit} \quad \mathcal{B}_{n,i}(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}.$$

Die Kontrollpunkte sind unabhängig von t. Somit reduziert sich die Errechnung der Ableitung von $Q(t)$ zur Errechnung der Ableitungen der Bernstein-Polynome $\mathcal{B}_{n,i}(t)$. Dies ist für $i \in \{0, \dots, n\}$

$$\frac{d}{dt}\mathcal{B}_{i,n}(t) = \mathcal{B}'_{n,i}(t) = n(\mathcal{B}_{n-1,i-1}(t) - \mathcal{B}_{n-1,i}(t)),$$

wobei $\mathcal{B}_{n-1,-1} = \mathcal{B}_{n-1,n} = 0$ ist. Die Ableitung errechnet sich also aus der Differenz zweier Bezierkurven vom Grad $n-1$. Setzen wir dies in die Formel von $Q(t)$ ein, so erhalten wir

$$\begin{aligned}\frac{d}{dt}Q(t) = Q'(t) &= n(-\mathcal{B}_{n-1,0}(t))v_0 \\ &\quad + n(\mathcal{B}_{n-1,0}(t) - \mathcal{B}_{n-1,1}(t))v_1 \\ &\quad + \dots \\ &\quad + n(\mathcal{B}_{n-1,n-1}(t) - \mathcal{B}_{n-1,n}(t))v_n \\ &= \sum_{i=0}^{n-1} \mathcal{B}_{n-1,i}(t)(v_{i+1} - v_i).\end{aligned}$$

Für $m = n-1$ gilt

$$Q'(t) = \sum_{i=0}^{n-1} \mathcal{B}_{n-1,i}(t)w_i = \sum_{i=0}^{m} \mathcal{B}_{m,i}(t)w_i.$$

Somit ist $Q'(t)$ wieder eine Bezierkurve. □

Insbesondere gilt für die erste und zweite Ableitung an den Endpunkten

$$\begin{aligned}Q'(0) &= n(v_1 - v_0), \\ Q'(1) &= n(v_n - v_{n-1}), \\ Q''(0) &= (n-1)(w_1 - w_0) && \text{mit Gleichung 2.4} \\ &= (n-1)\left(n\left(v_2 - v_1\right) - n\left(v_1 - v_0\right)\right) \\ &= n(n-1)(v_2 - 2v_1 + v_0), \\ Q''(1) &= n(n-1)(v_n - 2v_{n-1} + v_{n-2}).\end{aligned}$$

Mit den Bezierkurven haben wir eine Möglichkeit gefunden, zwei Punkte unseres Kurvenpfades miteinander zu verbinden. Um den gesamten Kurvenverlauf darzustellen, müssen wir die Bezierkurven zu einer Kette verbinden. Diese aneinandergereihten Kurven nennt man *Bezier-Spline*.

2.4.4. Bezier-Splines

Wie in Definition 18 eingeführt können wir eine Menge von Bezierkurven zu einem Bezier-Spline verbinden. Stellen wir keine weiteren Anforderungen an die Verbindungsstellen, so erhalten wir auf dem Spline eine lokale Kontrolle. Die Verschiebung eines Kontrollpunktes wirkt sich nur auf die Bezierkurven aus, von denen er ein Teil ist, also maximal zwei Kurven. Um einen harmonischen Kantenverlauf zu erreichen, müssen wir jedoch Stetigkeitsanforderungen an die Verbindungspunkte stellen, wie sie in Definition 14 und Definition 15 definiert wurden. Doch hierdurch verlieren wir die erreichte lokale Kontrolle.

2.5. Satz. *Soll eine o.B.d.A. kubische Bezierkurve Q mit Kontrollpunkten $v_0, \ldots, v_3$ parametrisch stetig fortgesetzt werden, so ergibt sich eine feste Position für einige der Kontrollpunkte der fortsetzenden Kurve, in Abhängigkeit von der geforderten Stetigkeit:*

- *C^0: Erster Kontrollpunkt ist fest.*
- *C^1: Die ersten zwei Kontrollpunkte sind fest.*
- *C^2: Die ersten drei Kontrollpunkte sind fest.*

Beweis. Die kubische Bezierkurve R mit Kontrollpunkten $w_0, \ldots, w_3$ sei die parametrisch stetige Fortsetzung von Q.

1. Fall: C^0

Nach Definition 14 muss dann $Q(1) = R(0)$, und somit $w_0 = v_3$ gelten.

2. Fall: C^1

Es muss wieder $w_0 = v_3$ gelten, da aus C^1 auch C^0 folgt. Zudem muss nach Definition 14 auch $Q'(1) = R'(0)$ gelten. Es gilt also

$$Q'(1) = 3(v_3 - v_2) \qquad \text{Bemerkung nach Satz 2.3} \tag{2.6}$$
$$R'(0) = 3(w_1 - w_0) \qquad \text{Bemerkung nach Satz 2.3} \tag{2.7}$$

Damit ergibt sich:

$$\begin{aligned} & Q'(1) = R'(0) && \text{nach Definition 14} && (2.8)\\ \Leftrightarrow\ & 3(v_3 - v_2) = 3(w_1 - w_0) && \text{Gleichungen 2.6 und 2.7 in 2.8} \\ \Leftrightarrow\ & v_3 - v_2 = w_1 - v_3 && \text{da } w_0 = v_3 \\ \Leftrightarrow\ & w_1 = 2v_3 - v_2 && && (2.9) \end{aligned}$$

3. Fall: C^2

Erneut gelten auch C^1 und C^0, und somit die Bedingungen an w_0, w_1 wie in den vorhergehenden Fällen. Nach Definition 14 muss weiterhin $Q''(1) = R''(0)$ gelten. Analog zum zweiten Fall gilt dann

$$\begin{aligned} v_3 - 2v_2 + v_1 &= w_0 - 2w_1 + w_2 && \text{Bemerkung nach Satz 2.3} \\ v_3 - 2v_2 + v_1 &= v_3 - 2(2v_3 - v_2) + w_2 && \text{nach Gleichung 2.9} \\ w_2 &= -2v_2 + v_1 + 4v_3 - 2v_2 \\ w_2 &= v_1 + 4(v_3 - v_2) \end{aligned}$$

□

Fordern wir also C^2-Stetigkeit von unseren Splines, so sind die einzigen frei platzierbaren Kontrollpunkte die Verbindungsstellen und die inneren Kontrollpunkte der ersten Bezierkurve. Da wir die Verbindungsstellen gerade auf unsere Dummy-Knoten setzen wollen, können wir den Kurvenverlauf nur noch durch Variieren der zwei inneren

Kontrollpunkte der ersten Kurve beeinflussen. Wir haben keine lokale Kontrolle mehr, obwohl wir doch gerade das durch die Nutzung der Splines erreichen wollten. Eine Lösung ist die Abschwächung unserer Übergangsbedingungen. Wir haben bisher C^2-Stetigkeit gefordert. Im folgenden Abschnitt betrachten wir, welche Möglichkeiten sich ergeben, wenn wir die Bedingung auf G^2-Stetigkeit abschwächen.

2.5. Beta-Bezier-Splines

Wir haben nach Definition 15 bereits auf die Arbeit von Barksy und DeRose verwiesen, in der sie die geometrische Stetigkeit vorstellten [BD89]. In einer Anschlussarbeit haben sie Konzepte erarbeitet, mit denen es möglich ist geometrisch stetige Bezier-Splines zu erzeugen [BD90]. Eines der Konzepte wollen wir hier vorstellen: die Erzeugung geometrisch stetiger Bezier-Splines mit Hilfe von *Formparametern*. Wir wollen sie *Beta-Bezier-Splines* nennen. Dazu müssen wir zunächst zwei Bedingungen einführen, die Barsky und DeRose herleiteten, um eine geometrische Stetigkeit sicherzustellen.

2.10. Satz. *Es seien Q und R zwei reguläre C^2-stetige Kurven. Des weiteren seien $\beta_1 \in \mathbb{R}_+$ und $\beta_2 \in \mathbb{R}$. Wir nennen diese β_1 und β_2 die* Formparameter. *Q geht genau dann G^2-stetig in R über, wenn gilt*

$$\begin{aligned} R(0) &= Q(1), \\ R'(0) &= \beta_1 Q'(1), \\ R''(0) &= \beta_1^2 Q''(1) + \beta_2 Q'(1). \end{aligned}$$

Zum Beweis sei auf die Arbeit von Barsky und DeRose verwiesen. Welche Auswirkungen diese Bedingungen auf die Konstruktion einer G^2-stetigen Fortsetzung eines Bezier-Splines haben, lässt sich gut graphisch vermitteln.

2.5.1. Farin-Böhm Konstruktion

Nehmen wir uns eine reguläre Bezier-Kurve Q, mit den Bezier-Punkte $v_0, \ldots, v_3$ her. Wir wollen sie nach den in Satz 2.10 definierten Bedingungen G^2-stetig mit einer regulären Bezier-Kurve R fortsetzen. Die Bezier-Punkte von R seien $w_0, \ldots, w_3$. Dann gilt:

$$\gamma = \frac{2 + 2\beta_1}{\beta_2 + 2(\beta_1 + \beta_1^2)} \tag{2.11}$$

$$t = v_2 + \beta_1^2 \gamma (v_2 - v_1) \tag{2.12}$$

$$w_0 = v_3 \tag{2.13}$$

$$w_1 = w_0 + \beta_1 (v_3 - v_2) \tag{2.14}$$

$$w_2 = w_1 + \frac{1}{\gamma}(w_1 - t) \tag{2.15}$$

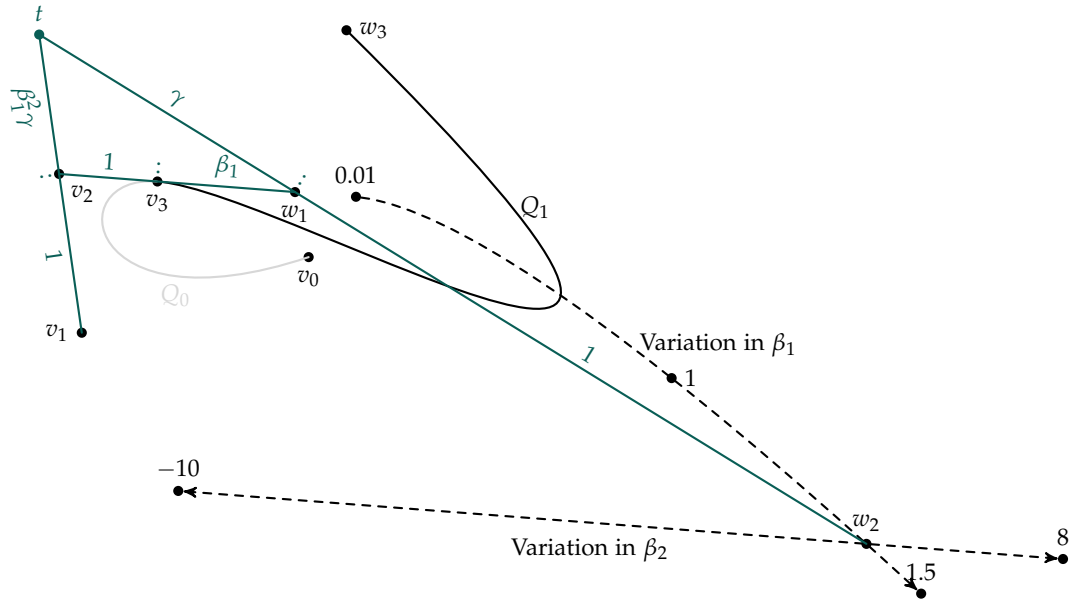


Abbildung 2.9. Die Farin-Böhm Konstruktion, hier mit $\beta_1 = 1.4$ und $\beta_2 = 4.0$. Gestrichelt ist der Verlauf der Position von w_2 bei Variation eines der Formparameter eingezeichnet.

Abbildung 2.9 zeigt das Ergebnis dieser Konstruktion. Mit den Beta-Beziers haben wir also eine Konstruktion eines G^2-stetigen Bezier-Splines. Die Formparameter sind unsere gewonnenen Freiheitsgrade gegenüber eines C^2-stetigen Bezier-Splines.

2.5.2. Eigenschaften von Beta-Bezier-Splines

Durch die Abschwächung der Stetigkeitsanforderung auf geometrische Stetigkeit, gewinnen wir einige Freiheitsgrade zurück. Trotzdem unterliegen auch die Beta-Bezier-Splines gewissen Einschränkungen, die in der Notation von Abbildung 2.9 folgende sind:

1. w_1 liegt stets auf der Fortsetzung des Vektors $\overrightarrow{v_2v_3}$ und es gilt $w_1 \neq v_3$, da $\beta_1 > 0$ gefordert ist.
2. w_2 bewegt sich auf einer Geraden parallel zu $\overline{v_2v_3}$, wenn wir β_2 variieren.
3. w_2 liegt immer gemeinsam mit v_1 auf einer Seite der Geraden $\overline{v_2v_3}$, da bei negativem β_2 sowohl γ als auch $\beta_1^2\gamma$ negativ werden.
4. Die Kurve Q_1 hat zu Beginn die selbe Krümmungsrichtung (links oder rechts), wie Q_0 zum Ende. Dies folgt direkt aus Punkt 3.

2.5.3. Beta-Beziers mit festem Zielwinkel

Die zwei Formparameter scheinen in ihrer Handhabung nicht intuitiv. Wie müssen wir β_1 oder β_2 verändern, um ein bestimmtes Verhalten der Kurve zu erzeugen? Dies scheint

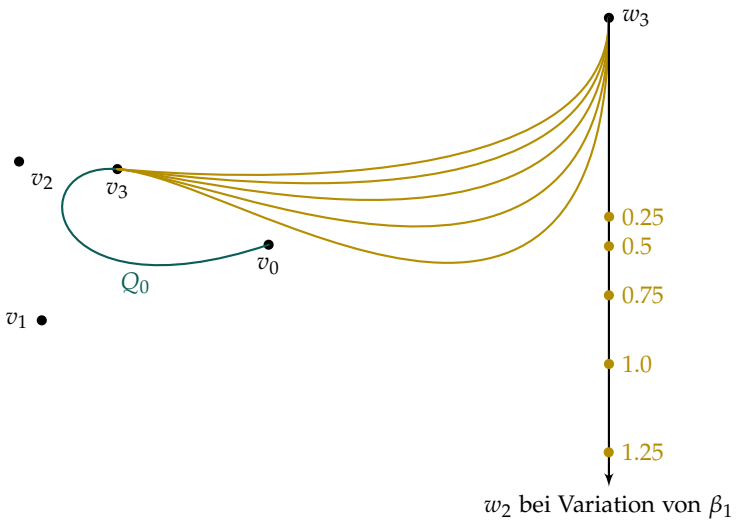


Abbildung 2.10. Ein Beta-Bezier-Spline mit nur einem Formparameter. Der Zielwinkel in w_3 ist mit 315° definiert.

schwer abschätzbar. Doch welche Eigenschaften der Kurve wären wünschenswert? Führen wir uns noch einmal vor Augen, zu welchem Zweck wir uns Splines ansehen: Wir wollen ein Kantenrouting mit ihnen erzeugen. Nach der Kurve Q_1 in Abbildung 2.9 kommt aller Wahrscheinlichkeit nach ein weiterer Punkt, der mit einer anschließenden Kurve erreicht werden muss. Somit wäre es hilfreich, wenn die Kurven Q_1 bereits eine sinnvolle Richtung in w_3 hätte. Also versuchen wir die Formel der Beta-Bezier-Kurven um eine Richtung im Zielpunkt zu erweitern.

Es sei α der Winkel in Bogenmaß, in dem wir w_3 erreichen wollen. Erinnern wir uns an Abbildung 1.2, so kommen wir auf folgende Bedingung:

$$w_2 = w_3 - r \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix} \qquad \text{mit } \alpha \in [0, 2\pi[\text{ und } r \in \mathbb{R}_+ \tag{2.16}$$

Dies in die Formeln 2.11 bis 2.15 eingesetzt ermöglicht es uns β_2 in Abhängigkeit von β_1, v_0, v_1, v_2, v_3, w_3 und α zu berechnen. Leider ist die Formel recht lang, weshalb sie hier nicht abgedruckt werden soll. Sie ist jedoch effizient berechenbar. Somit haben wir nun nur noch einen Formparameter. Abbildung 2.10 zeigt das Verhalten der Fortsetzung bei Variation dieses Parameters.

2.5.4. Anomalien bei Beta-Bezierkurven mit festem Zielwinkel

Wir sind nun in der Lage G^2-stetige Bezier-Splines zu erstellen. Vor allem in der Version mit festem Zielwinkel versprechen sie einen hohen praktischen Nutzen. Es stellen sich in der Anwendung jedoch noch einige Schwierigkeiten. Noch immer stellt sich die Frage wie wir den (nun einzigen) Formparameter wählen. Im Folgenden wollen wir uns daher

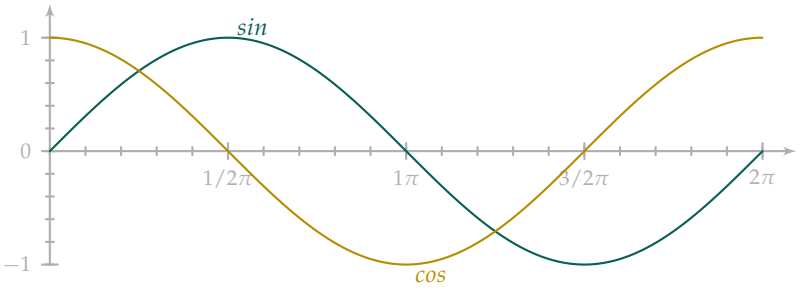


Abbildung 2.11. Die Sinus und Kosinus Funktionen.

mit den Schwierigkeiten befassen, die bei der Erzeugung einer G^2-stetigen Fortsetzung einer Bezierkurve mittels der Beta-Bezierkurven auftreten können.

Richtung im Zielpunkt

Durch Gleichung 2.16 stellen wir sicher, dass die Beta-Bezier-Fortsetzung die gewünschte Richtung in w_3 hat. In Abschnitt 2.5.3 wurde festgestellt, dass unsere Formel zur Konstruktion des β_2 von den Variablen β_1, v_0, v_1, v_2, v_3, w_3 und α abhängig ist. Das in Abschnitt 2.5.3 auf $\mathbb{R}$ eingeschränkte r ist also keine unserer freien Variablen. Falls $r < 0$ gilt würden wir w_3 in genau entgegengesetzter Richtung erreichen. Wir müssen also $r > 0$ sicherstellen, wobei

$$w_2 = w_3 - r \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix} \text{ und somit}$$

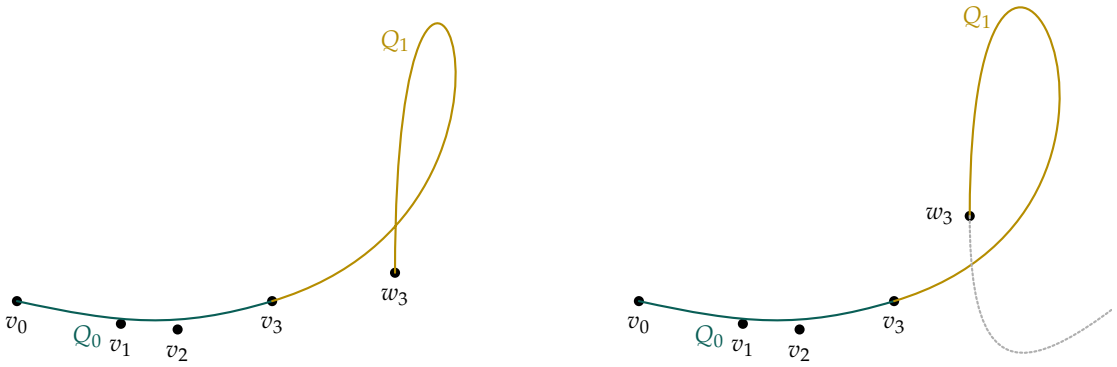
$$r = \frac{w_3^y - w_2^y}{\sin\alpha} = \frac{w_3^x - w_2^x}{\cos\alpha}.$$

Hierbei benennt w^x die x-Koordinate und w^y die y-Koordinate von w. Betrachten wir die Sinus- und Kosinus-Funktionen in Abbildung 2.11, so müssen wir für $\alpha \in [0, 2\pi[$ folgende Fallunterscheidung vornehmen:

$$\sin\alpha = \begin{cases} 0 & \text{falls } \alpha \in \{0, \pi\} \\ < 0 & \text{falls } \alpha \in]\pi, 2\pi[\\ > 0 & \text{falls } \alpha \in]0, \pi[\end{cases} \qquad \cos\alpha = \begin{cases} 0 & \text{falls } \alpha \in \{\frac{1}{2}\pi, \frac{3}{2}\pi\} \\ < 0 & \text{falls } \alpha \in]\frac{1}{2}\pi, \frac{3}{2}\pi[\\ > 0 & \text{sonst} \end{cases}$$

Fall 1: $\alpha \in \{0, \pi\}$
Dann gilt: $\cos\alpha > 0$.
Es muss gelten: $w_2^x < w_3^x$.

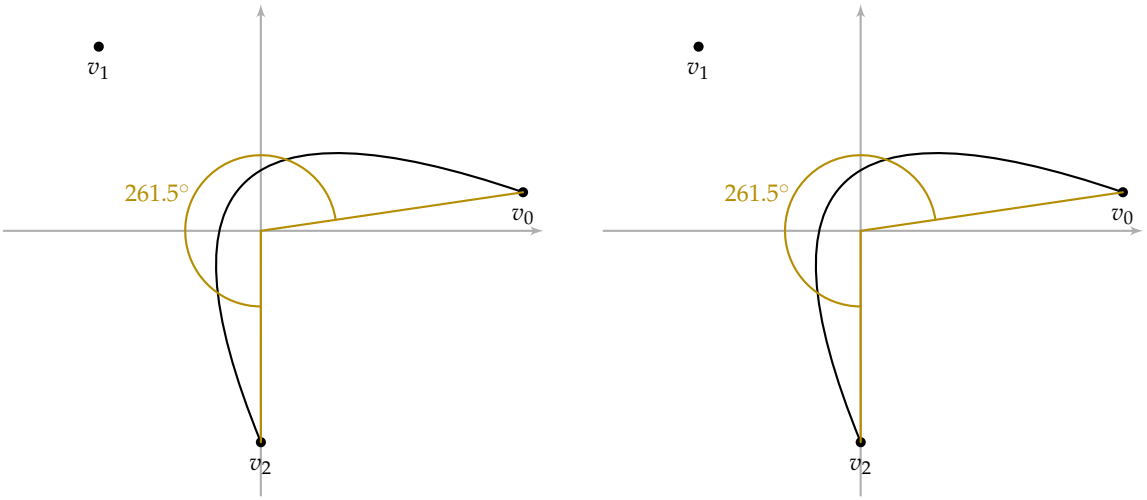
Fall 2: $\alpha \in]0, \pi[$
Dann gilt: $\sin\alpha > 0$.
Es muss gelten: $w_2^y < w_3^y$.



(a) Überschneidung einer Bezierkurve mit sich selbst.

(b) Überschneidung zweier aufeinander folgenden Bezierkurven.

Abbildung 2.12. Zwei Beta-Bezier-Splines die zur Schleifenbildung führen.



(a) Erste Ableitung von Q_1 aus Abbildung 2.12a.

(b) Erste Ableitung von Q_1 aus Abbildung 2.12b.

Abbildung 2.13. Ableitung der sich Schleifen bildenden Beta-Bezier-Splines aus Abbildung 2.12

Fall 3: $\alpha \in]\pi, 2\pi[$
Dann gilt: $\sin\alpha < 0$
Es muss gelten: $w_2^y > w_3^y$.

Die Position von w_2 können wir direkt mit β_1 beeinflussen.

Schleifenbildung

Es ist möglich, dass der Beta-Bezier-Spline eine Schleife bildet. Abbildung 2.12 zeigt die zwei Varianten, die auftreten können. In der ersten Variante kreuzt eine Beta-Bezier-Kurve ihren eigenen Pfad, in der zweiten kommt es zu einer Überschneidung zweier aufeinander folgenden Kurven. Eine Bezierkurve bildet dann eine Schleife, wenn ihre Richtung einen Bereich von mehr als 180° abdeckt. Wie in Abbildung 2.13 zu sehen, benötigen wir in unserem kubischen Fall die Richtung der drei Kontrollpunkte der ersten Ableitung. Falls wir eine Schleife feststellen, so müssen wir den Formparameter verringern. In dem Fall, der in Abbildung 2.12a dargestellt ist, führt dies jedoch dazu, dass w_3 in der falschen Richtung erreicht wird. Das Auflösen einer Anomalie führt also zur Erzeugung einer anderen. Diese Situation können wir nur dadurch auflösen, dass wir die Kurve Q_1 in zwei Bezierkurven aufteilen.

2.6. B-Spline

Bei den *B-Splines* (oder *Basis-Splines*) handelt es sich um eine Verallgemeinerung der Bezier-Splines. Genau wie die Bezier-Splines werden B-Splines durch eine Menge von Kontrollpunkten definiert.

23. Definition ((uniformer)B-Spline, Knotenvektor, de-Boor-Punkte)**.** Ein *B-Spline* $Q(t)$ vom Grad n besteht aus m Kontrollpunkten $v_1, \ldots, v_m$ und einem *Knotenvektor* $\tau = (\tau_0, \ldots, \tau_{m+n})$. Die Knoten des Knotenvektors sind aufsteigend sortiert. Der Knotenvektor eines B-Splines ist *uniform*. In einem uniformen Knotenvektor ist die Differenz zweier benachbarter Knoten entweder null, oder gleich einer Konstanten, die für den gesamten Knotenvektor gilt. Die Kontrollpunkte werden auch *de-Boor-Punkte* genannt und bilden das Kontrollpolygon. Die Kurve Q errechnet sich aus der Formel

$$Q(t) = \sum_{i=1}^{m} \mathcal{N}_{i,n}(t) v_i \quad \text{mit } t \in [\tau_n, \tau_{m+1}].$$

Hierbei sind $\mathcal{N}_{i,n}$ die Basisfunktionen der B-Splines. Sie sind rekursiv definiert als

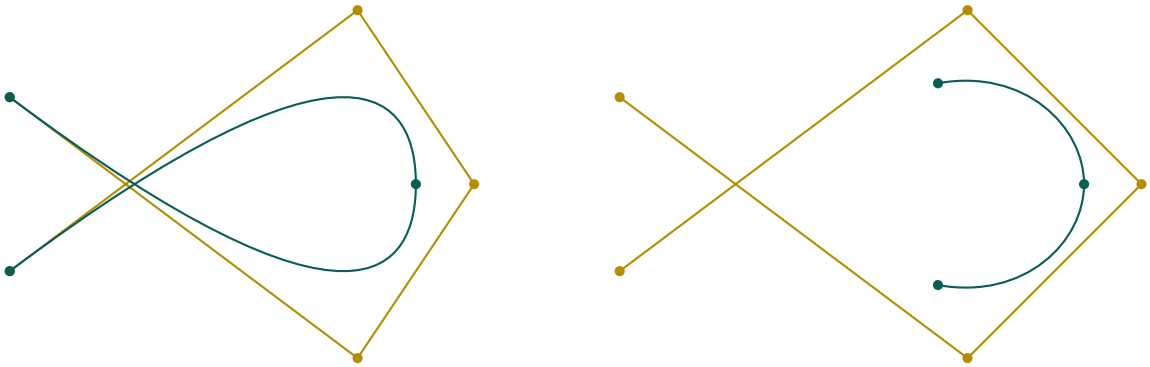
$$\mathcal{N}_{i,1}(t) = \begin{cases} 1 & \text{falls } t \in [\tau_i, \tau_{i+1}[\\ 0 & \text{sonst} \end{cases}$$

$$\mathcal{N}_{i,k}(t) = \frac{t - \tau_i}{\tau_{i+k} - \tau_i} \mathcal{N}_{i,k-1}(t) + \frac{\tau_{i+k+1} - t}{\tau_{i+k+1} - \tau_{i+1}} \mathcal{N}_{i+1,k-1}(t).$$

24. Definition (Multiplizität von Knoten)**.** Als *Multiplizität* eines Knotens bezeichnet man die Häufigkeit seines Vorkommens im Knotenvektor. Einen Knoten mit der Multiplizität eins bezeichnen wir als *einfach*.

Wieder wollen wir einige wichtige Eigenschaften eines B-Splines vom Grad n nennen:

- Der B-Spline muss mindestens $n + 1$ Kontrollpunkte besitzen.



(a) Geklemmt. $\tau = \left(0, 0, 0, \frac{1}{2}, 1, 1, 1\right)$

(b) Ungeklemmt. $\tau = \left(\frac{1}{8}, \frac{2}{8}, \frac{3}{8}, \frac{4}{8}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}\right)$

Abbildung 2.14. Zwei B-Splines mit identischen Kontrollpunkten. In der geklemmten Variante verläuft die Kurve von $t = 0$ bis $t = 1$. In der ungeklemmten Version verläuft sie von $t = \frac{3}{8}$ bis $t = \frac{5}{8}$. Außerhalb dieses Intervalls ist die Kurve undefiniert. Der Knotenvektor entspricht bereits der Konvention 2.17.

- Es gilt stets $\mathcal{N}_{i,k}(t) \geqslant 0$.
- Die Basisfunktionen bilden für $t \in [\tau_0, \tau_{m+n}]$ eine Zerlegung der Eins.
- Es gilt $\mathcal{N}_{i,\,k}(t) = 0$, falls $t \notin [\tau_i, \tau_{i+k+1}[$. Diese Eigenschaft wird auch *lokaler Träger* genannt. Dadurch gewinnen wir die lokale Kontrolle, da jeder Punkt auf der Kurve nur von einer begrenzten Menge an Kontrollpunkten beeinflusst wird.
- Sind alle Knoten des Knotenvektors einfach, so sind die ersten $n - 1$ Ableitungen der Kurve stetig, sie ist also C^{n-1}-stetig.
- Wird ein Knoten des Knotenvektors wiederholt, so reduziert sich die Stetigkeit für jede Wiederholung um eins.
- Ohne Knotenwiederholungen wird keiner der Kontrollpunkte durchlaufen.
- Gilt $\tau_i = \tau_{i+1} = \ldots = \tau_{i+n-1}$ mit $i \in \{0, \ldots, m-1\}$, so durchläuft die Kurve den Kontrollpunkt v_{i-1}.
- Wird der erste und letzte Knoten des Knotenvektors $n + 1$ mal wiederholt, so beginnt die Kurve im ersten, und endet im letzten Kontrollpunkt. Einen solchen B-Spline nennen wir *geklemmt* (engl. *clamped*).

An dieser Stelle schlägt die etwas unverständliche Notation des Knotenvektors zu. Es ist nicht ersichtlich, warum die Anfangs- und Endknoten eines geklemmter Knotenvektor die Multiplizität $n + 1$ haben muss. n Knoten würden für die eindeutige Darstellung genügen. Ebenso ließe sich die Anzahl an Knoten ohne Informationsverlust auf $m + n - 1$ reduzieren. Auch in der Literatur lässt sich hierfür keine Begründung finden, außer der Aussage „For historical reasons, knot vectors are traditionally described as requiring n

end-condition knots, and in the real world you will always find a meaningless knot at the beginning and end of a knot vector.“ [Sed05]

2.17. Konvention. Für eine einfachere Schreibweise verzichten wir im Folgenden auf die Nennung dieser bedeutungslosen Knoten. Unser Knotenvektor erstreckt sich im Folgenden also auf $(\tau_1, \ldots, \tau_{m+n-1})$.

Abbildung 2.14 zeigt einen geklemmten und einen ungeklemmten B-Spline mit der vereinfachten Notation für den Knotenvektor.

Die Tatsache, dass B-Splines auf einfache Art C^{n-1}-stetig konstruiert werden können, ist ihr entscheidende Vorteil gegenüber den Bezier-Splines. Ihr Nachteil ist, dass keiner der inneren Kontrollpunkte durchlaufen wird, wenn wir die Stetigkeit nicht aufgeben wollen. Wollen wir eine Kante durch mehrere Dummy-Knoten führen, so müssen wir auf andere Art sicherstellen, dass sie auch wirklich durch die Dummy-Knoten läuft.

2.6.1. Ableitung von B-Splines

Analog zu Satz 2.3, lässt sich auch die Ableitung eines B-Splines auf die Ableitung seiner Basisfunktion zurückführen.

2.18. Satz. *Es sei $Q(t)$ ein B-Spline vom Grad n, seine Kontrollpunkte seien $v_1, \ldots, v_m$ und $\tau = (\tau_1, \ldots, \tau_{m+n-1})$ sein Knotenvektor. Die erste Ableitung von Q lautet dann*

$$Q'(t) = \sum_{i=1}^{m-1} \mathcal{N}_{i,n-1}(t) w_i \qquad \textit{mit}$$
$$w_i = \frac{n}{\tau_{i+n} - \tau_i}(v_{i+1} - v_i) \quad i \in \{1, \ldots, m-1\}$$

Q' ist dann ein B-Spline vom Grad $n-1$mit Knotenvektor $(\tau_2, \ldots, \tau_{m+n-3})$ und Kontrollpunkten $w_1, \ldots, w_{m-1}$.

Beweis. Der Beweis verläuft analog zum Beweis von Satz 2.3. Wie bei den Bezierkurven sind die Kontrollpunkte unabhängig von t, womit nur die Ableitung der Basisfunktion zu errechnen bleibt. Sie lautet

$$\frac{d}{dt}\mathcal{N}_{i,n}(t) = \mathcal{N}'_{i,n}(t) = \frac{n}{\tau_{i+n} - \tau_i}\mathcal{N}_{i,n-1}(t) - \frac{n}{\tau_{i+n+1} - \tau_{i+1}}\mathcal{N}_{i+1,n-1}(t)$$

Auch bei den B-Splines errechnet sich die Ableitung eines Splines vom Grad m aus der, diesmal gewichteten, Differenz zweier B-Splines vom Grad $m-1$. Rechnen wir nun die Gewichtung der beiden Basen direkt auf die Kontrollpunkte, so erhalten wir die Gleichung aus der Behauptung. □

2.6.2. Extrema von B-Splines

Eine weitere Herausforderung, bei der Arbeit mit Splines ist es ihre Extrema zu berechnen. Das Vorgehen ist hier das selbe wie bei gewöhnlichen Funktionen: Wir müssen die erste

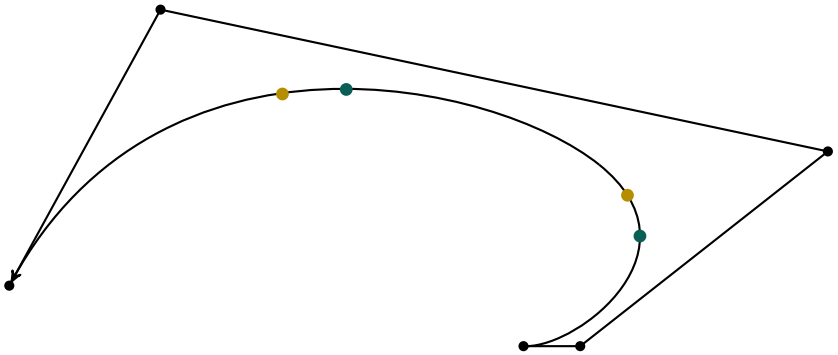


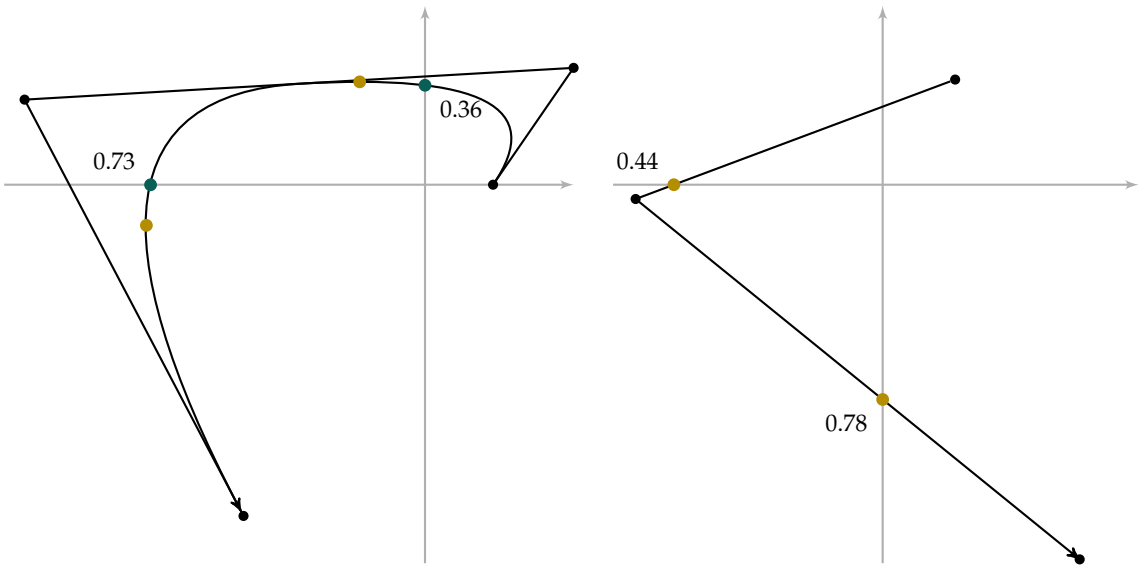
Abbildung 2.15. Gesucht sind die Maximalwerte des Splines in x- und y-Richtung. Eingezeichnet sind die Nullstellen aus der ersten und zweiten Ableitung.

Ableitung gleich null setzen. Hierbei ist zwischen Extrema auf der x- und der y-Achse zu unterscheiden. Wir werden in dieser Arbeit vor allem kubische nutzen. Nehmen wir uns also als Beispiel einen kubischen B-Spline (siehe Abbildung 2.15) und suchen seine Nullstellen.

Wie wir Ableitungen von B-Splines berechnen haben wir im letzten Kapitel gesehen. Die eigentliche Herausforderung bei der Berechnung von Extrema liegt in der Berechnung der Nullstellen. Diese Aufgabe ist im Allgemeinen nicht einfach zu lösen. Für Funktionen bietet sich das *Newton-Verfahren* an. Für Kurven ist es jedoch nicht geeignet, alleine schon da wir nicht wissen, wie wir einen Spline durch einen anderen dividieren. Für unseren Spezialfall, dem kubische Spline, lässt sich eine Annäherung durch Berechnung der Nullstellen der zweiten Ableitung sehr schnell berechnen. Wir berechnen somit nicht das eigentliche Extremum, also die Stelle im Spline, an der beispielsweise der x-Wert unverändert bleibt. Statt dessen berechnen wir die Stelle, an die Veränderung der Geschwindigkeit der Bewegung in y-Richtung gleich null ist. Die zweite Ableitung besteht in unserem kubischen Fall aus einem Polynomenzug, wie in Abbildung 2.16b zu sehen. Hier ist es einfach die Nullstellen zu finden. Jedoch ist sowohl anhand der ersten Ableitung (Abbildung 2.16b), als auch an der Originalkurve die Abweichung von der tatsächlichen Lösung zu erkennen.

Ein für Kurven geeignetes Verfahren für die beliebig genaue Berechnung von Nullstellen geht auf Mørken und Reimers zurück [MR07]. Nach ihrem Verfahren ist zunächst die Nullstelle des Kontrollpolygons zu finden. Der Fortschritt t auf dem Polynomenzug wird dann als neuer Knoten in den Knotenvektor eingefügt. Diese zwei Schritte werden bis zu einer beliebigen Genauigkeit wiederholt. In ihrer Arbeit ist auch ein Konvergenzbeweis zu finden.

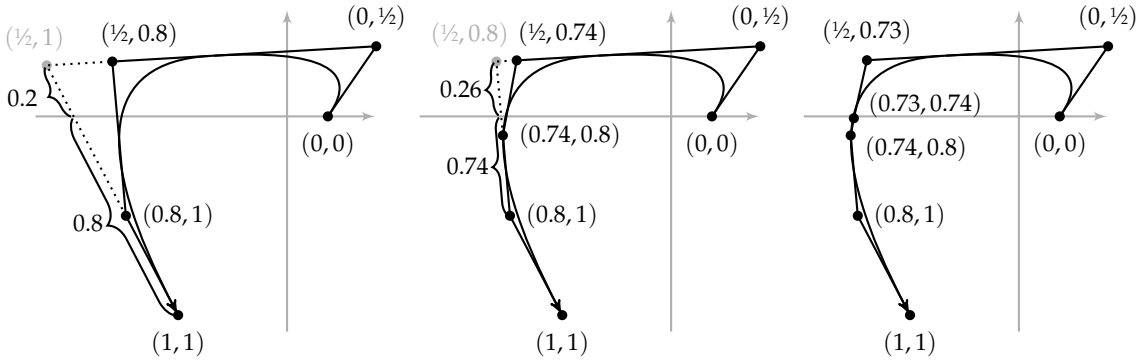
Um die einzufügenden Werte für t zu berechnen, greifen wir wieder auf die Notation in Polarform zurück. Betrachten wir Abbildung 2.17a, wo die Einfügung des ersten Knotens dargestellt ist. Die y-Nullstelle des Kontrollpolygons liegt zwischen den Punkten $(½, 1)$ und $(1, 1)$. Mit Hilfe des Strahlensatzes lässt sich die genaue Position auf dieser Strecke berechnen. Gewichtet mit diesen Faktor (der Fortschritt auf der Strecke) bilden



(a) Erste Ableitung, ein quadratischer B-Spline

(b) Zweite Ableitung, ein Polynomenzug

Abbildung 2.16. Die ersten beiden Ableitungen des B-Splines aus Abbildung 2.15



(a) $\tau = \{0, 0, ½, 0.8, 1, 1\}$

(b) $\tau = \{0, 0, ½, 0.74, 8, 1, 1\}$

(c) $\tau = \{0, 0, ½, 0.73, 0.74, 8, 1, 1\}$

Abbildung 2.17. Berechnung der y-Nullstelle des B-Splines aus Abbildung 2.16a

wir den Durchschnitt aller beteiligten Knoten. Also

$$rel = \frac{v(½, 1)_x}{v(½, 1)_x - v(1, 1)_x} = 0.20436$$
$$\frac{rel \cdot (½ + 1) + (1 - rel) \cdot (1 + 1)}{4} = 0.80109.$$

Dies ist der erste Knoten den wir einfügen. Dadurch erhalten wir eine Verfeinerung des

Kontrollpolygons und beginnen von vorn, wie in Abbildung 2.17 zu sehen.

Eine denkbare Erweiterung dieses Algorithmus ist es, den errechneten Punkt zwei mal einzufügen. Eine zweite Einfügung des selben Knotens ist immer weniger rechenaufwändig als die erste, da stets ein Kontrollpunkt weniger zu berechnen ist. Für die Berechnung der Extremwerte unseres Kubischen Splines ergäbe sich der zweite Vorteil, die erste Ableitung ein quadratischer Spline ist, und nach zweimaligem Einfügen eines Knotens k ein Kontrollpunkt (k, k) entstehen würde, der auf dem Spline liegt.

2.6.3. Umrechnung B-Splines in Bezier-Splines

In diesem Kapitel wollen wir zeigen, wie es möglich ist, einen B-Spline in einen Bezier-Spline umzurechnen. Zum einfacheren Verständnis führen wir zunächst eine neue Notation ein.

25. Definition (Polarform von Kontrollpunkten)**.** Sei $Q(t)$ ein B-Spline vom Grad n mit Kontrollpunkten $V = \{v_1, \ldots, v_m\}$ und Knotenvektor $\tau = (\tau_1, \ldots, \tau_{m+n-1})$. Die Benennung des Kontrollpunktes $v_i \in V$ in *Polarform* lautet $v(\tau_i, \ldots, \tau_{i+n-1})$ [Ram89].

2.19. Korollar. *Die Polarform ist symmetrisch. Dies bedeutet, dass die Reihenfolge der Argumente beliebig ist. Sind zwei Kontrollpunkte $v_1 = v(\tau_1, \ldots, \tau_n, a)$ und $v_2 = v(\tau_1, \ldots, \tau_n, b)$ gegeben, so kann ein dritter Kontrollpunkt $v_3 = v(\tau_1, \ldots, \tau_n, c)$ mit $c \in [a, b]$ nach folgender Formel berechnet werden:*

$$v_3 = \frac{(b-c)v_1 + (c-a)v_2}{b-a}$$

(Siehe hierzu auch [Sed05].)

In Unterkapitel 2.6 haben wir festgehalten, dass die B-Splines eine Verallgemeinerung der Bezier-Splines sind. Also lässt sich jede Bezier-Kurve auch als B-Spline darstellen.

2.20. Korollar. *Eine Bezierkurve vom Grad n mit Kontrollpunkten $v_0, \ldots, v_n$ ist ein B-Spline selben Grades, welcher geklemmt ist. Sein Knotenvektor lautet $\tau = (0_1, \ldots, 0_n, 1_1, \ldots, 1_n)$.*

Dies lässt sich durch einfaches Nachrechnen der Basen überprüfen, worauf wir an dieser Stelle verzichten wollen. Aufbauend auf Korollar 2.20 lässt sich nun folgender Satz konstruieren.

2.21. Satz. *Es sei Q ein B-Spline vom Grad n, dessen Knotenvektor $k+1$ eindeutige Knoten enthält, die jeweils eine Multiplizität von n haben. Dann können wir einen Bezier-Spline, bestehend aus k Bezierkurven konstruieren, der Q überdeckt. (Siehe Abbildung 2.18)*

Beweis. Der Knotenvektor τ von Q hat folgende Form:

$$\tau = (\overbrace{\tau_0, \ldots, \tau_0}^{n\text{-mal}}, \overbrace{\tau_1, \ldots, \tau_1}^{n\text{-mal}}, \ldots, \overbrace{\tau_k, \ldots, \tau_k}^{n\text{-mal}}).$$

Q hat dann $kn + 1$ Kontrollpunkte. Wir benennen sie als $v_0, \ldots, v_{kn}$.

$$v_0 = v(\overbrace{\tau_0, \ldots, \tau_0}^{n\text{-mal}}), \qquad v_1 = v(\overbrace{\tau_0, \ldots, \tau_0}^{(n-1)\text{-mal}}, \tau_1), \ \ldots \ ,$$

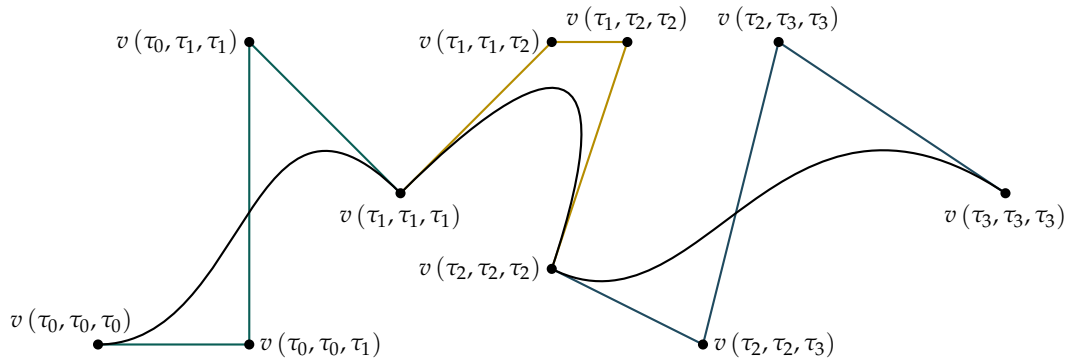


Abbildung 2.18. Ein kubischer B-Spline mit Knotenvektor $\tau = (\tau_0,\tau_0,\tau_0,\tau_1,\tau_1,\tau_1,\tau_2,\tau_2,\tau_2,\tau_3,\tau_3,\tau_3)$. Da die Multiplizität seiner inneren Knoten gleich seiner Dimension ist, hat er an den Stellen $v(\tau_1,\tau_1,\tau_1)$ und $v(\tau_2,\tau_2,\tau_2)$ nur eine C^0-Stetigkeit. Wegen der Struktur seines Knotenvektors entspricht er drei Bezier-Splines, die seine Kontrollpunkte teilen.

$$v_n = v(\overbrace{\tau_1,\dots,\tau_1}^{n\text{-mal}}), \qquad v_{n+1} = v(\overbrace{\tau_1,\dots,\tau_1}^{(n-1)\text{-mal}},\tau_1),\ \dots\ ,$$
$$v_{kn-1} = v(\tau_{k-1},\overbrace{\tau_k,\dots,\tau_k}^{(n-1)\text{-mal}}), \quad v_{kn} = v(\overbrace{\tau_k,\dots,\tau_k}^{n\text{-mal}})$$

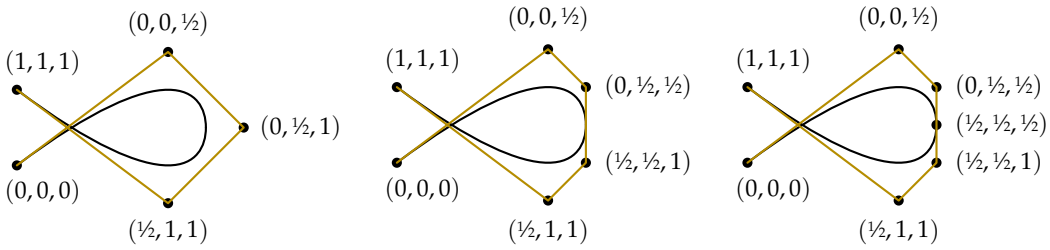
Wir könnten nun k B-Splines folgendermaßen definieren:

$$\forall i \in \{1,\dots,k\}\colon \text{ B-Spline } R_i(t) \text{ mit } t \in [\tau_{i-1},\tau_i] \text{ und}$$
$$\text{Knotenvektor } \tau = (\overbrace{\tau_{i-1},\dots,\tau_{i-1}}^{n\text{-mal}},\overbrace{\tau_i,\dots,\tau_i}^{n\text{-mal}})$$
$$\text{Kontrollpunkten } V = \left\{v_{(i-1)n},\dots,v_{in}\right\}.$$

Die B-Splines R_i überdecken Q vollständig. Zudem lässt sich für jeden Spline R_i eine Reparametrisierung finden, so dass sein Knotenvektor gerade $(0_1,\dots,0_n,1_1,1_n)$ ist. Somit ist er nach Korollar 2.20 auch ein Bezier-Spline. □

Wir können also aus einem B-Spline, dessen Knoten alle eine Multiplizität gleich seines Grades aufweise, auf direktem Weg einen überdeckenden Bezier-Spline ableiten. Bleibt die Frage, ob wir auch für einen beliebigen B-Spline einen äquvalenten Bezier-Spline finden können, also ob wir in den Knotenvektor eines B-Splines weitere Knoten einfügen können, ohne die resultierende Kurve zu verändern. Denn könnten wir die Multiplizität aller Knoten auf die Dimension des Splines erhöhen, so hätten wir nach Satz 2.21 einen überdeckenden Bezier-Spline gefunden.

Für diese Operation gibt es verschiedene Algorithmen. So fügt *Böhms Algorithmus* beispielsweise einen einzigen Knoten ein [Boe80]. Der *Oslo-Algorithmus* ermöglicht das Einfügen mehrerer Knoten gleichzeitig [CLR80], ist jedoch auch dementsprechend komplexer. Für diese Arbeit ist Böhms Algorithmus ausreichend.



(a) $\tau = \{0, 0, 0, ½, 1, 1, 1\}$ **(b)** $\tau = \{0, 0, 0, ½, ½, 1, 1, 1\}$ **(c)** $\tau = \{0, 0, 0, ½, ½, ½, 1, 1, 1\}$

Abbildung 2.19. Darstellung von Böhms Algorithmus.

Böhms Algorithmus

Wir werden nun am Beispiel der Kurve von Abbildung 2.14b die Umrechnung von B-Splines in Bezier-Splines auf Basis von Böhms Algorithmus vornehmen. Wir haben einen B-Spline $Q(t)$ mit fünf Kontrollpunkten und dem Knotenvektor $\tau = \{0, 0, 0, ½, 1, 1, 1\}$ gegeben. Nach Korollar 2.20 müssen wir den B-Spline $Q'(t)$ errechnen, der den Knotenvektor $\tau' = (0, 0, 0, ½, ½, ½, 1, 1, 1)$ hat. Hierzu ist zwei mal der Knoten ½ einzufügen.

Erste Einfügung

Alt

$\tau =$ $(0, 0, 0, ½, 1, 1, 1)$

$V =$ $(0, 0, 0)$ $(0, 0, ½)$ $(0, ½, 1)$ $(½, 1, 1)$ $(1, 1, 1)$

Neu

$\tau =$ $(0, 0, 0, ½, ½, 1, 1, 1)$

$V =$ $(0, 0, 0)$ $(0, 0, ½)$ $(0, ½, ½)$ $(½, ½, 1)$ $(½, 1, 1)$ $(1, 1, 1)$

Berechnung

$(0, ½, ½)$ aus $v\,(0, 0, ½)$ und $v\,(0, ½, 1)$ $[a = 0,\ b = 1,\ c = ½]$

$(½, ½, 1)$ aus $v\,(0, ½, 1)$ und $v\,(½, 1, 1)$ $[a = 0,\ b = 1,\ c = ½]$

$$(0, ½, ½) = \frac{(1 - ½) \cdot v\,(0, 0, ½) + (½ - 0) \cdot v\,(0, ½, 1)}{1 - 0} = \frac{v\,(0, 0, ½) + v\,(0, ½, 1)}{2}$$

$$(½, ½, 1) = \frac{(1 - ½) \cdot v\,(0, ½, 1) + (½ - 0) \cdot v\,(½, ½, 1)}{1 - 0} = \frac{v\,(0, ½, 1) + v\,(½, ½, 1)}{2}$$

Zweite Einfügung

Alt						
$\tau =$	$(0,0,0,½,½,1,1,1)$					
$V =$	$(0,0,0)$	$(0,0,½)$	$(0,½,½)$	$(½,½,1)$	$(½,1,1)$	$(1,1,1)$
Neu						
$\tau =$	$(0,0,0,½,½,½,1,1,1)$					
$V =$	$(0,0,0)$	$(0,0,½)$	$(0,½,½)$	$(½,½,½)$	$(½,½,1)$	$(½,1,1)$	$(1,1,1)$

Berechnung

$(½,½,½)$ aus $v\,(0,½,½)$ und $v\,(½,½,1)$ $[a=0,\, b=1,\, c=½]$

$$(½,½,½) = \frac{v\,(0,½,½) + v\,(½,1,1)}{2}$$

Mit Böhms Algorithmus kann man einen beliebiger Punkt auf dem B-Spline berechnen. Hierzu kann alternativ auch der *de Boor-Algorithmus*, eine Verallgemeinerung des de Casteljau Algorithmus (siehe Abschnitt 2.4.2) genutzt werden. Er ist nach dem deutschen Mathematiker Carl-Wilhelm de Boor benannt, welcher auch Namensgeber für die alternative Benennung der Kontrollpunkte eines B-Splines ist.

2.7. Gebrochenrationale Splines

Mit den bisher vorgestellten Splines ist es nicht möglich, jede beliebige geometrische zweidimensionale Form darzustellen. So kann man beispielsweise mit den Splines keinen Kreis formen. Dies ist erst mit *gebrochenrationalen Splines* möglich [BP97].

26. Definition ((gebrochen-)rationale Splines)**.** Ein *gebrochenrationaler Spline* (engl. *rational*) ist eine Erweiterung seines *ganzrationalen* Verwandten. Beim gebrochenrationalen Spline werden die Kontrollpunkte *gewichtet*, wodurch sich weitere Freiheitsgrade bei der Konstruktion des Splines ergeben.

27. Definition (gebrochenrationale Bezierkurve)**.** Eine *gebrochenrationale Bezierkurve* $Q(t)$ des Grades n hat $n+1$ Kontrollpunkte, und zu jedem Kontrollpunkt v_i ein nicht negatives Gewicht w_i. Die Kurve ist dann definiert als

$$Q(t) = \sum_{i=0}^{n} \mathcal{R}^{\mathcal{B}}_{i,n}(t) v_i,$$

wobei $\mathcal{R}^{\mathcal{B}}_{i,n}$ nun die gewichteten Bernstein-Polynome sind:

$$\mathcal{R}^{\mathcal{B}}_{i,n}(t) = \frac{\mathcal{B}_{i,n}(t) w_i}{\sum_{j=0}^{n} \mathcal{B}_{j,n}(t) w_i}.$$

Nach demselben Prinzip lassen sich alle Splines zu gebrochenrationalen Varianten erweitern. Besondere Bedeutung fällt einer Erweiterung der in Unterkapitel 2.6 vorgestellten B-Splines zu.

28. Definition (NURBS)**.** *NURBS* sind eine gebrochenrationale Erweiterung der B-Splines. *Non-Uniform* bezieht sich auf den Knotenvektor. Die Knoten des Knotenvektors sind noch immer aufsteigend sortiert, jedoch kann die Differenz zwischen zwei benachbarten Knoten nun beliebig sein. Die Kurve $Q(t)$ eines NURBS des Grades n, versehen mit m Kontrollpunkten, einem Knotenvektor $\tau = (\tau_1, \ldots, \tau_{m+n-1})$ (in vereinfachter Notation) und Gewichten $w_i \geqslant 0$, $i \in \{0, \ldots, m-1\}$, hat für $t \in [\tau_1, \ldots, \tau_{m+n-1}]$ die Form

$$Q(t) = \sum_{i=0}^{m-1} R_{i,n}^{\mathcal{N}}(t) \quad \text{mit} \quad R_{i,n}^{\mathcal{N}}(t) = \frac{\mathcal{N}_{i,n}(t) w_i}{\sum_{j=1}^{k} \mathcal{N}_{j,n}(t) w_j}.$$

Die NURBS haben die selben Eigenschaften wie die B-Splines. Da sie gebrochenrationale Splines sind ist es mit ihnen nun möglich, beliebige zweidimensionale, geometrische Formen darzustellen.

2.22. Korollar. *Rationale B-Splines lassen sich analog zu unserem Vorgehen in Abschnitt 2.6.3 in rationale Bezier-Splines umrechnen. Hier muss der Gewichtsfaktor der Kontrollpunkte auf die selbe Art wie die Kontrollpunkte umgerechnet werden.*

Kapitel 3

Entwurf

Nachdem wir die theoretischen Grundlagen für diese Arbeit vorgestellt haben, kommen wir nun zum Entwurf unseres Kantenroutings mit Splines in ebenbasierten Graphen. Wir werden zunächst Kriterien bestimmen, nach denen wir die Splines gestalten wollen. Anschließend befassen wir uns mit den verschiedenen Kantenvarianten, die von uns behandelt werden. Wir werden die Basis schaffen, auf der wir in Kapitel 4 die konkrete Implementierung entwickeln.

3.1. Ästhetik

Neben die technische Anforderung, zwei Knoten um Hindernisse herum zu verbinden, stellen wir noch eine weitere Anforderung, die deutlich schwerer zu fassen ist. Der Anforderungskatalog, den wir in diesem Abschnitt erarbeiten, ist schwer in konkreten Dimensionen zu messen. Wir wollen definieren, wann ein Layout nicht nur korrekt, sonder auch „gut“ ist. Doch welche Kriterien gibt es, um „gut“ zu definieren? Hierzu gibt es eine Vielzahl von Untersuchungen, von denen im Folgenden eine kleine Auswahl genannt sei.

Kim Mariott et al. untersuchten in einer Arbeit, welche Charakteristiken eines Layouts gut in Erinnerung behalten werden [MPWG12]. Sie mutmaßen, dass ein Graph, dessen Layout diese einprägsamen Charakteristiken aufweist, ebenfalls besser in Erinnerung bleibt, und dessen Layout somit gut sein muss. Statistisch signifikant positive Eigenschaften waren Orthogonalität und Kollinearität von Kanten, sowie Symmetrie von Knoten. Eine Ausrichtung der Kanten in horizontaler- und vertikaler Richtung erwies sich, wie von ihnen erwartet, als weniger einprägsam. Erstaunt zeigten sich die Autoren, dass parallele Linienführung und eine auf virtuellen Geraden ausgerichtete Knotenpositionierung weniger zu einem einprägsamen Layout beitrugen.

Helen Purchase et al. untersuchten manuell erzeugte Layouts, und versuchten hieraus Rückschlüsse auf ein gutes Layout zu ziehen [PPP12]. Layoutkriterien, die den Versuchsteilnehmern bei ihren manuellen Entwürfen wichtig waren, könnten auch bei einem automatischen Layout zu einem guten Layout führen. Es zeigte sich, dass die Versuchsteilnehmer sich bemühten Kantenkreuzungen zu vermeiden. Im Gegensatz zu der Arbeit von Kim Mariott et al. stellten sie eine hohe Bedeutung von geraden, horizontal und vertikal ausgerichtete Kanten fest. Oft lag den erzeugten Zeichnungen ein virtuelles Gittermuster zu Grunde.

Colin Ware et al. maßen den Einfluss verschiedener Graphencharakteristika auf die Zeit, die ihre Versuchsteilnehmer für das Auffinden des kürzesten Pfades zwischen zwei Knoten benötigen [WPCM02]. Ein stark negativer Einfluss auf die Suchzeit lässt auf Graphencharakteristika schließen, die besser vermieden werden. Relevant bei der Suche nach dem kürzesten Pfad zeigten sich die Kantenlänge, die Anzahl der Kantenkreuzungen und die Kontinuität des zu findenden Pfades. Dies bedeutet auch, dass diese drei Variablen auf den Gesamtgraph bezogen wenig relevant waren. Durch ihren Ansatz lässt sich der Einfluss der einzelnen Aspekte direkt miteinander vergleichen. So fanden sie heraus, dass eine Krümmung von 100° auf dem Pfad $1,7$ Sekunden Suchzeit verursachten. Jede Kantenkreuzung erhöhte die Suchzeit um $0,65$ Sekunden. Dies impliziert, dass unter gewissen Umständen eine Kantenkreuzung in Kauf genommen werden kann, wenn sich dadurch die Krümmung der Kanten reduziert.

3.1.1. Wann ist ein Layout „gut“?

Die von uns erzeugten Graphen dienen der Informationsvermittlung. Somit ist das oberste Ziel, dass der Graph eine gute Lesbarkeit aufweist. Es muss für den Leser einfach zu erfassen sein, welche Knoten miteinander über welche Kanten verbunden sind. Die Arbeit von Colin Ware et al. bietet Anhaltspunkte, wie wir dieses Ziel erreichen können [WPCM02].

Wir müssen für ein gutes Layout jedoch auch schöngeistige Kriterien wie „Schönheit“ und „Harmonie“ des Graphen in Betracht ziehen, wie Chris Bennett et al. herausfanden. Wenn unser Graph gut aussieht, so fällt es dem Betrachter leichter ihn eingehender zu betrachten. Er wird eher geneigt sein, sich die Mühe zu machen, ihn zu verstehen [BRSG07]. Wir fassen im Folgenden die Begrifflichkeiten wie „Schönheit“, „Harmonie“ und „ansprechend“ als *Ästhetik* des Graphen zusammen. Ein ästhetischer Graph führt also zu einem lesbaren Graphen, und somit zu einem guten Layout.

Das Ziel die Ästhetik eines Graphen zu verbessern lässt sich jedoch nur dann erreichen, wenn wir konkrete Parameter zur Verfügung haben, nach denen wir ein Layout bewerten können. Ein gutes Gesamtbild über die verschiedenen Parameter, die zur Beurteilung eines Layouts dienen können, bietet die Arbeit von Bennett et al. [BRSG07]. Variablen, die ein gutes Layout ausmachen, und von verschieden Arbeiten bestätigt wurden sind:

- *Kantenkreuzungen:* Weniger ist besser. Abhängig vom Kantenrouting meist leicht zu messen. Leider für das Routing mit Kurven deutlich schwerer als beim Routing mit Geraden. Eine der wichtigsten Metriken für gutes Layout.
- *Kantenlänge:* Kürzer ist besser. Leicht zu messen. Ebenfalls eine der wichtigsten Metriken. Alternativ kann auch eine einheitliche Kantenlänge wünschenswert sein.
- *Kontinuität:* Keine engen Kantenkrümmungen oder -knicke. Ein- und ausgehende Kanten eines Knotens sollten einander gegenüberliegen.
- *Symmetrie:* Eine hohe Symmetrie ist einem guten Layout zuträglich. Wir wollen zwischen *Mikro-* und *Makrosymmetrie* unterscheiden. Mit Makrosymmetrie seien Grobstrukturen

wie die Symmetrie der Knotenplazierung oder des Kantengrobverlaufs (zum Beispiel links oder rechts am Knoten vorbei) bezeichnet. Mit Mikrosymmetie bezeichnen wir feinteilige Symmetrien, wie Positionierung der Ports oder Winkel der Kanten an einem Knoten.

- *Orthogonalität:* Knoten, die auf einem Raster angeordnet werden, sind vorteilhaft. Sind auch die Kanten in einem rechtwinkligen Muster angeordnet, so spricht man vom *Manhattan-Layout*.
- *Anzahl der Biegepunkte:* Weniger Richtungsänderungen der Kante sind besser.
- *Winkel:* Sowohl der Winkel der Kanten an Knoten, als auch bei Kantenkreuzungen ist zu beachten. Beide Winkel sollten möglichst groß sein.
- *Fläche des Graphen:* Kleiner ist besser. Auch als *Kompaktheit* des Graphen benannt.

3.1.2. Objektwahrnehmung

Um den Prozess des Erkennens und Verstehens eines Graphens besser zu erfassen, wollen wir einen Blick auf ein Forschungsgebiet der Psychologie werfen. Unter *Objektwahrnehmung* versteht man die Wahrnehmung von Objekten des täglichen Lebens, also von Objekten, die uns bekannt oder zumindest scheinbar bekannt sind. Für uns interessante Theorien aus dem Bereich der Objektwahrnehmung sind einerseits die *Gestaltpsychologie* und andererseits *konstruktivistische Erklärungstheorien*. Im Folgenden werden wichtige Aspekte kurz vorgestellt. Wir werden im weiteren Verlauf dieser Arbeit dann versuchen, Implikationen aus der Objektwahrnehmung abzuleiten.

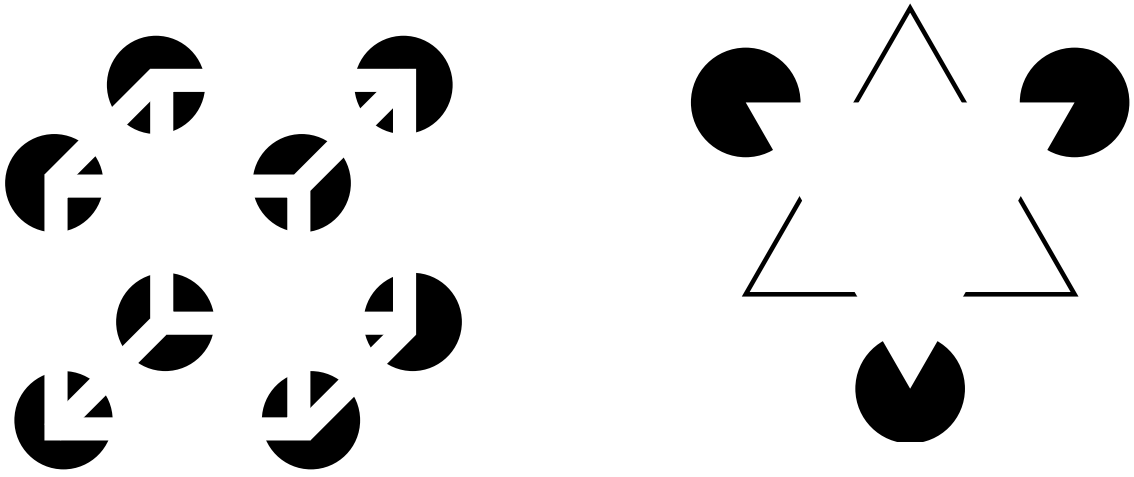
Gestaltpsychologie

Die Gestaltpsychologie wurde zu Beginn des 20. Jahrhunderts von einer Gruppe um die Psychologen Max Wertheimer, Wolfgang Köhler und Kurt Koffka entwickelt. Die Gruppe, auch *Berliner Schule* genannt, entwickelte *gestalttheoretische Gesetze*, mit deren Hilfe sie die Wahrnehmung von Objekten unserer Umwelt zu erklären versuchten. Folgende ihrer Erkenntnisse sind für unsere Arbeit interessant:

- *Gesetz der Prägnanz* (auch *Gesetz der guten Gestalt* oder *Gesetz der Einfachheit*)*:* Jeder Reiz wird so wahrgenommen, dass er eine möglichst einfache und einprägsame Gestalt darstellt, dass die entstehende Figur also so einfach wie möglich ist.

- *Gesetz der Ähnlichkeit*: Ähnlich aussehende Objekte werden eher als zusammengehörig wahrgenommen als unähnliche.

- *Gesetz der Kontinuität*: Reizelemente, die als Fortsetzung vorhergehender Reizelemente gesehen werden können, werden als einander zugehörig gesehen. So werden Linien an Schnittpunkten bevorzugt im Sinne einer Fortführung ihrer bisherigen Linienführung gesehen.



(a) Kein Würfel[1]

(b) Kanizsas Dreieck[2]

Abbildung 3.1. Durch Berechnung illusionärer Konturen sehen wir Strukturen, die nicht dargestellt sind. Sie sind nur durch ihre Ecken angedeutet.

Konstruktivistische Theorien

Die konstruktivistischen Theorien fassen die Wahrnehmung als Prozess auf, der in aufeinander folgenden Stufen durchlaufen wird. Zu Beginn des Prozesses steht die zweidimensionale Abbildung des Objektes auf der Netzhaut, an seinem Ende die Erkennung desselben. David Marr fasste diesen Prozess in vier Stufen [Mar82]:

1. Retinales Abbild des Objektes auf der Netzhaut
2. Ecken, Kanten und zusammenhängende Flächen (Elementarmerkmale) identifizieren
3. Identifizierte Elementarmerkmale gruppieren und verarbeiten
4. Dreidimensionales Objekt wahrnehmen

Interessant an diesem Prozess ist, dass Ecken und Kanten bereits sehr früh identifiziert werden. Details des Bildes werden erst später ergänzt. Dies kann auch zu Fehlinterpretationen des Bildes führen. Abbildung 3.1 zeigt hierfür zwei Beispiele. Wir erkennen Strukturen die nicht dargestellt sind. Sie sind nur durch ihre Ecken angedeutet. Erst bei genauerer Betrachtung des Bildes werden uns die eigentlichen Strukturen bewusst.

[1] Abbildung: Gemeinfrei von www.de.wikipedia.org

[2] Abbildung: Fibonacci, www.de.wikipedia.org, Lizenz: CC BY-SA 3.0

3.1.3. Warum Kurven?

Es ist keine empirische Arbeit bekannt, die Kurven mit Geraden hinsichtlich ihrer Eignung für die Informationsvermittlung vergleicht. Es scheint jedoch möglich, aus der Objektwahrnehmung einige Implikationen für ein gutes Kantenrouting abzuleiten. Das Gesetz der Kontinuität legt nahe, dass glatte und kontinuierliche Kanten als ein zusammenhängendes Objekt wahrgenommen werden, wie auch Colin Ware vermutete [War99]. Aus dem Gesetz der Ähnlichkeit könnte man schließen, dass glatte Kurven gut von rechteckigen Knoten zu unterscheiden sind. Marrs Arbeit und das Gesetz der guten Gestalt lässt uns folgern, dass wahrscheinlich vor allem die Anfangs- und Endstücke einer Kante von besonderer Bedeutung sind. Der genauer Kantenverlauf wird mutmaßlich erst bei längerer Bildbetrachtung wahrgenommen.

Betrachten wir die in Abschnitt 3.1.1 genannten Kriterien, so bietet das Kantenrouting mit Kurven sowohl Vor-, als auch Nachteile gegenüber des verbreiteten Manhattan-Routings:

- ▻ *Vorteile:* Die Kontinuität nimmt zu, da wir Kantenknicke durch Krümmungen ersetzen. Sowohl die Kantenlänge als auch die Anzahl der Biegepunkte nimmt ab (Siehe auch Abbildung 3.2b).

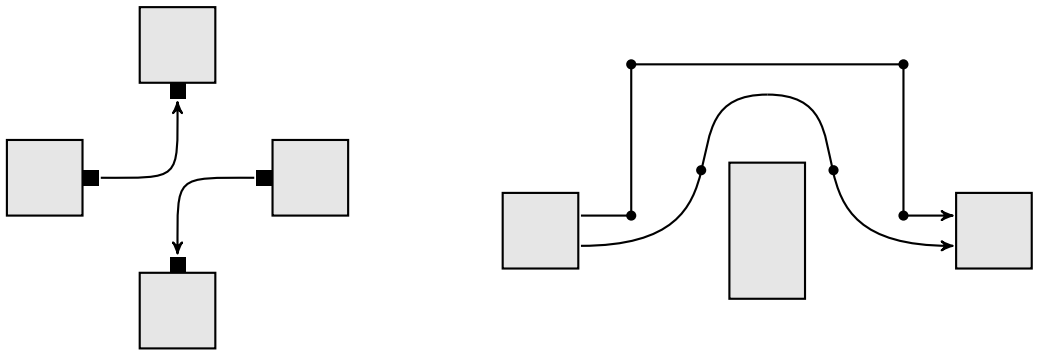
- ▻ *Nachteile:* Die Orthogonalität nimmt ab, da Kanten jetzt nicht mehr streng in horizontaler oder vertikaler Richtung liegen. Die Winkel an Kantenkreuzungen werden schlechter, denn im Manhattan-Routing sind diese immer gleich der optimalen 90°.

Auch bezüglich der persönlichen Präferenzen, also ob ein Anwender Kurven oder Geraden vorzieht, gibt es keine eindeutigen Ergebnisse. Männer tendieren beispielsweise eher zur Erzeugung eines Layouts mit geraden Kanten als Frauen [PPP12]. Kurven sind also nicht zwangsweise die besseren Kanten. Die Entscheidung über die Art des Kantenroutings ist somit in erster Linie eine Designentscheidung des Anwenders und hängt vom Anwendungsfall und den persönlichen Präferenzen ab.

3.1.4. Geeignete Parameter für das Kantenrouting

Durch unser Kantenrouting können wir nur auf eine kleine Auswahl der in Abschnitt 3.1.1 genannten Parameter Einfluss nehmen:

- *Kantenkreuzungen:* Auch wenn die Kreuzungsminimierung des Sugiyama-Algorithmus hier mehr Einfluss hat, so ist es trotzdem denkbar, die Anzahl der Kantenkreuzungen weiter zu senken. Abbildung 3.2a zeigt, in welchen Situationen dies geschehen kann. Da die Ports der Kanten jeweils auf gleicher Höhe beziehungsweise Breite liegen, ist ein Überschneidungsfreies Routing mit einem Manhattan-Routing nicht möglich. Es müssten entweder die Positionen der Knoten oder der Ports verändert werden. Gleichzeitig ist es jedoch auch denkbar, dass neue Kantenkreuzungen, zum Beispiel



(a) Vermeidung von Kantenkreuzungen **(b)** Zwei Richtungsänderungen gegenüber vier Knickpunkten

Abbildung 3.2. Vorteile von Kurven gegenüber eines Manhatttan-Routings.

durch zu kleine Radien beim umrunden von Knoten, entstehen. Dies sollte möglichst vermieden werden.

- *Kontinuität:* Durch den Einsatz von Kurven können wir die Kontinuität unserer Kanten verbessern. Zu kleine Kurvenradien würden diesen Vorteil wieder zunichte machen. Ein Schwellwert lässt sich in der Literatur nicht finden.
- *Mikrosymmetie:* Die Makrosymmetrie des Graphen können wir durch das Kantenrouting nur marginal beeinflussen. Gleichwohl können wir die Mikrosymmetie sehr wohl verbessern. So sollten wir beispielsweise dafür sorgen, dass Kanten an Knoten gleichmäßig auseinander- beziehungsweise zusammenlaufen .
- *Winkel der Kanten an Knoten und bei Kantenkreuzungen:* Kreuzen sich die Kanten in einem zu flachen Winkel, so fällt es schwer, die korrekte Fortführung der Kante zu erkennen. Je näher der Winkel an 90° liegt, desto besser. Biologisch bedingt sollte dieser Winkel nicht unter 30° fallen [BH85].

3.2. Kantenbeschriftungen

Zusätzliche Komplexität erhält unsere Aufgabe durch Beschriftungen (engl. *labels*), die sich an den Kanten befinden können. KLay Layered unterscheidet zwischen *Head-*, *Tail-* und *Center-Labels*. Für sie muss sowohl eine Position gefunden, als auch genügend Platz reserviert werden. Während die Platzierung eine lösbare Aufgabe darstellt, unterliegen wir bei der Reservierung des benötigten Raums gewissen Einschränkungen, da eine Anpassung der Portplatzierung nicht Teil dieser Arbeit ist. Es ist jedoch nicht in allen Situationen möglich, den für die Beschriftungen benötigten Raum zu reservieren ohne Einfluss auf die Portpositionen zu nehmen. Abbildung 3.3 zeigt, dass es bei Head- und Tail-Beschriftungen zwingend erforderlich ist, den Platz zum nächsten Port zu vergrößern.

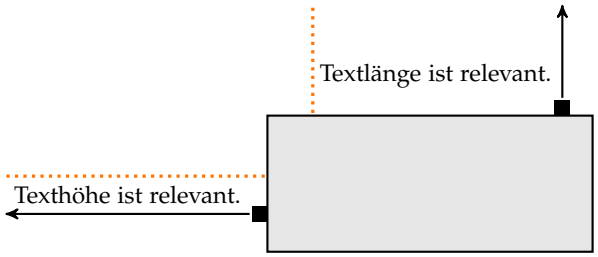


Abbildung 3.3. Für Head- und Tail-Beschriftungen muss Platz zum benachbarten Port reserviert werden. Gerade bei Nord- und Süd-Ports führt dies zu großen Abständen.

Eine denkbare und einfache Lösung wurde bei der Erstellung dieser Arbeit angedacht und wieder verworfen. KLay Layered stellt auch für Ports eine sogenannte *Margin* bereit. Diese Margin definiert den Platz um den Port herum, der frei gehalten werden soll. Benötigen wir für ein Label seitlich mehr Platz, so könnte man die Margin des Ports entsprechend erweitern. Erweitern wir die Margin noch vor der dritten Phase, so wird sie bei der Portplazierung berücksichtigt. Dies führt jedoch zu Problemen an anderer Stelle. Haben wir beispielsweise einen Knoten mit fester Knotengröße vorliegen, so können wir die Margin der Ports nicht beliebig erhöhen. Ist die Knotengröße variabel, so wüchse der Knoten zwar um das benötigte Maß, dies kann jedoch zu einem unausgewogenen Gesamtbild des Graphen führen. Gerade bei Nord- oder Süd-Ports kann der benötigte Platz bei horizontaler Schrift sehr groß werden. Daher wurde dieser Ansatz verworfen. Dieses Problem muss in einer zukünftigen Arbeit gelöst werden. Wir werden Head- und Tail-Labels nur platzieren und Kollisionen nicht berücksichtigen.

Center-Labels können jedoch von uns sowohl platziert als auch freigestellt werden. Wie dies konkret geschieht, werden wir bei der folgenden Behandlung der einzelnen Kantenarten sehen.

3.3. Routen normaler Kanten

Wir werden uns in diesem Abschnitt zunächst mit dem Routing im normalen Fall beschäftigen: Eine Kante verläuft von einem Knoten a zu einem Knoten $b \neq a$ durch den Graphen. In den folgenden Abschnitten beschäftigen wir uns dann mit zwei Sonderfällen.

Das Routing normaler Kanten durch den Graphen stellt sich als eine Abfolge von Segmenten zwischen je zwei benachbarten Ebenen dar. Kurze Kanten bestehen nur aus einem Segment, lange Kanten aus mehreren. Wir betrachten zunächst das Routing der kurzen Kanten, um darauf aufbauend das Routing der langen Kanten zu betrachten.

3.3.1. Kurze Kanten

Kurze Kanten müssen nur von einem Layer zum nächsten geroutet werden. Dank des KLay Layered-Algorithmus können wir davon ausgehen, dass wir von einem Ost-Port zu

einem West-Port routen. Welche Form die Kanten annehmen sollen ist hier die eigentliche Herausforderung. Die Objektwahrnehmung wird uns hier Anhaltspunkte für ein gutes Layout geben. Drei Aspekte sind beim Routen kurzer Kanten zu beachten:

Festlegung der Anfangs- und Endwinkel
Für die Wahl des Winkels, in dem eine Kante einen ihrer Knoten berührt gibt es grundsätzlich zwei Alternativen: entweder sie trifft in einem rechten Winkel auf den Knoten, oder der Winkel weicht im Berührungspunkt vom rechten Winkel Richtung des anderen Knotens ab.

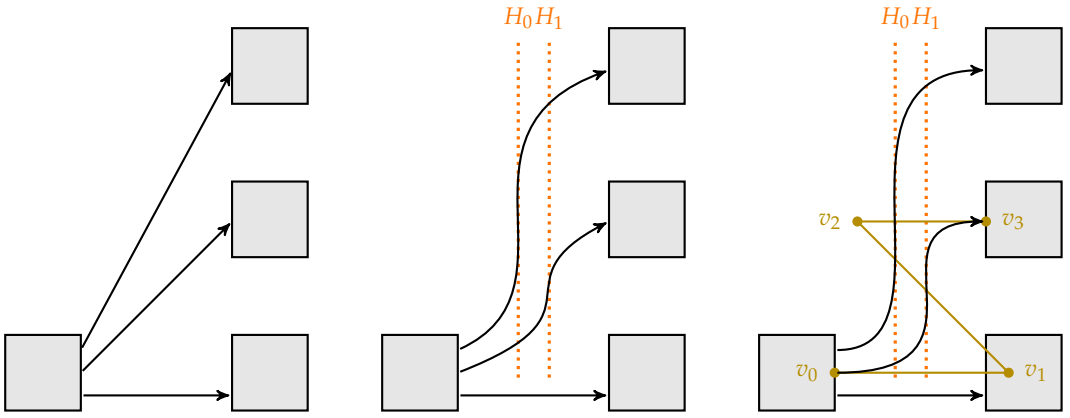
Nach anfänglichen Experimenten mit dem zweiten Ansatz zeigte sich der erste als geeigneter. Eine Abweichung vom rechten Winkel hat den Vorteil, dass die Kurve eine geringere Krümmung aufweisen muss. Ein anderer Aspekt, der für diesen Ansatz spricht leitet, sich aus der Objektwahrnehmung ab. Wie in Abschnitt 3.1.2 besprochen sind für eine schnelle Erfassung einer Form ihre Ecken und Enden wichtiger als ihre Mitte. Dies spricht für den zweiten Ansatz, da bei diesem die Kanten bereits an ihrem Ansatz ihre spätere Verlaufsrichtung andeuten.

Der erste, orthogonale Ansatz hingegen bietet ein ruhigeres Bild für den Betrachter. Demonstrationen im kleinen Kreis zeigten eine höhere Akzeptanz für diesen Ansatz. Zudem krümmen sich die Kanten kurz nach ihrem Beginn in Richtung des Zieles. Somit erfüllen sie auch in dieser Variante das Gesetz der guten Fortsetzung, womit einer der Vorzüge des zweiten Ansatzes aufgewogen wird.

Variation der Kurvenkrümmung
Eine schwächere Krümmung führt nach den Ergebnissen der Objektwahrnehmung dazu, dass der Kantenverlauf besser erfassbar wird. Die Gesamtkante wird eher als ein zusammenhängendes Objekt wahrgenommen. Gleichzeitig führen schwächere Krümmungen möglicherweise zu Überschneidungen. Ein gutes Maß kann hier nur experimentell bestimmt werden.

Höhenunterschied zwischen Start- und Zielknoten
Auch hier sind wieder zwei Ansätze denkbar: wir können die Knoten durch eine Gerade verbinden, oder wir bilden eine gekrümmte Kurve, so dass sich ein mehr oder minder stark ausgeprägtes vertikales Segment ergibt. Alle in Unterkapitel 1.4 vorgestellten Arbeiten folgen dem ersten Ansatz. Er führt, vor allem bei runden Knoten, zu sehr harmonischen Ergebnissen. Verbindet man bei kreisförmigen Knoten die Zentren der Knoten durch eine gerade Linie, so trifft man die Knotenkante gerade im rechten Winkel. Dieses Routing ist einfach, ästhetisch und effizient.

Jedoch berücksichtigt keine der Implementierungen Ports. Die Ports zwingen uns eine vordefinierte Stelle am Knoten anzusteuern. Verbinden wir die Ports mit einer geraden Linie, so ergeben sich bei großen Höhenunterschieden sehr spitze Winkel an den Ports. Daher werden wir den zweiten Ansatz implementieren.



(a) direkt **(b)** angewinkelt **(c)** orthogonale Enden

Abbildung 3.4. Drei Varianten des Routings kurzer Kanten. Nehmen die Kanten eine vertikale Richtung an, so müssen sie horizontal nebeneinander sortiert werden. Die in der dritten Abbildung eingezeichneten Kontrollpunkte zeigen, wie ein solches vertikales Segment auch mit nur einer Bezierkurve erzeugt werden kann.

Abbildung 3.4 zeigt einige Variationen dieser drei Parameter. Je stärker wir von einem Routing mit Geraden abweichen, desto eher laufen die Kanten auf einem Teil ihres Weges in vertikaler Richtung. Dadurch kann es dazu kommen, dass sich die Kanten überdecken. Daher müssen wir vertikale Ebenen zwischen den Layern einführen und die Kanten horizontal nebeneinander auf ihnen sortieren. In Abbildung 3.4b und 3.4c sind diese Ebenen als H_0 und H_1 dargestellt. Die Zuordnung der Kanten zu einem der vertikalen Segmente beeinflusst die Anzahl der daraus resultierenden Kantenkreuzungen. Wie wir dieses Problems lösen, werden wir erst in Abschnitt 3.5.1 betrachten, da wir hierfür zunächst noch weitere Begriffe und Zusammenhänge erörtern müssen. Alle Kurven in den beiden Abbildungen 3.4b und 3.4c sind durch eine Bezier-Kurve darstellbar. In Abbildung 3.4c ist das Kontrollpolygon einer der Kurven eingezeichnet. Die Berechnung der Kontrollpunkte ist in dem dargestellten Fall sehr einfach. Um ein vertikales Segment zu erhalten, muss der horizontale Abstand zwischen v_1 und v_2 gerade dem zwischen v_0 und v_3 entsprechen. Leider stellen sich nicht alle Fälle so einfach dar, wie wir im folgenden Abschnitt sehen werden.

3.3.2. Lange Kanten

Im Gegensatz zu kurzen Kanten laufen lange Kanten über mehrere Ebenen unserer Ebenenzuweisung. Hierbei durchlaufen sie in jeder Ebene einen Dummy-Knoten. Abbildung 3.5 zeigt, wie wir mit der Methode der kurzen Kanten auch lange Kanten routen könnten. Wir verlieren im dargestellten Beispiel jedoch die C^2-Stetigkeit: An den Dummy-Knoten d_0 und d_1 ist der Übergang zwischen den Kurven nur noch C^1-stetig. Wenn

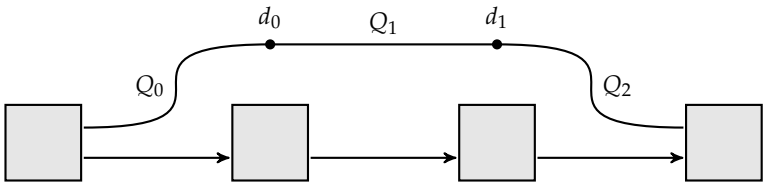


Abbildung 3.5. Routing einer langen Kanten auf Basis der kurzen Kanten. Dieses Routing ist an den Stellen d_0 und d_1 nur noch C^1-stetig.

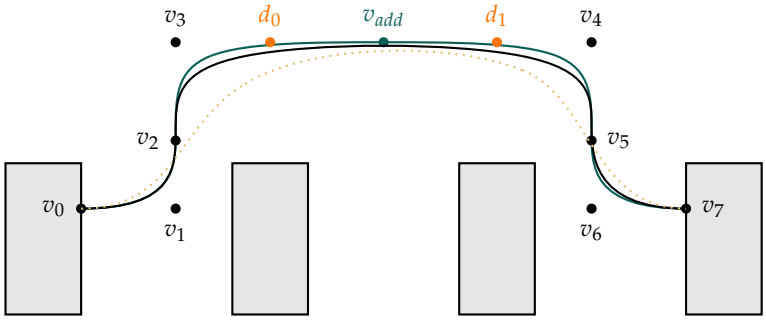


Abbildung 3.6. Eine lange Kante mit B-Splines geroutet. Um den Kurvenverlauf mehr in die Horizontale zu bringen, kann ein zusätzlicher Knoten in der Mitte zwischen d_0 und d_1 platziert werden. Ohne vertikale Mittel-Kontrollpunkte v_2 und v_5 wird die Kante nicht vertikal, wie die gepunktete Linie zeigt. Die Abbildung ist im Vergleich zu Abbildung 3.5 vertikal um den Faktor 2 gestreckt, um das Kurvenverhalten zu verdeutlichen.

beispielsweise Q_2 eine C^2-stetige Fortsetzung von Q_1 sein sollte, so müssten die beiden inneren Kontrollpunkte von Q_2 auf $\overline{d_0 d_1}$ liegen, da es sich bei Q_1 um eine Gerade handelt. Auch eine Reduzierung der Anforderungen auf eine G^2-Stetigkeit löst dieses Problem nicht. Bei Beta-Beziers haben wir in Punkt 4. ihrer Eigenschaften festgehalten, dass Q_1 unseres Beispiels eine Rechtskrümmung in d_1 aufweisen müsste, wenn wir die Kurven mit Beta-Beziers routen wollen. Somit bieten sich uns zwei Lösungsansätze: die Nutzung von B-Splines oder eine weitere Unterteilung der Vertikalen Segmente Q_1 und Q_2, etwa in ihrer Mitte.

In Abbildung 3.6 sehen wir eine Lösung mit B-Splines. Der Vorteil der B-Splines ist es, dass die Platzierung der Kontrollpunkte intuitiv erfolgen kann. Die resultierende Kurve ist automatisch C^2-stetig. Der Nachteil ist, dass der Kurvenverlauf nicht so genau vorbestimmbar ist. Der erste Versuch, der durch acht Kontrollpunkte bestimmt wird, verfehlt die Dummy-Knoten deutlich. Eine Besserung können wir erreichen, indem wir einen zusätzlichen Kontrollpunkt v_{add} in die Mitte der Geraden zwischen d_0 und d_1 platzieren. Ebenso müssen wir einen Kontrollpunkt auf den vertikalen Segmenten setzen (v_2 und v_5). Ohne diesen Kontrollpunkte würde die Kurve nicht in die Vertikale gehen, wie an der gepunkteten Kurve zu erkennen ist. Sie hat die Kontrollpunkte v_0, v_1, v_3, v_4, v_6 und v_7.

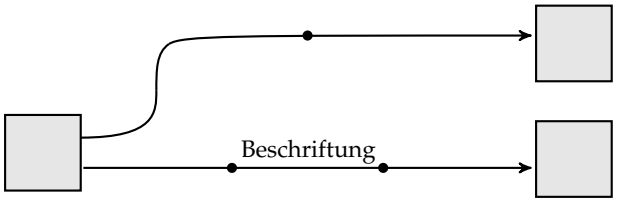


Abbildung 3.7. Für Center-Labels in normalen Kanten werden Dummy-Knoten eingefügt.

3.3.3. Center-Labels

Normale Kantenbeschriftungen, also solche, die sich in der Mitte der Kante befinden, lassen sich beim Kantenrouting mit Kurven auf dieselbe Weise platzieren und freistellen wie dies auch beim orthogonalen Routing mit Geraden geschieht. Für das Label wird ein Dummy-Knoten in die Kante eingefügt, der so groß wie das Label ist. Dies bedeutet für kurze Kanten, dass sie zu langen Kanten werden, da sie nun durch einen Dummy-Knoten laufen müssen. Der Beschriftungsknoten wird dann fast wie ein normaler Knoten von KLay Layered behandelt. Der einzige Unterschied ist, dass die Kante nicht durch den Knoten laufen darf, da sie ansonsten genau durch die Beschriftung läuft. Sie muss entweder über oder unter dem Knoten durchgeführt werden. Die Strecke vom Eingangs- zum Ausgangsport des Dummy-Knotens wird dann einfach verbunden. Im orthogonalen Fall durch eine Gerade, bei der Nutzung von B-Splines muss sie gar nicht weiter beachtet werden. Es werden einfach beide Ports als Kontrollpunkte dem Spline beigefügt. Abbildung 3.7 zeigt dieses Vorgehen exemplarisch.

3.4. Routen von Selfloops

Selfloops sind Kanten, die wieder an ihren Startknoten zurück laufen.

29. Definition (Selfloop)**.** Sei $G = (V, E, P, f)$ ein portbasierter Graph. Eine Kante $e = (p_1, p_2) \in E$ heißt *Selfloop*, wenn $v(p_1) = v(p_2)$ gilt. Es gilt dann $l(v(p_1)) = l(v(p_2))$. Wir definieren $l(e)$ als Notation für die Ebene eines Selfloops.

Aus der Definition der Selfloops folgt direkt, dass sie gegen die Regeln einer Ebenenzuweisung verstoßen. In einer Ebenenzuweisung müssen alle Kanten zu einem höheren Layer laufen, siehe hierzu Definition 6. Enthält ein portbasierter Graph $G = (V, E, P, f)$ einen Selfloop $e = (p_1, p_2)$, so müssen wir den Graphen modifizieren, um eine Ebenenzuweisung erzeugen zu können. Hierzu fügen wir einen Dummy-Knoten d mit $|l(d) - l(e)| = 1$ in den Graphen ein, und ersetzen die Kante e durch zwei Kanten $e_0 = (p_1, p_3)$ und $e_1 = (p_4, p_2)$ mit $p_3, p_4 \in P(d)$. Durch Drehen einer der Kanten erhalten wir eine strenge Ebenenzuweisung. Somit kann ein Layout für den modifizierten Graphen nach dem Sugiyama-Algorithmus erstellt werden. Das VCG-Tool verfährt auf diese Art (siehe Abbildung 1.6).

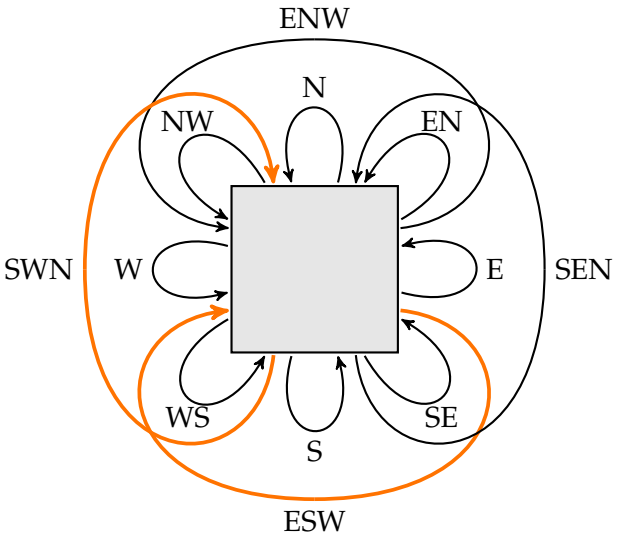


Abbildung 3.8. Alle Seitenkombinationen für Selfloops. Hervorgehoben die Kombinationen, die im Uhrzeigersinn gezeichnet werden.

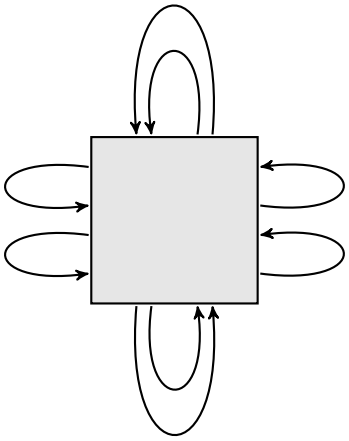


Abbildung 3.9. Selfloops an der Seite eines Knotens. Sie können gestapelt oder aufgereiht dargestellt werden.

Ein alternativer Ansatz ist es, Selfloops separiert zu behandeln und vom restlichen Sugiyama-Algorithmus weitestgehend auszuklammern. Dieser Weg wird in dieser Arbeit gewählt. Bereits vor der ersten Phase entfernen wir alle Selfloops aus dem Graphen, routen sie unabhängig vom restlichen Layoutprozess, und fügen sie noch vor Phase vier wieder in den Graphen ein. So werden sie bei der Brechung der Zykel in Phase eins ausgenommen. Wir routen die Selfloops dann im direkten Umfeld des Knotens v, an dem sie liegen. Hierbei müssen wir zur Vermeidung von Überschneidung weder andere Knoten berücksichtigen, noch Kanten, die nicht mit dem Knoten v verbunden sind. Nachdem das Routing der Loops fertig gestellt ist, erhöhen wir die *margin* des Knotens v so, dass es in den Phasen vier und fünf nicht zu neuen Überschneidungen kommen kann. Beim Kantenrouting in Phase fünf müssen wir die Selfloops dann explizit ignorieren, so dass sie nicht erneut geroutet werden.

3.4.1. Darstellungsformen

KLay Layered unterscheidet zwischen Nord-, Ost-, Süd- und Westports. Somit ergeben sich die in Abbildung 3.8 dargestellten 12 verschiedene Kombinationen der Portseiten. Sind die Portseiten nicht durch die Graphendefinition festgelegt, so werden die Loops nach folgenden Regeln ausgerichtet:

1. Flussrichtung Ost nach West, Süd nach Nord.
2. Drehrichtung gegen den Uhrzeigersinn.

Hierbei bindet die erste Regel stärker als die zweite, denn für zwei Kombinationen (SWN und ESW) widersprechen sich die Festlegungen. Sie sind in Abbildung 3.8 hervorgehoben.

30. Definition (Seitenloop, Eckloop, Diametralloop)**.** In einem portbasierten Graphen bezeichnen wir einen Selfloop e als

- *Seitenloop*, falls $ps(p_s(e)) = ps(p_t(e))$,
- *Eckloop*, falls e kein Seitenloop ist und es gilt

$$ps(p_s(e)),\, ps(p_t(e)) \in S \text{ mit } S \in \{\{n,e\},\, \{n,w\},\, \{s,e\},\, \{s,w\}\}\,,$$

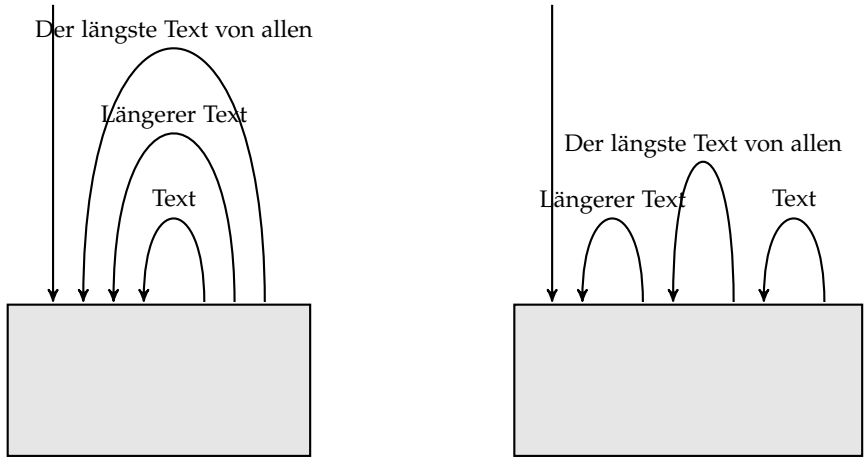
- *Diametralloop*, falls e kein Seitenloop ist und es gilt

$$ps(p_s(e)),\, ps(p_t(e)) \in S \text{ mit } S \in \{\{w,e\},\, \{n,s\}\}\,.$$

Gibt es für eine Portseite mehrere Seitenloops, so haben wir wiederum zwei Möglichkeiten diese Ports anzuordnen. Die Kanten können entweder gestapelt oder aufgereiht werden, wie in Abbildung 3.9 dargestellt. Welche dieser verschiedenen Darstellungsformen das beste Layout ergibt lässt sich nicht allgemeingültig sagen, dies ist von vielen Faktoren abhängig und letztendlich eine Designentscheidung. Es ist möglich, dass die Portseiten oder sogar die Portreihenfolge in der Graphendefinition festgelegt sind. Daher muss unsere Implementierung grundsätzlich alle 12 Kombinationen erfassen.

Sind die Portseite frei wählbar, so müssen wir uns um eine vorteilhafte Verteilung bemühen. Hierfür werden zwei Algorithmen angeboten. Der erste platziert alle Loops an eine (die nördliche) Seite des Knotens. Diese Verteilung ist vor allem bei großen Knoten vorteilhaft, da alle Loops in ihrer Gesamtheit vom Betrachter mit einem Blick erfasst werden können. Der zweite Algorithmus bemüht sich um eine gleichmäßige Verteilung der Loops um den Knoten. Dies führt vor allem bei kleinen Knoten zu einem ästhetisch angenehmen Erscheinungsbild führt.

Einseitige Platzierung Bei der Einseitigen Platzierung werden alle Selfloops an der Nordseite des Knotens platziert. Sowohl die gestapelte als auch die aufgereihte Variante lässt sich für Loops ohne Beschriftungen leicht realisieren. Werden den Kanten jedoch Beschriftungen beigefügt, stößt dieses Routing schnell an seine Grenzen. Ist der Text breiter als der Abstand zwischen den Ports des Loops kann es zu Kanten- oder Textüberschneidungen kommen. Um dieses Problem zu minimieren werden die Loops nach der Länge ihres Textes sortiert. Bei gestapelter Anordnung sortieren wir die Loops nach der Länge ihrer Beschriftungen von innen nach außen. Bei der aufgereihten Anordnung kommt der Loop mit dem längsten Text in die Mitte und wird höher als seine Nachbarn gezogen. In beiden Ausführungen kann es jedoch trotzdem zu Überschneidungen der Beschriftungen mit anderen Kanten kommen, wie in Abbildung 3.10 zu sehen ist. Bei der gestapelten Ausrichtung kommt es spätestens dann zu Überschneidungen, wenn die Beschriftung einen inneren Loops breiter ist als der Abstand der Ports des ihn umgebenden Loops. Bei der aufgereihten Ausrichtung kann es immer dann zu Überschneidungen kommen,



(a) Gestapelte Ausführung **(b)** Aufgereihte Ausführung

Abbildung 3.10. In beiden Anordnungsvarianten der Seitenloops kann es zu Überschneidungen kommen. Überschneidungen mit Beschriftungen anderer Loops lassen sich vermeiden, Überschneidungen mit anderen Kanten jedoch nicht.

wenn die Beschriftung breiter ist als der Abstand zwischen seinen Nachbarports. Eine Ausnahme bildet hier der mittlere Loop. Seine Beschriftung ist so positioniert, dass sie nicht mit anderen Loops kollidieren kann. Es ist jedoch noch immer möglich, dass sie mit nicht-Selfloop-Kanten kollidiert. Gleiches gilt für den äußeren Loop in der gestapelten Variante. Um dies zu verhindern, müssten wir wieder Einfluss auf die Positionierung der Ports nehmen.

Gleichmäßige Verteilung Bei der gleichmäßigen Verteilung von Selfloops haben wir größere Freiheiten bezüglich der Seitenwahl. Alle in Abbildung 3.8 dargestellten Alternativen sind denkbar. Ziel ist es, durch gleichmäßiges Verteilen der Loops ein harmonisches und ästhetisches Erscheinungsbild zu erreichen. Kanten ohne Beschriftungen verteilen wir auf die acht Eck- und Seitenloop-Varianten. Wir gehen in zwei Phasen vor. Das Ergebnis kann in Abbildung 3.11 betrachtet werden.

1. Zunächst stellen wir fest, an welchen Portseiten (N, E, S, W) sich bereits Kanten zu anderen Knoten befinden. Aus der Anzahl der belegten Portseiten und deren Lage wird ein *Symmetriezentrum* für die Loop-Verteilung gewählt. Das Symmetriezentrum liegt an einer der acht Loop-Seiten (Eck- oder Seitenloops) und ist in Abhängigkeit der Anzahl belegter Portseiten wie folgt definiert

   Alle Portseiten frei: Die Südseite.

   Eine Portseite belegt: Die belegte Portseite.

   Zwei Portseiten belegt:

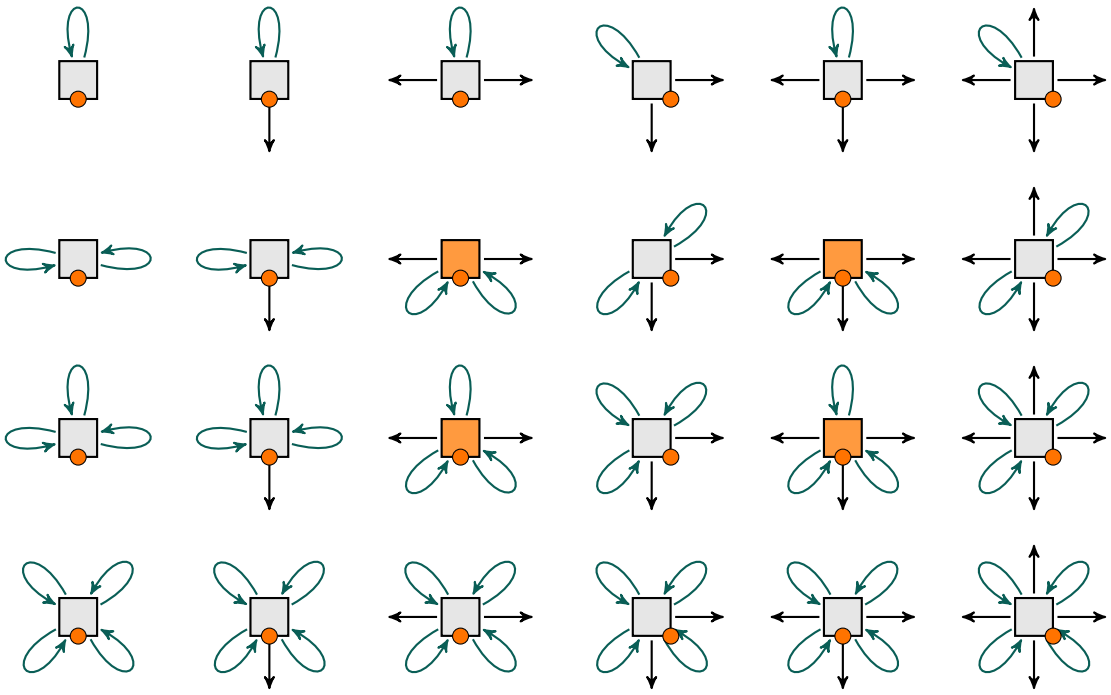


Abbildung 3.11. Gleichmäßige Verteilung der Selfloops nach Algorithmus 1. Von links nach rechts die Anzahl der belegten Portseiten. Von oben nach unten die Anzahl der Loops. Der Punkt stellt das Symmetriezentrum dar. Die vier hervorgehobenen Kombinationen stellen die in Zeile 10 behandelten Sonderfälle dar. Sollten mehr als vier Loops zu verteilen sein, so werden zunächst vier an den Ecken platziert, die restlichen dann entsprechend ihrer verringerten Anzahl.

- Falls zwei benachbarte Portseiten belegt sind, die Ecke zwischen ihnen.
- Sonst: Ost oder Süd, je nachdem welche Portseite frei ist.

Drei Portseiten belegt: Die mittlere, belegte Portseite. Äquivalent: immer gegenüber der freien Portseite.

Alle vier Portseiten belegt: Die süd-östliche Ecke.

2. Anschließend werden die Selfloops in Abhängigkeit ihrer Anzahl verteilt. Dies erfolgt rekursiv nach Algorithmus 1. Bei dem Algorithmus ist zu beachten, dass im Fall von drei zu erzeugenden Loopseiten beide Fälle der `switch`-Anweisung in Zeile 5 durchlaufen werden.

Für den Fall, dass Kanten mit Text versehen sind, wird noch eine weitere Phase zwischen den beschriebenen Phasen eingefügt. In dieser werden die beschrifteten Kanten gesondert behandelt. Wie in Abbildung 3.12 zu sehen, ist ein Routing beschrifteter Kanten als Eck- oder horizontaler Diametralloop vorteilhaft. Diese Loops können theoretisch

Algorithmus 1 : Algorithmus zur gleichmäßigen Verteilung der Selfloops

Eingabe : n: Anzahl benötigter Loopseiten
Z: Symmetriezentrum
NotFree: Liste der belegten Portseiten

Ausgabe : Eine Liste von Selfloop-Seiten

Inizialisierung : LoopSides ← List()

```
1  while (n ⩾ 4) do
2      Falls noch vier oder mehr Loops verbleiben, fügen wir einen an jede Ecke
3      LoopSides <-add- {NE, SE, SW, NW};
4      n ← n − 4;
5  switch n do
6      case (1 ∨ 3)
7          Fügt die Loopside hinzu, die dem Zentrum gegenüber liegt
8          LoopSides <-add- gegenüber(Z);
9      case (2 ∨ 3)
10         if (linkerNachbar(linkerNachbar(Z)) ∈ NotFree) then
11             Falls die Loopside zwei weiter links mit einer externen Kante belegt ist, so
               fügen wir die direkten Nachbarn hinzu
12             LoopSides <-add- linkerNachbar(Z);
13             LoopSides <-add- rechterNachbar(Z);
14         else
15             Ansonsten fügen wir die beiden Loopsides hinzu, die zwei weiter links,
               beziehungsweise rechts liegen
16             LoopSides <-add- linkerNachbar(linkerNachbar(Z));
17             LoopSides <-add- rechterNachbar(rechterNachbar(Z));
18 return LoopSides;
```

beliebig weit nach Außen wachsen. Hierbei können die Diametralloops deutlich längere Texte aufnehmen, ohne, dass die Ästhetik leidet.

Kanten, deren Beschriftung eine definierte Länge überschreitet, werden zu horizontalen Diametralloops, also zu ENW- oder ESW-Loops. Ist dies nicht möglich, weil bereits Kanten zu anderen Knoten sowohl an der Nord- als auch an der Südseite des Knoten liegen, so werden die Kanten zu Eckloops. Beschriftete Kanten, deren Beschriftung die definierte Länge nicht überschreiten, werden ebenfalls zu Eckloops. Eckloops lassen sich immer Überschneidungsfrei anordnen, selbst dann, wenn alle Seiten des Knotens Kanten zu anderen Knoten aufweisen. Werden die Texte zu lang, so erhalten wir wieder ein unansehnliches Layout. Die Korrektheit bezüglich der Überschneidungsfreiheit bleibt trotzdem erhalten.

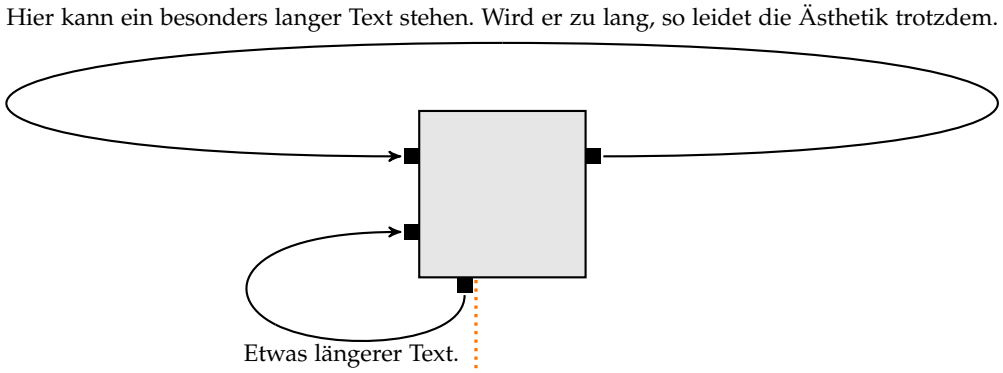


Abbildung 3.12. Selfloops mit Beschriftungen beliebiger Länge können an den horizontalen Diametralloops (ENW, ESW) und an allen vier Eckloops platziert werden. Beschriftungen an Eckloops dürfen nicht über den inneren Port ihres Loops hinauswandern.

3.5. Routen von Hyperkanten

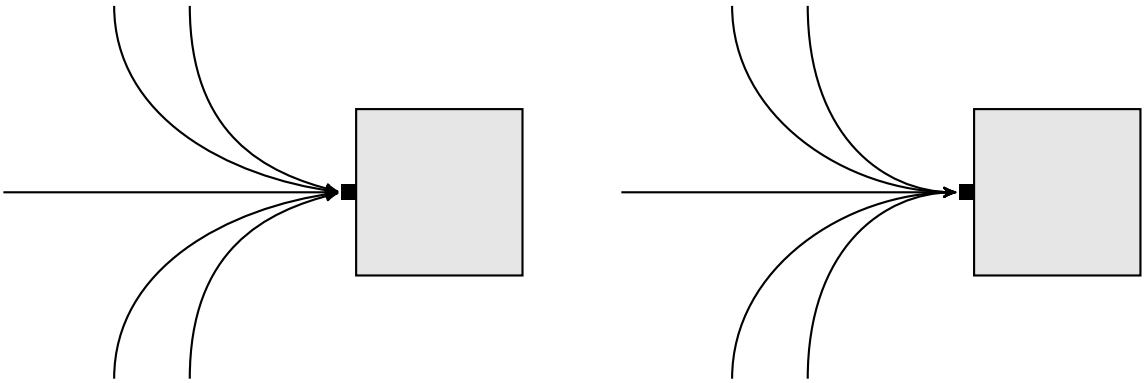
Es ist in KLay Layered möglich, dass an einem Port mehrere Kanten liegen. Betrachten wir den Fall, dass an einem Zielport mehrere Kanten liegen. Würde jede kante einen individuellen Winkel am Zielport haben, so würden sich die Pfeilspitzen gegenseitig überlagern. Das unschöne Ergebnis kann in Abbildung 3.13a betrachtet werden. Dies ist ein weiterer Grund dafür, dass wir uns in Abschnitt 3.3.1 für eine orthogonale Ausrichtung der Kanten an den Knoten entschieden haben. Alle Kanten haben am Zielport den selben Winkel, und ihre Pfeilspitzen sind deckungsgleich, wie in Abbildung 3.13b zu sehen. Um das in Abschnitt 3.1.2 definierte Ziel der Symmetrie zu erfüllen, müssen wir dann auch die Winkel in den Startports einheitlich wählen.

Aber wenn wir schon die Kantenenden deckungsgleich gestalten, so ist es doch denkbar die Kanten auf einem längeren Stück deckungsgleich darzustellen. Warum sollten die beiden nach oben beziehungsweise nach unten abbiegenden Kanten in Abbildung 3.13 nicht den selben Verlauf nehmen? Wir werden in diesem Abschnitt also Kanten betrachten, die sich einen Port teilen, und wir geben ihnen den Nahmen *Hyperkante*.

31. Definition (Hyperkante)**.** Alle gemeinsam an einem Port liegenden Kanten bilden eine *Hyperkante*.

Der Begriff der Hyperkante kommt eigentlich aus der Graphentheorie. Dort ist ein *Hypergraph* ein ungerichteter Graph, in dem eine Kante nicht nur zwei, sondern eine beliebige Anzahl von Knoten miteinander verbindet. Die Kanten in einem Hypergraphen werden dann Hyperkanten (engl. *hyperedges*) genannt.

Unsere Hyperkanten unterscheiden sich jedoch deutlich von den Hyperkanten der Graphentheorie. Zunächst einmal sind unsere Kanten gerichtet. Zudem kann in KLay Layered ein Port entweder nur eingehende oder nur ausgehende Kanten haben. Des weiteren ist



(a) Zielwinkel variabel **(b)** Zielwinkel einheitlich

Abbildung 3.13. Variabler oder fixer Zielwinkel von Kanten.

unsere Definition der Hyperkante nicht transitiv. Zwei Kanten ohne gemeinsamen Port fassen wir also nicht zusammen, auch wenn sie eine gemeinsame Kante haben, mit der sie jeweils einen Port teilen. Die Entscheidung gegen eine transitive Definition fiel, da es zu Verständnisproblemen kommen kann, wenn wir zu viele Kanten zusammenfassen. Auf diesen Punkt kommen wir später zurück.

Eine Hyperkante besteht also bei uns entweder aus einem Start- und mehreren Zielports, oder aus einem Ziel- und mehreren Startports. Es ist theoretisch möglich, dass eine Kante in zwei verschiedenen Hyperkanten liegt. Es muss dann entschieden werden, welcher dieser Hyperkanten sie zugeordnet wird.

Betrachten wir einen konkreten Beispielgraphen. Es sei $G = (V, E, l)$ ein ebenenbasierter Graph, mit

$$\begin{aligned} V =& \{v_0, v_1, v_2, v_3, v_4\} \\ E =& \{(v_0, v_3), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4)\} \\ l :=& \{v_0, v_1, v_2\} \to 0,\ \{v_3, v_4\} \to 1 \end{aligned}$$

Ports lassen wir hier außer acht; wir gehen davon aus, dass jeder Knoten nur einen Port hat. Abbildung 3.14 zeigt verschiedene Darstellungen dieses Graphen. In einem orthogonalen Kantenrouting fällt die Gestaltung der Hyperkanten, die nur Knoten benachbarter Layer verbinden nicht schwer. Die in Abbildung 3.14a dargestellte Hyperkante umfasst nun alle Kanten, die an den Knoten v_1 oder v_2 starten. Er geht also über unsere Definition hinaus. Dies ist im orthogonalen Fall gut darstellbar. Jedoch kann die Darstellung fehlinterpretiert werden. Hat Knoten v_0 eine Kante zu Knoten v_4?

Das dies nicht der Fall ist, ist in den beiden Versionen des Routings mit Kurven deutlicher zu erkennen. Dafür ergeben sich andere Probleme. Die Kante von v_1 zu v_3 ist nicht Teil einer Hyperkante, da ihr Höhenunterschied zu gering ist. In der Version eins wurden möglichst viele der Kanten zu einer Hyperkante vereinigt, mehr als in

unserer Definition. Dies führt dazu, dass wegen der Kurvenform eine zusätzliche vertikale Kante gezogen werden muss. Das entstehende Dreieck kann ebenfalls zu Irritationen führen. In Version zwei wurde nur eine Hyperkante aus den Kanten (v_2, v_3) und (v_2, v_4) gebildet. Dies führt dazu, dass wir eine zusätzliche vertikale Ebene einführen müssen, da ansonsten eine Überlagerung mit der Kante (v_1, v_4) entstehen würde. Zudem ist eine weitere Kantenkreuzung entstanden.

Es ist zu erkennen, dass auch bei Hyperkanten sowohl Vorteile als auch Nachteile im Kantenrouting mit Splines liegen. Außerdem gibt es verschiedene Ansätze, wie Hyperkanten gebildet und dargestellt werden. Wir werden in dieser Arbeit Hyperkanten nach folgenden Regeln bilden:

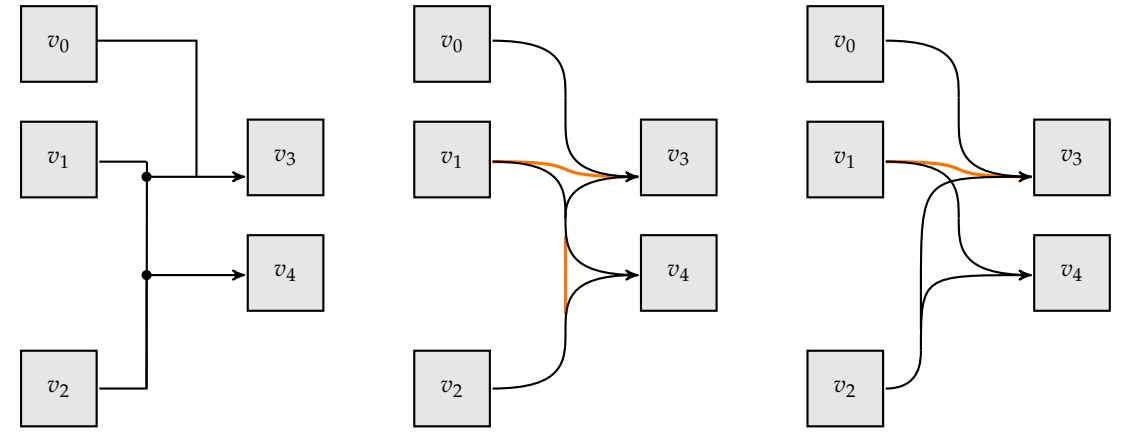
- Alle Kanten, die gemeinsam an einem Port liegen und deren Zielport in der gleichen vertikalen Richtung (oben oder unten) liegen, bilden eine Hyperkante.
- Ist die Höhendifferenz des Gegenports einer Kante jedoch unterhalb eines zu definierenden Schwellwerts, wird die Kante nicht der Hyperkante zugeordnet.
- Ist die Zuordnung einer Kante in mehrere Hyperkanten möglich, ist eine Hyperkante nach Belieben wählbar.

Kantenbeschriftungen bei Hyperkanten treten nicht auf. Alle Kanten, die in Hyperkanten eingehen, sind zuvor einfache Kanten. Werden wie in Unterkapitel 3.3 beschrieben Beschriftungsknoten eingefügt, so bildet das Kurvensegment, welches für eine Hyperkante in Frage kommt, eine kurze Kante ohne Beschriftung. Ein weiterer Aspekt der Hyperkanten ist die Möglichkeit, layerübergreifende Kanten zusammen zu fassen. Eine solche Zusammenfassung ist jedoch nicht Teil dieser Arbeit, da diese Fragestellung nicht Spline-spezifisch ist, sondern für alle Kantenrouting-Algorithmen entschieden werden muss. Der HyperEdgeDummyMerger, ein Zwischenprozessor von KLay Layered erfüllt genau diese Aufgabe. er wurde jedoch noch nicht in das entwickelte Kantenrouting mit Splines eingebunden.

3.5.1. Vertikale Segmente sortieren

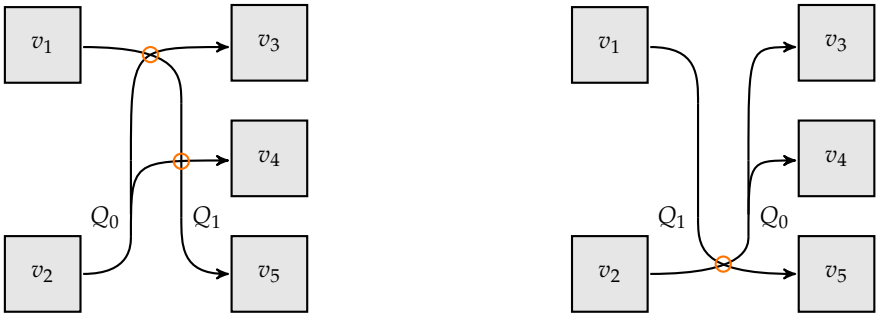
Nachdem wir im letzten Abschnitt Hyperkanten definiert haben, kommen wir nun zu der Aufgabe, die vertikalen Segmente anzuordnen. Hierbei müssen wir die Hyperkanten und die verbliebenen Kanten, die keiner Hyperkante zugeordnet wurden, betrachten. Wie bereits in Abschnitt 3.3.1 erwähnt ist es nötig, diese Segmente nebeneinander anzuordnen, so dass sie sich nicht überlagern. Neben dieser Grundanforderung, stellt sich noch eine Optimierungsaufgabe. Durch eine optimale Anordnung der Elemente kann die Anzahl der entstehenden Kantenkreuzungen minimiert werden, wie in Abbildung 3.15 zu sehen.

Da eine Lösung für diese Aufgabe bereits für das orthogonale Routing entwickelt wurde, kann sie, mit Anpassungen, auch für das Routing mit Splines übernommen werden. Grundlage für die Implementierung waren die Arbeiten von Georg Sander sowie Giuseppe di Battista et al. [San04, DETT99]. Der Algorithmus besteht für jedes Zwischenlayer-Segment aus folgenden Schritten:



(a) Orthogonales Kantenrouting **(b)** Kurvenrouting, Version eins. **(c)** Kurvenrouting, Version zwei.

Abbildung 3.14. Verschiedene Darstellungen von Hyperkanten. Beide Versionen des Routings mit Kurven haben ihre Vorteile. (b) Hat weniger Kanten und eine Zwischenebene weniger. (c) weist dafür eine „anormale“ Kante weniger auf.



(a) Zwei Kreuzungen **(b)** Eine Kreuzung

Abbildung 3.15. Durch eine schlechte Sortierung der Hyperkanten können unnötige Kantenkreuzungen entstehen.

- Erzeugung der Hyperkanten. Dies erfolgt wie in Unterkapitel 3.5 beschrieben. Die verbliebenen, nicht zugeordneten Kanten werden nun ebenfalls als Hyperkante mit einem Eingangs- und einem Ausgangsport behandelt.
- Erzeugung eines *Abhängigkeitsgraphen* der Hyperkanten, in dem die Hyperkanten nun die Knoten darstellen. Zwischen allen Hyperkantenpaaren, die ein kollidierendes

vertikales Segment haben, wird eine Kante gezogen. Die Richtung der Abhängigkeitsbeziehung ist hierbei von der optimaleren Sortierung abhängig. In der in Abbildung 3.15 dargestellten Situation würde eine Abhängigkeitskante von Q_1 nach Q_0 laufen, da wir weniger Kantenkreuzungen erzeugen, wenn Q_0 rechts von Q_1 liegt.

- Elimination aller Zykel. Für den folgenden Schritt ist ein azyklischer Graph notwendig. Es kommt derselbe Algorithmus zum Einsatz, der von KLay Layered in der ersten Phase genutzt wird (siehe Unterkapitel 2.2).

- Erzeugung einer *topologischen Ordnung* auf den Hyperkanten. Dieser Ordnung stellt nun die Ebenenzugehörigkeit der einzelnen Hyperkanten dar.

Gegenüber der Implementierung für das orthogonale Routing sind einige Aspekte zu berücksichtigen.

- Feststellung der vertikalen Segmente. Im orthogonalen Routing ist das vertikale Segment gerade als die Spanne zwischen dem höchsten und niedrigsten Port definiert. Wie in Abbildung 3.2a zu sehen war, ist diese Definition für Splines möglicherweise zu scharf formuliert. Hier ist eine sinnvolle Lösung zu finden.

- Die Definition der Abhängigkeit, also welche Sortierung zweier kollidierender Hyperkanten besser ist, muss getroffen werden. Meistens führen beide möglichen Sortierungen zu Kantenüberschneidungen, wie in Abbildung 3.15 zu sehen ist. Für das Routing mit Splines genügt es die entstehenden Kantenkreuzungen zu zählen. Die zweite Sortierung im Beispiel ist also besser. Im Fall des orthogonalen Routings wäre es zudem noch zu einer Kollision der horizontalen Segmente gekommen, was in die Berechnung der Abhängigkeit einfließen muss. Dieser Fall kann, dank des Spline-Routings, bei uns nicht auftreten.

Kapitel 4

Implementierung

Nachdem wir nun Lösungsansätze für unsere Aufgaben erarbeitet haben, kommen wir nun zur konkreten Implementierung. Wir werden hier nur die finale Implementierung mit B-Splines vorstellen und die im Verlauf der Entwicklung verworfene Beta-Bezier-Implementierung nicht weiter beachten. Abbildung 4.1 zeigt eine Übersicht der wichtigsten implementierten Klassen. Doch bevor wir auf die konkrete Verwendung und Implementierung dieser Klassen eingehen, müssen wir noch ein Detail von KLay Layered betrachten: die *Port Constraints*.

4.1. Entscheidung für B-Splines

Bisher haben wir beide Lösungsansätze eines Kantenroutings, sowohl mit Beta-Bezier-Splines, als auch mit B-Splines parallel verfolgt. In diesem Kapitel werden wir uns jedoch nur noch mit der Implementierung des B-Spline-Routings befassen. Der Entschluss B-Splines für das Kantenrouting zu verwenden war eine Folge der reduzierten lokalen Kontrolle von Beta-Bezier-Splines. Es sei Q ein kubischer, G^2-stetiger Beta-Bezier-Spline, bestehend aus den kubischen Beta-Bezier-Kurven Q_1, Q_2, Q_3. Die Kontrollpunkte der Kurve Q_i seien für $i \in 1, 2, 3$ als v_i^0, v_i^1, v_i^2 und v_i^3 benannt. Wegen der G^2-Stetigkeit des Splines haben einige Kontrollpunkte Einfluss auf die Lage von Kontrollpunkten der

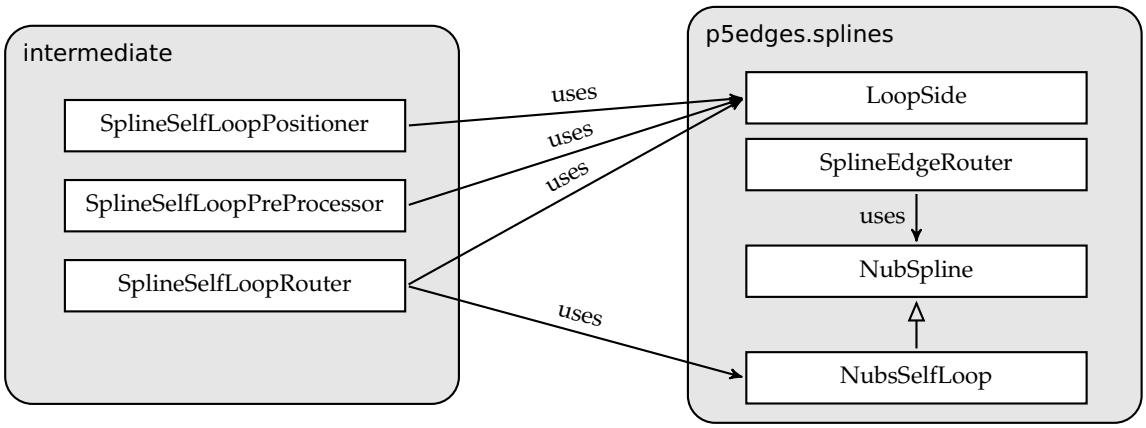


Abbildung 4.1. Die wichtigsten der entwickelten Klassen und ihre Beziehungen. Die von fast allen anderen Klassen genutzten Hilfsklassen *SplinesMath* und *Rectangle* aus dem *p5edges.splines*-Paket werden nicht dargestellt.

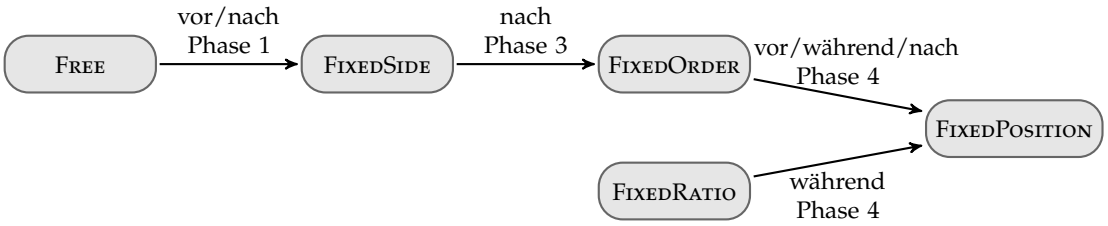


Abbildung 4.2. Übersicht über die *PortConstraints*. Sie können in der Graphendefinition beliebig streng gesetzt werden, nehmen aber im Verlauf des KLay Layered-Algorithmus mindestens die dargestellten Werte an.[1]

folgenden Kurve:

$$v_1^1 \rightarrow v_2^1, v_2^2 \qquad v_2^1 \rightarrow v_3^1, v_3^2$$
$$v_1^2 \rightarrow v_2^1 \qquad v_2^2 \rightarrow v_3^1$$

Diese Einflüsse können mit Hilfe der Formparameter β_1 und β_2 variiert werden. Es ist aber zu erkennen, dass eine Veränderung von v_1^1 einen Einfluss auf die Position von $v_2^1, v_2^2, v_3^1, v_3^2$ hat. Es ist denkbar dies Veränderung von v_1^1 durch entsprechende Anpassung der Formparameter auszugleichen. Es ist jedoch absehbar, dass wir hierfür gerade bei langen Kanten einen großen Rechenaufwand betreiben müssten. Zudem ist es unwahrscheinlich, wurde aber im Laufe dieser Arbeit nicht überprüft, dass jede Veränderung auf diese Art kompensiert werden kann.

B-Splines sind von sich aus entsprechend stetig (abhängig vom Grad des Splines). Ihr Nachteil ist, dass wir nicht wie bei den kubischen Beta-Splines jeden vierten Kontrollpunkt durchlaufen. Somit sind ungewünschte Überschneidungen wahrscheinlicher.

Port Constraints

In KLay Layered kann für jeden Knoten ein *Port Constraint* definiert werden. Folgende Constraints sind definiert:

- Free: Wir haben volle Entscheidungsfreiheit über die Lage der Ports.
- FixedSide: Die Seite jedes Ports steht fest.
- FixedOrder: Die Reihenfolge der Ports darf nicht verändert werden; zudem steht ihre Portseite fest.
- FixedRatio: Das Verhältnis der Positionen der Ports zur Größe ihres Knotens darf nicht verändert werden.
- FixedPosition: Die Position der Ports darf nicht verändert werden.
- Undefined: Wird von uns genau so wie free behandelt.

[1] Übernommen aus [SSVH14].

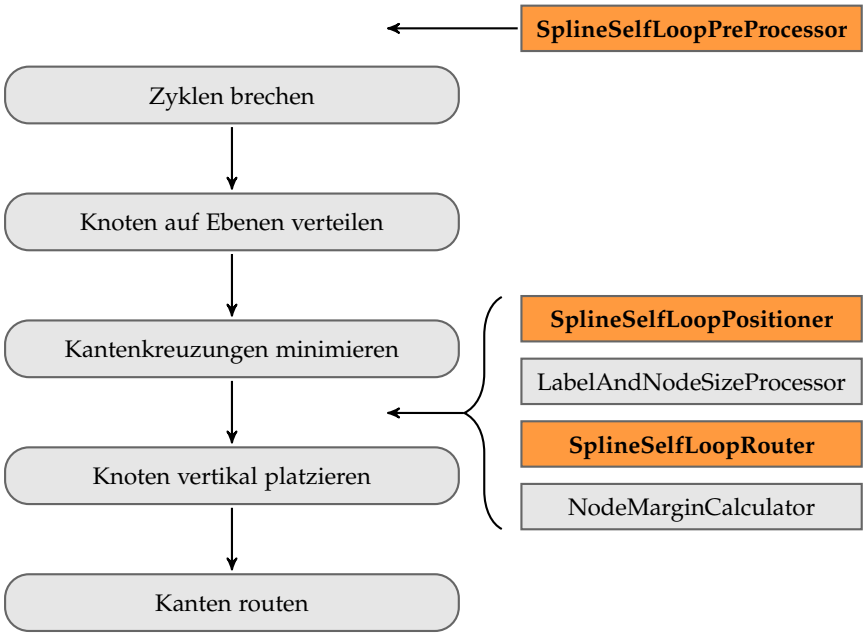


Abbildung 4.3. Für die Selfloops benötigte Zwischenprozessoren

Standardmäßig sind die *Port Constraints* aller Knoten zu Beginn `Free` und durchlaufen während des KLay Layered-Algorithmus die einzelnen Stufen. Abbildung 4.2 stellt dar, wann die Stufe der Constraints angehoben wird. Wichtig ist für uns, dass die Stufe in den einzelnen Phasen auch höher sein kann. Es ist also beispielsweise möglich, dass die Portpositionen eines Graphen von Beginn an feststehen. Dies muss bei der Implementierung beachtet werden.

4.2. Selfloops

Das Routing von Selfloops erfolgt in mehreren Schritten, die in verschiedene Zwischenprozessoren aufgeteilt sind, wie in Abbildung 4.3 zu sehen ist. Es folgt eine kurze Zusammenfassung der einzelnen Prozessoren, bevor wir uns eingehender mit jedem von ihnen befassen.

- *SplineSelfLoopPreProcessor:* Der *SplineSelfLoopPreProcessor* wird noch vor der ersten Phase von KLay Layered ausgeführt. Er erzeugt aus den Selfloops eines Knotens *Zusammenhangskomponenten*, und fügt sie als eine *InternalProperty* dem Knoten hinzu. Die Ports beziehungsweise Kanten werden versteckt.
- *SplineSelfLoopPositioner:* Nachdem in der dritten Phase von KLay Layered die Anzahl der Kantenkreuzungen minimiert wurden, werden die Selfloops vom *SplineSelfLoop-*

Positioner verteilt. Dies erfolgt je nach gewählter Strategie, entweder gleichmäßig um den Knoten verteilt, an der Nordseite aufgereiht oder aufgestapelt. Dabei werden die im *SplineSelfLoopPreProcessor* entfernten Ports und Kanten wieder in den Graphen eingefügt. Dies erfolgt sortiert, so dass die resultierenden Loops optimal angeordnet sind. Nach dieser Phase haben alle Ports eine definierte Portseite und Reihenfolge. Der Port Constraint des Knotens kann also auf FixedOrder erhöht werden.

▷ *SplineSelfLoopRouter:* Hier werden schlussendlich die Biegepunkte der Selfloops berechnet und die Kantenbeschriftungen positioniert. Aus dem für Loops und Beschriftungen resultierenden Platz ergibt sich die für die Selfloops benötigte *NodeMargin*. Sie wird im *NodeMarginCalculator* berücksichtigt, der im Anschluss ausgeführt wird.

Kommen wir nun zu den einzelnen Prozessoren im Detail.

4.2.1. Präprozessor

Der *SplineSelfLoopPreProcessor* ist der erste Zwischenprozessor, der am Routing der SelfLoops beteiligt ist. Er stellt keine weiteren Vorbedingungen an den zu bearbeitenden Graphen. Seine Arbeit besteht aus zwei Teilaufgaben.

Berechnung der Zusammenhangskomponenten Wir bilden für jeden Knoten u wie folgt einen Hilfsgraphen:

$$\begin{aligned} G_S &= (V_S, E_S) \\ V_S &\hat{=} \text{ Menge der Selfloops an Knoten } u \\ (v_1, v_2) \in E_S &\Leftrightarrow \text{Die Selfloops } v_1, v_2 \in V_S \text{ teilen sich (mindestens) einen Port} \end{aligned}$$

In diesem Graphen berechnen wir nun die Zusammenhangskomponenten. Diese Komponenten werden dann in einer *InternalProperty* des Knotens gespeichert. Im Folgenden wird davon ausgegangen, dass alle Selfloops in so einer Komponente gespeichert sind. Die einzelnen Kanten einer Komponente sind in ihr mit einer Ordnung versehen, die sich nach der Größe ihrer Kantenbeschriftung richtet. Zudem werden noch zwei Größen in Abhängigkeit der Kantenbeschriftungen für jede Zusammenhangskomponente berechnet und gespeichert:

- *Gesamte Texthöhe:* Die die Summe der Höhe aller Beschriftungen aller Kanten dieser Komponente.
- *Maximale Textbreite:* Die maximale Breite aller Beschriftungen aller Kanten dieser Komponente.

Diese beiden Werte benötigen wir später, um die Komponenten zu sortieren.

Verstecken gewisser Selfloop-Ports und -Kanten Als zweites werden die Selfloops versteckt. Dies geschieht vorzugsweise durch das Entfernen der Ports, an denen ein

Selfloop liegt. Ist dies nicht möglich, so muss der Loops selbst entfernt werden. Durch das Verstecken der Ports beziehungsweise Kanten können wir sicher sein, dass sie nicht durch andere Phasen von KLay Layered modifiziert werden. Zudem reduzieren wir so die Komplexität einiger Aufgaben.

Ports an denen auch Kanten zu anderen Knoten liegen dürfen nicht entfernt werden. Diese Ports werden für die weiteren Phasen von KLay Layered, insbesondere für die Kreuzungsminimierung, noch benötigt. Wir entfernen einen Port auch dann nicht, wenn der PortConstraint seines Knotens größer oder gleich FixedOrder ist. Diese Ports können wir beim Wiedereinfügen nicht beliebig sortieren, sondern es muss deren Originalordnung erhalten bleiben. Daher wäre ein Entfernen und Wiedereinfügen unnötige Arbeit.

Ist es nicht möglich beide Ports eines Selfloops zu verstecken, so entfernen wir statt dessen die Kante selbst. Hierzu löschen wir sie aus der Kantenliste der beiden mit ihr verbundenen Ports. Eine versteckte Kante vermerken wir als solche in ihrer Zusammenhangskomponente, um sie später wieder einfügen zu können.

4.2.2. Positionierer

Als zweiter Zwischenprozessor schreitet bei der Bearbeitung der Selfloops der *SplineSelfLoopPositioner* zur Tat. Auch sein Aufgabengebiet lässt sich in zwei Komplexe teilen.

Zuweisung der Loop-Seiten Jede Zusammenhangskomponente bekommt eine der in Abbildung 3.8 dargestellten Loop-Seiten zugeordnet. In der Regel bekommen alle Selfloops der Komponente ebenfalls diese Seite zugeteilt. Welche Seite zugeteilt wird, ist vom gewählten Algorithmus abhängig. Im Fall der einseitigen Platzierung ist dies stets Nord (N). Im Fall der gleichmäßigen Platzierung wird nach dem auf Seite 60 in Algorithmus 1 beschriebenen Verfahren vorgegangen. Auch hier erhält jeder Selfloop einer Komponente dieselbe Loop-Seite wie die Komponente.

Sollte eine Komponente mindestens einen Port enthalten, für den bereits eine Portseite feststeht, so wird anders verfahren. Sollten alle bereits in der Komponente definierten Portseiten identisch sein, so erhält die Komponente diese Portseite. Sollten verschiedene Portseiten enthalten sein, so erhält jeder Selfloop individuell eine Loop-Seite zugeteilt: Sind beide Portseiten eines Loops definiert, so ergibt sich hieraus direkt seine Loop-Seite. Sollte bei einem Selfloop genau eine Portseite definiert sein, so wird er zu einem Seitenloop, und der zweite Port erhält die selbe Portseite. Wir verfahren hierbei rekursiv, so dass alle Kanten eine Loop-Seite erhalten. Die Loop-Seite der Komponente wird zu Undefined.

Wiederhinzufügen der Ports und Kanten Die Ports und Kanten, die wir im ersten Schritt entfernt haben, müssen nun wieder eingefügt werden. Für die Ports erfolgt dies sortiert. Hierfür werden die Komponenten der acht Loop-Seiten jeweils sortiert.

Für die Sortierung vergleichen wir zunächst die zuvor definierte Text-Gesamthöhe der Zusammenhangskomponenten, als zweites ihre Text-Maximalbreite. So stellen wir sicher, dass die Komponenten mit den kleinsten Texten nach innen kommen. Da die Kanten in den Komponenten ebenfalls sortiert sind, gilt dies auch für die Kanten einer Komponente. Nun werden die Ports aller Komponenten mit definierter Seite dem Knoten zugefügt.

Als letztes fehlen noch die Ports der Komponenten, deren Seite UNDEFINED ist. Ihre Loops werden außen um alle anderen Loops gelegt, hierbei kann es zu Kantenüberschneidungen kommen.

4.2.3. Selfloops routen

Als letzter Schritt werden die Selfloops vom *SplineSelfLoopRouter* berechnet. Dies ist der rechenintensivste Teil des gesamten Spline-Routings. Dieser Schritt darf erst dann erfolgen, wenn die Knotengröße vom *LabelAndNodeSizeProzessor* berechnet wurde. Er muss erfolgen bevor die Knoten-Margin vom *NodeMarginCalculator* berechnet wird, wie in Abbildung 4.3 dargestellt ist. Der *LabelAndNodeSizeProzessor* ist notwendig, da er die Größe der Knoten bestimmt. Der *NodeMarginCalculator* muss die von uns berechneten Margins in seine Berechnung einfließen lassen.

Die Selfloops werden in aufsteigender *Schrittgröße* berechnet. Mit Schrittgröße bezeichnen wir die Anzahl der Ports, die ein Loop umschließt. So wird sichergestellt, dass die inneren Loops zuerst berechnet werden. Die von einem Loop umschlossenen Ports (inklusive des Start- und Zielports) sind seine *Portspanne*. Bei der Berechnung eines Loops wird seine maximale Höhe festgehalten, und für seine Portspanne gespeichert. Hierbei wird je Portseite unterschieden. Jeder Port muss im Folgenden mindesten eine Höhe erreichen, die dem Maximum aller Portspannen entspricht, die Teilmengen seiner Portspanne sind.

Betrachten wir Abbildung 4.4. Dort sind acht Loops eingetragen. Sie haben folgende Schrittgrößen:

1: a, b, c, d

2: f, g, h

3: e

In welcher Reihenfolge die Elemente einer Gruppen geroutet werden ist nicht definiert. In der Darstellung wurde Kante c vor Kante d berechnet. Daher umschließt d die Portspanne von c, und muss seine Höhe berücksichtigen. Kante e muss auf der Nordseite die Höhen von den Loops a und b, auf der Ostseite nur die Höhen von b berücksichtigen. Die Loops f, g und h werden unabhängig voneinander berechnet.

Beim Routing der Loops müssen zudem ihre Beschriftungen positioniert werden. Bei W- und E-Seitenloops, sowie bei SWN- und SEN-Diametralloops sollen die Beschriftungen außerhalb der Stelle positioniert werden, an der die Kante genau in vertikaler Richtung verläuft. Die Beschriftungen aller anderen Loops erfolgt außerhalb der horizontalen Stelle der Kante. Um diese Position zu Berechnen muss die x- beziehungsweise y-Nullstelle der ersten Ableitung berechnet werden.

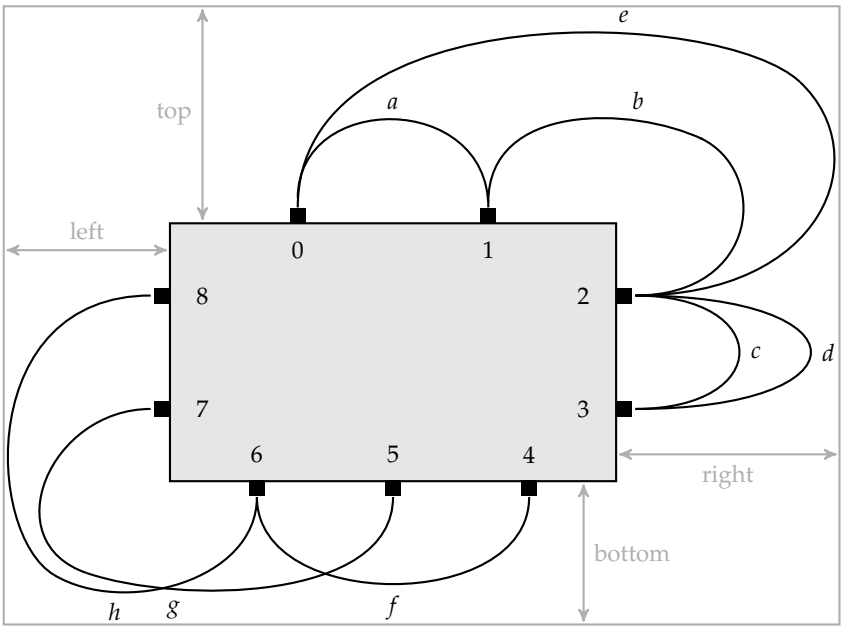


Abbildung 4.4. Routing der Selfloops. Loops mit kleiner Schrittgröße werden zuerst geroutet. Für jeden Loop muss bei seiner Berechnung die Größe aller Loops berücksichtigt werden, die er umfasst. Die Loops in diesem Beispiel bilden übrigens drei Zusammenhangskomponenten: $\{a, b, c, d, e\}$, $\{f, h\}$ und $\{g\}$.

Als letzter Schritt erfolgt nach Berechnung der Splines und der Platzierung ihrer Beschriftungen die Berechnung der sich ergebenden Knoten-Margin. Dazu wird zunächst Rechteck um jeden Selfloop und jede Beschriftung eines Selfloops gelegt. Diese Rechtecke werden vereinigt, und der Abstand zum Knoten berechnet. Die Margin ist in Abbildung 4.4 in grau eingezeichnet.

Bei der Berechnung der Selfloops ist die genaue Position der Knoten noch nicht bekannt, da sie noch nicht platziert sind. Daher müssen wir später die Position des Knotens auf die berechneten Biegepunkte und Beschriftungspositionen addieren. Dies kann geschehen, sobald die Knotenposition feststeht. In dem im folgenden Abschnitt 4.3.1 vorgestellten ersten Lauf der Berechnung der Nicht-Selfloop-Kanten wird dies geschehen.

4.3. Nicht-Selfloop-Kanten

Das Routing von Kanten, die keinen Selfloop formen, erfolgt in zwei Zügen. Zunächst durchlaufen wir alle Layer von links nach rechts. Hierbei werden B-Spline-Kontrollpunkte für jedes Kantensegment berechnet. Im zweiten Lauf werden diese in Bezier-Kontrollpunkte umgerechnet. Da dies nur für einen gesamten Kantenzug geschehen kann, also für eine Kante von ihrem Startknoten über alle Dummy-Knoten hinweg bis zu ihrem Zielknoten, müssen wir uns bereits im ersten Lauf eine Verkettung der Kantensegmente

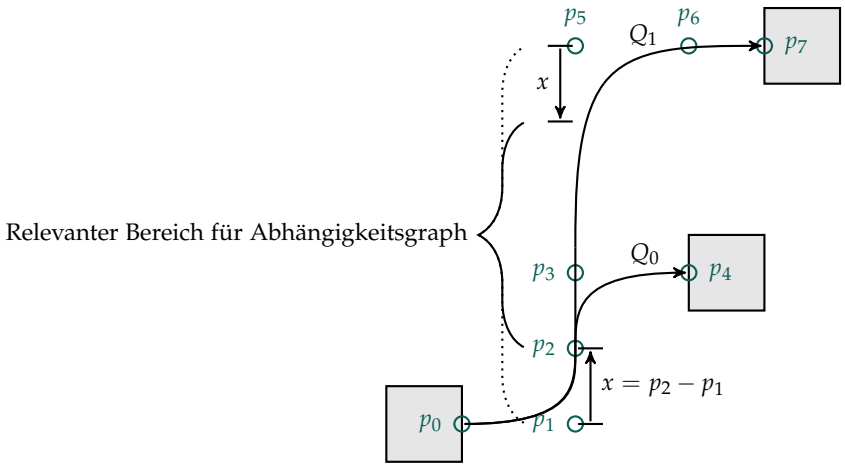


Abbildung 4.5. Die Kontrollpunkte aller Knoten in einer Hyperkante. Der für die Definition des Abhängigkeitsgraphen wichtige Bereich umfasst die äußeren y-Positionen abzüglich eines Anteils der Strecke zwischen p_1 und p_2.

erzeugen. So können wir dann alle Kanten im Graphen, auch wenn sie über mehrere Ebenen verlaufen, wieder rekonstruieren.

4.3.1. Erster Lauf

Im ersten Lauf müssen für alle Segmente zwischen zwei Layern verschiedene Aufgaben erledigt werden. Für den zweiten Lauf müssen wir eine Verknüpfung aller Kanten zu ihren Vorgängern, sowie eine Liste aller Kanten ohne Nachfolgekante bereitstellen.

Wichtigste Aufgabe des ersten Laufs ist es jedoch die Hyperkanten zu erzeugt, und die Kontrollpunkte der sie repräsentierenden B-Splines zu berechnet. Nachdem alle Hyperkanten gebildet sind, kann die topologische Ordnung über sie gebildet werden. Danach wissen wir, wie viele vertikale Segmente benötigt werden. Hieraus kann die endgültige Position der rechten Knoten errechnet werden. Dies muss vor der Berechnung der Kontrollpunkte erfolgen, da die Knotenpositionen für diesen Schritt benötigt werden. Nun kann auch die Position der Selfloops korrigiert werden, wie in Abschnitt 4.2.3 angekündigt. Abschließend werden die B-Spline- Kontrollpunkte der Nicht-Loop-Kanten berechnet.

Betrachten wir das Beispiel aus Abbildung 4.5. Die Hyperkante besteht aus den zwei Kanten Q_0 und Q_1. Sie hat einen Start- und zwei Zielports. Die Kante Q_0 hat die Kontrollpunkte p_0, p_1, p_2, p_3 und p_4. Die Kante Q_1 hat die Kontrollpunkte $p_0, p_1, p_2, p_3, p_5, p_6$ und p_7. Die x-Koordinate von p_1, p_2, p_3, p_5 entspricht der x-Koordinate des der Hyperkante zugeordneten vertikalen Segments. Die Kontrollpunkte p_0, p_4 und p_7 entsprechen gerade den Start- beziehungsweise Endkoordinaten. Der Punkt p_6 wurde eingefügt, da der obere Knoten in seinem Layer nicht linksbündig ausgerichtet ist. Immer wenn ein Knoten nicht bis an die Layergrenze reicht, fügen wir einen solchen Knoten ein. Dies bewahrt uns

davor, dass die Kante einen zu kleinen Radius annimmt und eventuell andere Knoten durchkreuzt. Die Gruppen $\{p_0, p_1\}$, $\{p_3, p_4\}$ und $\{p_5, p_6, p_7\}$ befinden sich jeweils auf gleicher Höhe. Die Höhe von p_2 ergibt sich aus der Mittlung der Höhen des Startpunktes und des ersten Endpunktes. Die ersten vier Kontrollpunkte der beiden Kurven sind identisch. Da wir kubische B-Splines verwenden wird damit auch die erste Bezierkurve der beiden Kanten identisch sein. Da p_2 genau zwischen p_1 und p_3 liegt, verläuft diese Bezierkurve genau bis p_2. Die Kanten trennen sich erst nach diesem Punkt.

Der kritische vertikale Bereich der Hyperkanten, der für die Berechnung der Abhängigkeiten relevant ist, kann angepasst werden. Wie in Abbildung 4.5 dargestellt umfasst er den y-Bereich zwischen dem Einzelport (in der Abbildung der Startport) und dem am weitesten entfernten Port auf der Mehrportseite (in der Abbildung der oberste Zielport). Aufgrund der Krümmung der Kurven kann dieser Bereich verkleinert werden, wodurch die Anzahl der Abhängigkeiten zwischen den Hyperkanten eventuell reduziert wird. dies geschieht mit einem konstanten Faktor der Strecke zwischen dem zweiten und dritten Kontrollpunkt, also zwischen p_1 und p_2. Dieser Faktor wurde experimentell bestimmt.

4.3.2. Zweiter Lauf

Der zweite Lauf dient der Umrechnung der erzeugten B-Spline-Kontrollpunkte in Bezier-Kontrollpunkte. Diese Umrechnung muss jedoch für die Gesamtheit einer Kante erfolgen, auch wenn es sich um eine lange Kante handelt. Daher wurden während des ersten Laufs Informationen für diesen Lauf gesammelt. In der aktuellen Implementierung wird eine Liste von Endkanten, also kurzen Kanten und den letzten Segmenten langer Kanten, sowie eine Verknüpfung jeder Kante zu ihrem Vorgänger gespeichert.

Wir durchlaufen die Liste der Endkanten und sammeln für jede von ihnen alle Kontrollpunkte, auch die ihrer Vorgängerkanten. Aus den Kontrollpunkten können wir nun Bezier Kontrollpunkte errechnen. In welches Kantensegment wir die Bezier Kontrollpunkte speichern ist nicht relevant. Wir können sie also alle im ersten Kantensegment abspeichern. Das Vorgehen „von hinten“ liegt in der Entwicklungsgeschichte der aktuellen Implementierung begründet, ist also beliebig. Ebenso wäre ein Vorgehen „von vorne“ denkbar.

Eine Besonderheit ist bei der Sammlung der B-Spline-Kontrollpunkte zu beachten: falls der Start- oder Zielport ein Nord- oder Südport ist, wurde durch den *NorthSouthPre-Processor* die Kante an einen Dummy-Knoten umgebogen. Der *NorthSouthPostProcessor* würde diesen Knoten in einen Biegepunkt auf unserer Kante umwandeln, die Kante wieder an den Originalport zurücksetzen und den Dummy-Knoten entfernen. Der *Post-Processor* läuft jedoch erst nach unserem Kantenrouting. Zu dem Zeitpunkt haben wir bereits die Bezierkurven errechnet und die Veränderung würde sinnlose Ergebnisse produzieren. Daher müssen wir die Korrektur des Kantenverlaufs selbst übernehmen. Dem *NorthSouthPostProcessor* bleibt dann nur noch die Aufgabe, den Dummy-Knoten zu entfernen.

Wir fügen jeweils einen Glättungskontrollpunkt auf Höhe des Start- und Zielports in

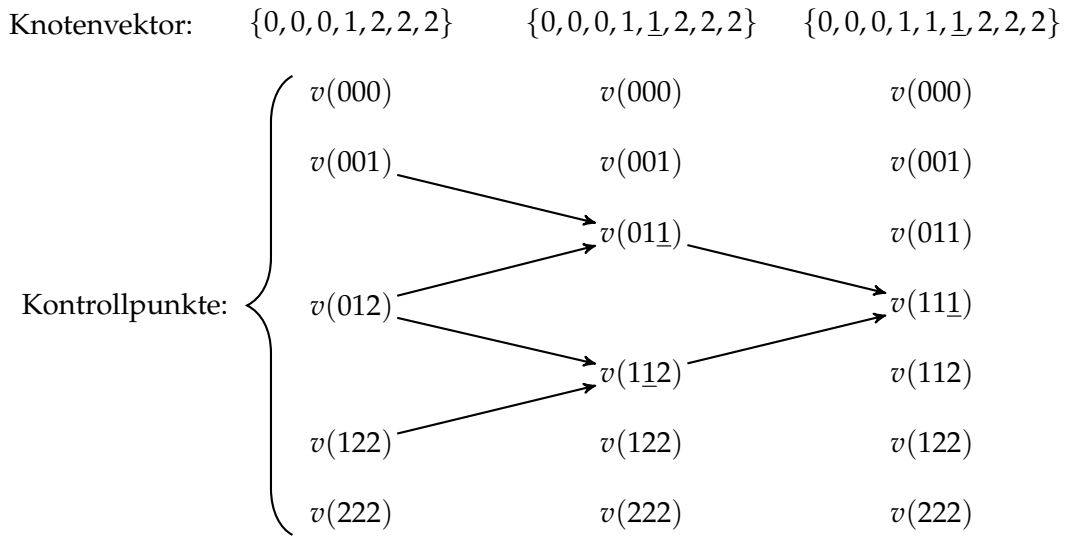


Abbildung 4.6. Berechnung der neuen Kontrollpunkte bei zweimaligem Einfügen des Knotens 1 in den Knotenvektor. Die neuen Knoten sind unterstrichen.

einem festen Abstand zu diesem ein. Dieser dient dazu, dass wir die Kanten tatsächlich horizontal auf die Knoten treffen lassen. Zwar kann man in Abbildung 4.5 sehen, dass sich die Kontrollpunkte p_1, p_3 und p_6 bereits auf derselben Höhe befinden. In ungünstigen Fällen ist der Radius am Kantenende jedoch so klein, dass der erzeugte Pfeil nicht mehr horizontal ausgerichtet wird. Dies wird durch den zusätzlichen Kontrollpunkt verhindert.

Als letzten Zusatzpunkt fügen wir noch einen Kontrollpunkt in die Mitte horizontaler Segmente ein, die sich neben einem Segment mit Krümmung befinden. Solche Segmente entstehen, wenn eine Kante eine Ebene gerade durchläuft. Würden wir den Zusatzpunkt nicht einfügen, so würde der Radius am Beginn des Geraden Segments eventuell zu klein werden. Es steigt dann die Gefahr, dass die Kante durch einen Knoten läuft.

4.4. B-Splines

Die B-Splines sind in der Klasse *NubSpline* und der sie erweiternden Klasse *NubsSelfloop* implementiert. *NubsSelfloop* erweitert die Oberklasse um Funktionen zur Erzeugung von Selfloops und der Berechnung von Beschriftungen an den Loops. Implementiert wurden nicht-uniforme ganzrationale B-Splines. Dies ist notwendig, da KIELER Lightweight Diagrams (KLighD) ganzrationale Bezier-Splines fordert. Die Splines sind nicht-uniform, so dass es uns möglich ist, beliebige Knoten in den Knotenvektor einzufügen. Der Knotenvektor und die Kontrollpunkte sind in einer *LinkedList* abgespeichert. Bei der oft benötigten Operation der Knoteneinfügung ist dieses Datenformat von Vorteil. Die einfügten Kontrollpunkte ergeben sich nach Böhms Algorithmus gerade aus zwei aufeinanderfolgenden Kontrollpunkten, wie in Abbildung 4.6 zu sehen. Wir können also die

Liste bis zu den benötigten Kontrollpunkten durchlaufen und diese Kontrollpunkte dann in Folge auslesen. Der neue Kontrollpunkt muss entweder nach den gelesenen Punkten oder genau zwischen ihnen abgespeichert werden. Aufgrund dieser Datenstruktur ist auch die Entscheidung gefallen, bei der Implementierung der Nullstellensuche die neuen Knoten stets mehrfach, und zwar entsprechend der Dimension Splines, einzufügen. Bei der Mehrfacheinfügung befinden wir uns für den nächsten Knoten bereits an der richtigen Stelle, und ein erneutes Durchlaufen der Datenstruktur entfällt. Für die Umrechnung in Bezier-Splines müssen wir die Liste ebenfalls genau einmal durchlaufen, Womit die verkettete Liste die beste Wahl ist.

Bei der Umrechnung der B-Splines in Bezier-Splines ist noch eine Besonderheit von KLighD zu berücksichtigen. Falls das Kantenrouting als ein Spline-Routing definiert ist, so werden alle Biegepunkte, sowie der Anfangs- und Endpunkt der Kurve von KLighD als Bezier-Kontrollpunkte interpretiert. KLighD sammelt, so denn möglich, die nächsten drei Punkte aus dieser Liste und interpretiert sie zusammen mit der aktuellen Position als Kontrollpolygon einer kubischen Bezierkurve. Sind nur noch zwei Punkte verfügbar, so wird eine quadratische Kurve gebildet, bei einem verbleibenden Punkt dementsprechend eine gerade. Es ist also wichtig, dass wir eine korrekte Anzahl an Punkten in die Liste der Biegepunkte einfügen, so dass unsere Kante wirklich als ein kubischer Bezierkurven-Spline gebildet wird.

Kapitel 5

Leistungsbewertung

In diesem Kapitel werden wir die erzielten Ergebnisse des entwickelten Kantenroutings betrachten. Zunächst werden wir uns bemühen Aussagen über die Performance zu treffen. Im darauf folgenden Kapitel betrachten wir dann tatsächliche Layouts und diskutieren, welche Schwächen das Routing noch aufweist.

5.1. Performance

Zur Evaluierung der Performance wurde Graph Analysis (GrAna), ein von Martin Rieß im Rahmen seiner Bachalor-Arbeit entwickeltes Analysewerkzeug, eingesetzt. Zur Analyse wurden mehrere *Sätze* an Graphen zufällig erzeugt. Mit GrAna wurde dann sowohl Eigenschaften der Graphen, als auch Layoutspezifische Daten erfasst. Gemessen wurden die Anzahl der Knoten, der Kanten, der Selfloops und der Ports. Sowohl für das Orthogonale Kantenrouting, als auch für das entwickelte Spline Routing wurden die Ausführungszeiten der einzelnen Prozessoren, sowie die resultierende Graphengröße notiert. GrAna unterstützt auch weitere Analyen, wie Zählung von Kantenkreuzungen oder Kanten-Kotenüberschneidungen. Hierfür müsste jedoch eine splinespezifische Erweiterung entwickelt werden, da die Überschneidungserkennung für Kurven noch nicht implementiert ist. Dies ist im Rahmen dieser Arbeit nicht geschehen.

Die nicht zu evaluierenden Prozessoren wurden auf den Standardwerten belassen. Das Layout wurde dann für jeden Graphen zehnmal wiederholt, wobei für jeden Prozessor die schnellste Wiederholung gespeichert wurde. Diese Durchläufe wurden mehrfach durchgeführt, um die Ergebnisse miteinander vergleichen zu können. Diese Durchgänge werden wir im Folgenden als *Lauf* bezeichnen. Vor jedem Testdurchgang, der aus jeweils einem Lauf über verschiedene Sätze bestand, wurde eine Warmup-Satz eingefügt. Die erstellten Graphensätze wurden nach folgenden Regeln erzeugt:

▻ Grundeinstellungen

- Selfloops: erlaubt
- Multiedges: erlaubt
- Knotengrößen: variabel von 10-100 (jeweils in der Höhe und der Breite)
- Ports: aktiviert, 20% der Ports werden wiederverwendet

▻ 1. Satz: Variable Knotenzahl

5. Leistungsbewertung

- Knotenzahl: 5-1000
- Kantenzahl: vier ausgehende Kanten je Knoten
- Port Constraints: keine
- Graphen: 250

▻ 2. Satz: Fixe Kotenzahl

- Knotenzahl: 100
- Kantenzahl: 1-4 ausgegende Kanten je Knoten
- Graphen: 100

▻ 3. Satz: Selfloops mit Constraints

- Knotenzahl: 1
- Kantenzahl: 1-45
- Port Constraints: 25% Wahrscheinlichkeit der Portplazierung je Seite
- Graphen: 100

▻ 4. Satz: Selfloops ohne Constraints

- Knotenzahl: 1
- Kantenzahl: 1-45
- Port Constraints: keine
- Graphen: 100

Leider waren die Ergebnisse der Tests zu großen Teilen nicht aussagekräftig. Im folgenden gehen wir auf die einzelnen Testsätze ein, und werden die jeweiligen Ergebnisse betrachten.

Selfloops

Bei den Selfloop-Sätzen variierte die durchschnittliche Laufzeit der Phasen, sowie des gesamten Layouts zwischen den einzelnen Läufen um 20% - 26%. Vermutlich liegen diese großen Abweichungen in den kurzen Laufzeiten von unter einer Millisekunde begründet. Als Tendenz ist zu erkennen, dass das Routing mit Port Constraints etwa doppelt so lange dauert, wie das Routing ohne. In beiden Fällen steigt die Laufzeit linear mit der Kantenzahl. In Abbildung 5.1 sind die summierten Ausführungszeiten der drei wichtigsten Selfloop-Prozessoren für alle vier Läufe abgebildet: dem *SplineSelfLoopPreProcessor*, dem *SplineSelfLoopPositioner* und dem *SplineSelfLoopRouter*. Mit zirka 50% der Gesamtlaufzeit nimmt der *SplineSelfLoopRouter* die meiste Rechenzeit in Anspruch. Ihm folgt der *SplineSelfLoopPreProzessor* mit zirka 15% der Rechenzeit. Unter Berücksichtigung der großen

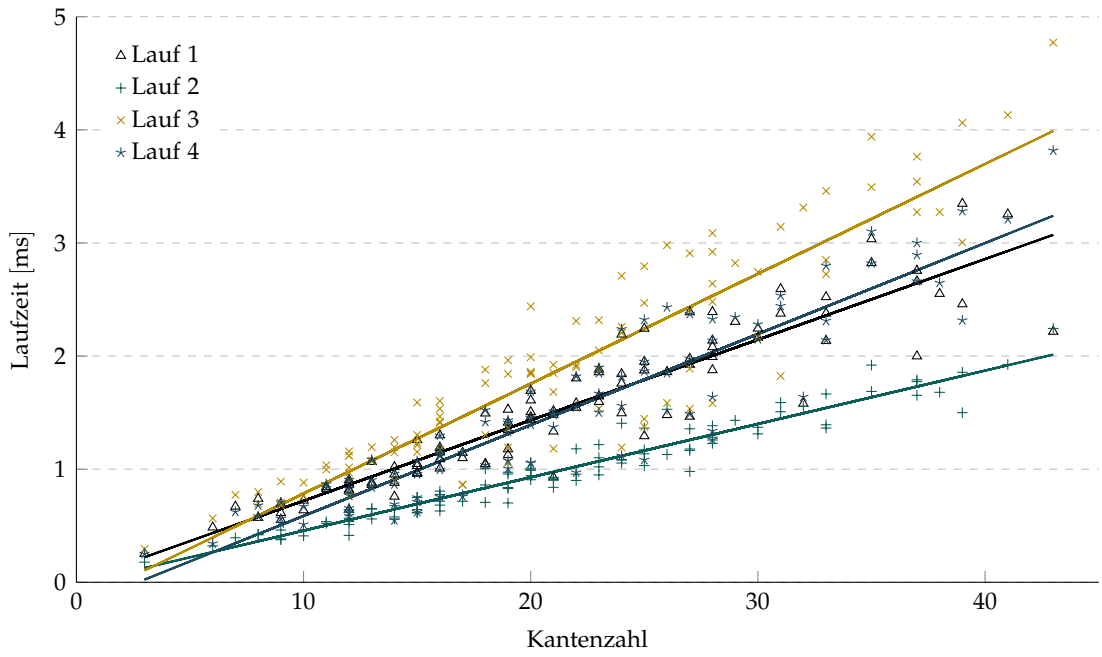


Abbildung 5.1. Vier Läufe des Splineroutings für einen Knoten mit Selfloops. Dargestellt ist die Summe der drei Loop-Prozessoren: *SplineSelfLoopPreProcessor*, *SplineSelfLoopPositioner* und *SplineSelfLoopRouter*.

Abweichungen in der gemessenen Laufzeit kann man sagen, dass das orthogonale- und das Spline-Routing für Selfloops sich bezüglich der Gesamtlaufzeit für alle Prozessoren wohl im selben Rahmen bewegen.

Variable Knotenzahl

Wegen der langen Laufzeit dieses Satzes wurde er nur zwei mal ausgeführt. Es zeigte sich in diesem Satz die geringste Varianz zwischen den Läufen. Sie lag unter 3%. Dies liegt vermutlich an den teilweise langen Laufzeiten zur Berechnung eines Layouts von bis zu 70 Sekunden, siehe hierzu Abbildung 5.2. Somit können wir hier am ehesten einen Vergleich zwischen Spline- und orthogonalem Routing vornehmen. Für die folgende Analyse wurde das Minimum über die beiden Läufe gebildet. Es zeigte sich, dass die Gesamtlaufzeiten bei beiden Routingverfahren in etwa gleich liegen. Zwar nimmt das eigentliche Kantenrouting bei Splines mehr Zeit in Anspruch, dafür ist die Laufzeit anderer Prozessoren verkürzt. Auffällig war hier beispielsweise der *LayerSpeepCrossingMinimizer*. Seine Laufzeit ist beim Spline-Kantenrouting durchschnittlich um 35% kürzer als beim orthogonalen, und dies unabhängig von der Anzahl an Kanten. Die absoluten Laufzeiten dieses Prozessors sind in Abbildung 5.3b dargestellt. In Abbildung 5.3a sind die Summen der Prozessoren dargestellt, die bei den beiden Layoutalgorithmen nur in einem von ihnen vorkommen:

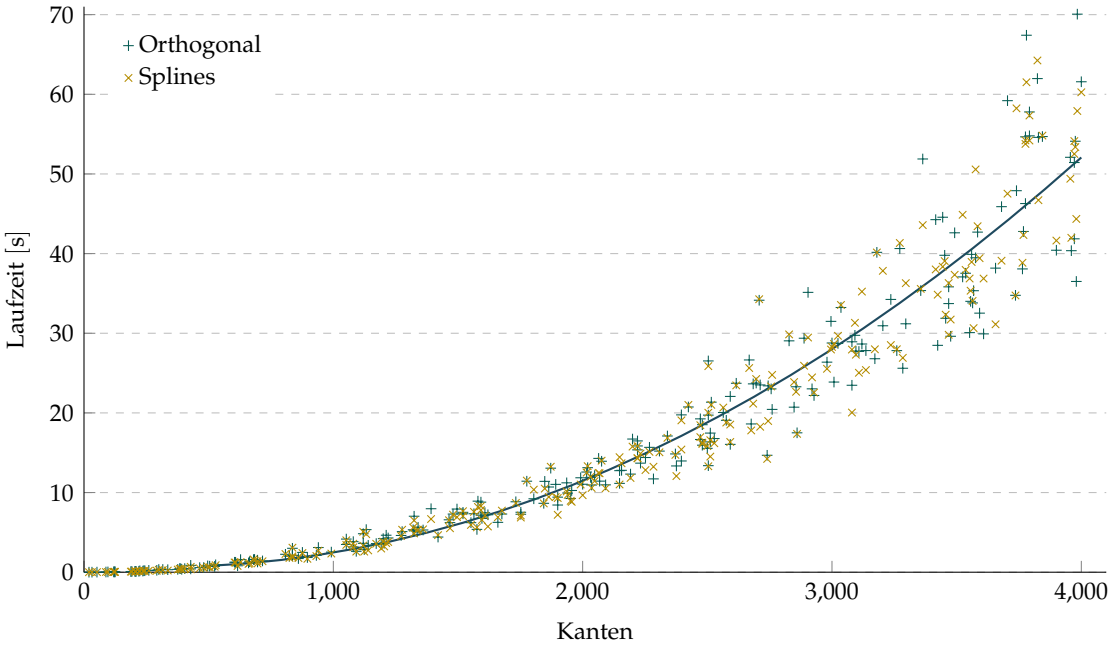
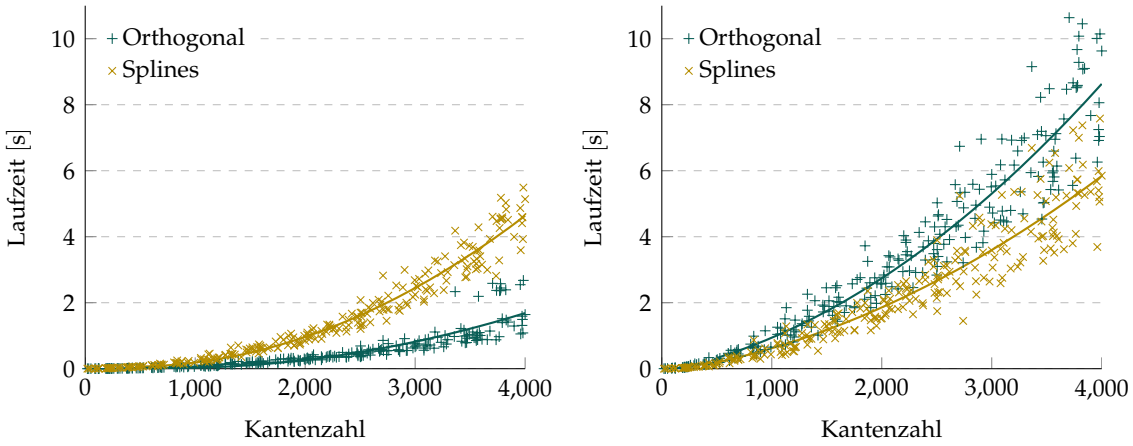


Abbildung 5.2. Die Gesamtlaufzeiten von KLay Layered mit orthogonalem und Spline-Routing, hier für den Graphensatz mit 5-1000 Knoten und jeweils vier Kanten pro Knoten.

Orthogonales Routing	Spline Routing
HyperedgeDummyMerger	*SplineSelfLoopPositioner*
InvertedPortProcessor	*SplineSelfLoopPreProcessor*
OrthogonalEdgeRouter	*SplineEdgeRouter*
SelfLoopProcessor	*SplineSelfLoopRouter*

Feste Knotenzahl

Die Ergebnisse des Satzes mit fester Knotenzahl von 100 zeigten in den drei vorgenommenen Messungen eine Varianz von 10% in der Gesamtlaufzeit. Betrachtet man sehr kurz laufende Prozesoren für sich, so stieg die Varianz auf bis zu 50%. Die zu erkennenden Ergebnisse sind die selben wie beim vorherigen Lauf mit variabler Knotenzahl. Die Gesamtleistung entspricht in etwa dem orthogonalen Routing. Die Spline spezifischen Prozessoren laufen länger als die des orthogonalen Routings. Andere Prozessoren, vor allem der *LayerSpeepCrossingMinimizer* laufen schneller. Als neue Erkenntnis bleibt, dass die Layoutzeit mit den Kanten und nicht mit den Knoten skaliert, was plausibel ist.



(a) Summe der Ausführungszeiten aller layouttypischer Prozessoren.

(b) Laufzeit der *Layer Sweep Crosing Minimisation*

Abbildung 5.3. Durch eine schlechte Sortierung der Hyperkanten können unnötige Kantenkreuzungen entstehen.

5.1.1. Zusammenfassung

Insgesamt zeigt das Routing mit Spline eine bessere Performance als erwartet. Betrachtet man den gesamten Layoutprozess, so erwies es sich insgesamt als dem orthogonalen Routing ebenbürtig. Die genauen Gründe müssten eingehender untersucht werden, folgende Vermutungen liegen jedoch nahe:

- Aufgrund der teilweise großen Abweichungen zwischen zwei eigentlich identischen Läufen, sind die erzielten Ergebnisse mit Vorsicht zu betrachten. Es ist durchaus möglich, dass eine konsistentere Performancemessung schlechtere Ergebnisse für das Spline-Routing liefert.

- Das Spline-Routing unterstützt in seiner jetzigen Form noch nicht alle Besonderheiten, wie dies vom orthogonalen Routing geleistet wird. Zu nennen ist beispielsweise der *HyperedgeDummyMerger*. Durch weitere Verfeinerungen des Layouts ist eine Verschlechterung der Performance zu erwarten.

- Wir haben definiert, dass Ports mit ein- und ausgehenden Kanten nicht unterstützt werden. Diese von uns verbotene Situation trat in den generierten Graphen jedoch auf. Beim Spline-Routing werden solche Kanten bis zu ihrem nächsten Knoten ignoriert und nicht berechnet. Das orthogonale Routing bemüht sich jedoch um ein ordnungsgemäßes Routing.

- Eine designbedingte und erhoffte Leistungsverbesserung anderer Prozessoren resultiert aus der Behandlung von Selfloops. Im orthogonalen Routing wird für jeden Selfloop

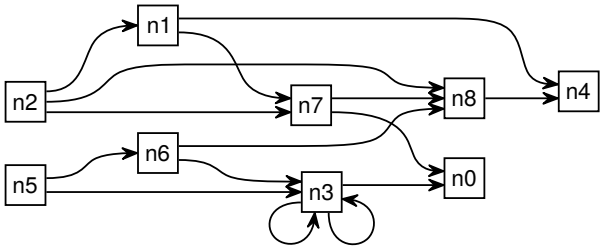


Abbildung 5.4. Der selbe Graph wie er in Unterkapitel 1.4 bereits gezeigt wurde.

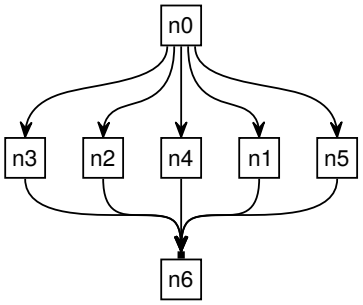


Abbildung 5.5. Angenehme Kantenverläufe, falls mehrere Kanten an einem Knoten liegen.

ein Dummy-Knoten erzeugt. Dies führt zu einer längeren Laufzeit anderer Prozessoren, wie zum Beispiel dem erwähnten *LayerSpeepCrossingMinimizer*.

- In der in Unterkapitel 5.2 folgenden Betrachtung konkreter Layoutergebnisse werden wir Kantenkreuzungen mit Knoten und Beschriftungen finden. Da wir uns gegen ein umgebungsabhängiges Layout entschieden haben, sind solche Kollisionen nicht auszuschließen. Würden wir die Kanten bei der Generierung auf Überschneidungen prüfen, so ließen sich diese Kreuzungen vermeiden. Dies würde jedoch die Performance negativ beeinflussen.

5.2. Bewertung der Ästhetik

Für eine gewissenhafte Bewertung der Ästhetik der erzielten Ergebnisse, wären Daten wie Kantenkreuzungen, Kantenlängen oder die Fläche des erzeugten Graphen notwendig. Diese Daten liegen leider nicht vor. Teilweise sind entsprechende Messmethoden noch nicht für GrAna entwickelt worden, dies gilt etwa für die Kantenlänge oder die Kantenkreuzungen. Für die Graphenfläche wurde in den Evaluierungsläufen, die im letzten Kapitel vorgestellt wurden, Daten gesammelt. Leider erscheinen auch diese teilweise nicht plausibel. Zeigte sich zwischen zwei Läufen des gleichen Satzes eine exakt identische Breite und Höhe des erzeugten Layouts, wich ein weiterer, identischer Lauf um nahezu 100% von den vorherigen Ergebnissen ab. Für einen Graphen wurde diese Abweichung überprüft. In dem für die Laufzeiten recht zuverlässigen Satz mit variabler Knotenzahl ergaben die Größenmessung von GrAna für das Spline-Routing eine Breite, die das 3,7-fache des orthogonalen Routings betrug. Nach dem Export der generierten Layouts ins svg-Format zeigte sich, dass die Breite der beiden Layouts ähnlich waren. Sie wichen um weniger als 4% voneinander ab.

Wir werden uns daher im Folgenden darauf beschränken das Layout einiger Beispielgraphen in Augenschein zu nehmen. Das entwickelte Kantenrouting hat im Layout noch Schwächen, die im Folgenden exemplarisch aufgezeigt werden.

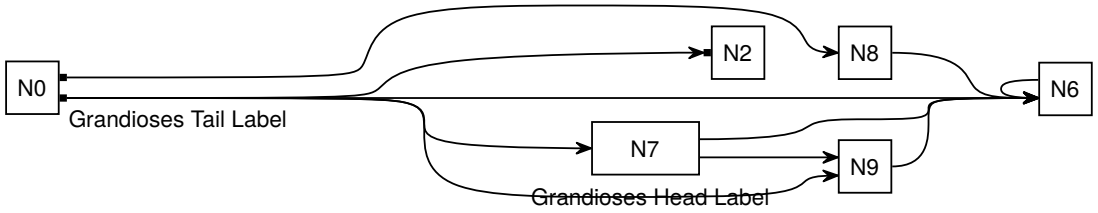


Abbildung 5.6. Beispielgraph. Head Tail label

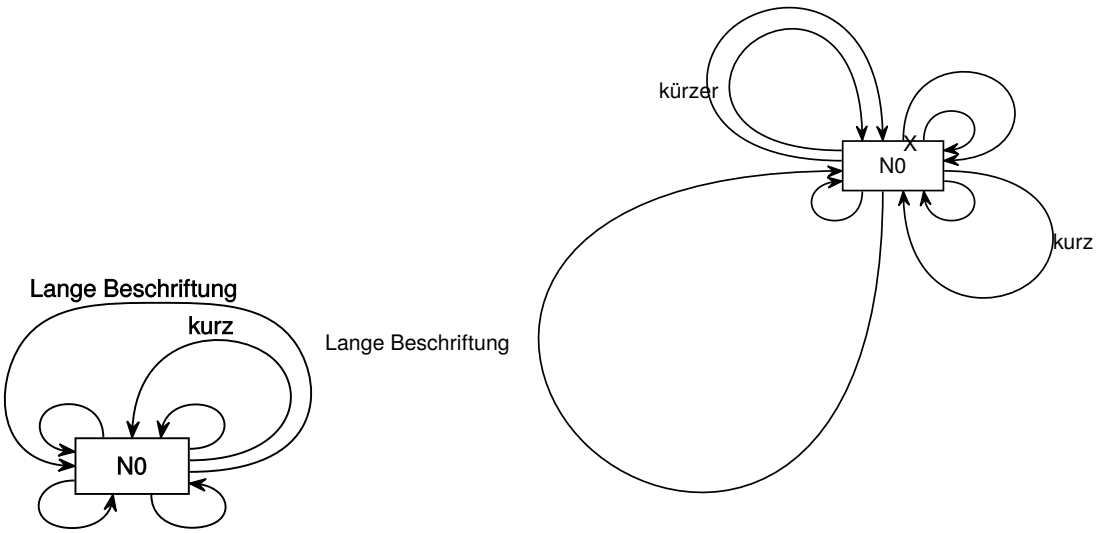
Beginnen wir mit zwei schönen Ergebnissen. In Abbildung 5.4 ist der selbe Graph dargestellt, wie in Unterkapitel 1.4. Gerade die Selfloops werden im entwickelten Routing deutlich besser als von den anderen Algorithmen dargestellt. Die Kantenkreuzungen können wir nicht beeinflussen. In Abbildung 5.5 ist zu erkennen, das die Kanten angenehm symmetrisch zwischen den Knoten verlaufen. Die Kanten, in den Knoten $n6$ hineinlaufen, werden von drei Hyperkanten dargestellt. Eine Kante für die Knoten $n2$ und $n3$, eine für Knoten $n1$ und $n5$ und eine eigene für Knoten $n4$, da diese gerade verläuft. Da die beiden äußeren Hyperkanten unabhängig voneinander sind, kommen sie in das selbe vertikale Segment (welches hier horizontal liegt, da die Layoutrichtung Nord-Süd ist).

In Abbildung 5.6 kann man erkennen, dass wir bei Head- und Tail-Beschriftungen an nicht-Selfloop-Kanten zwar die Portplazierung nicht beeinflussen können, wir können aber die Knoten-Margin vergrößern. So werden Kanten-Beschriftungs-Überschneidungen unwahrscheinlicher. Leider liegt die Head-Beschriftung auf der Kante. Zudem würde die Kante glatter verlaufen, würde wie über den Text hinweglaufen. An Knoten $n6$ hat sich ein Selfloop auf die Westseite zu seinem Zielport gesellt, da dieser eine eingehende Kante von einem anderen Knoten hat.

Abbildung 5.7 zeigt zwei Knoten, an denen nur Selfloops liegen. Im ersten Bild ist die Flussrichtung nach Osten. Das Ergebnis ist durchaus zufriedenstellend. Die Kante mit der Beschriftung „kurz“ könnte eventuell kleiner gezogen werden. Betrachten wir jedoch das zweite Bild, in dem die Flussrichtung gen Süden zeigt, so muss eingestanden werden, dass hier noch Arbeit wartet. Die Ursachen müssen noch untersucht werden. Ein Problem besteht sicherlich darin, dass bei einem Nord-Süd-Verlauf die Beschriftungshöhen und -breiten vertauscht werden müssten, was noch nicht geschieht.

In Abbildung 5.8 sind die bereits in Kapitel 3 theoretisch hergeleiteten Probleme mit Kantenbeschriftungen an Seitenloops dargestellt; hier in der gestapelten Ausführung. Diese Überschneidungen zu eliminieren ist eine schwere Aufgabe. Es ist Wahrscheinlich, dass die Lösung dieses Problems zu Problemen an anderer Stelle führen, beispielsweise bei Nicht-Selfloop-Kanten, die an einen Nord-Port geführt werden.

Auch das fehlerhafte Routing, welches in Abbildung 5.9 dargestellt ist, wurde bereits erörtert. Ports mit ein- und ausgehenden Kanten werden zur Zeit noch nicht unterstützt. Es ist fraglich, ob es einen Anwendungsfall für solche Ports gibt. Gegenwärtig führen sie dazu, dass eine Gruppe der Ports nicht geroutet werden.



(a) West-Ost-Ausrichtung

(b) Nord-Süd-Ausrichtung

Abbildung 5.7. Bei West-Ost Ausrichtung ergibt das verteilte Selfloop-Routing ein ordentliches Ergebnis, bei einer Nord-Süd Ausrichtung werden die Flächen für die Beschriftungen falsch reserviert.

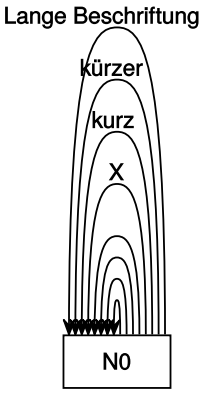


Abbildung 5.8. Bei den gestapelten Seitenloops kann es zu den beschriebenen Überschneidungen kommen. Sie zu verhindern ist keine leichte Aufgabe.

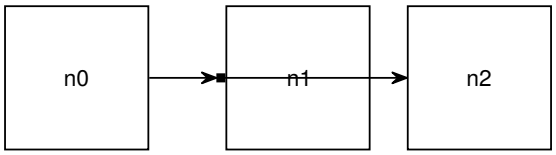


Abbildung 5.9. Sollten an einem Port sowohl ein- wie auch ausgehende Kanten liegen, so wird eine der beiden Kantengruppen nicht geroutet und verläuft gerade zum Gegenport.

Als letztes ist in Abbildung 5.10 noch ein Problem dargestellt, welches wir durch Kollisionsabfragen leicht beheben könnten. Da dies aber teuer ist, sollte darauf verzichtet werden. Im dargestellten Fall ist der Knoten so groß, dass die Höhenvorgaben für den Diametralloop nicht ausreichen, um die Kante um den Knoten herum zu routen. Bei der

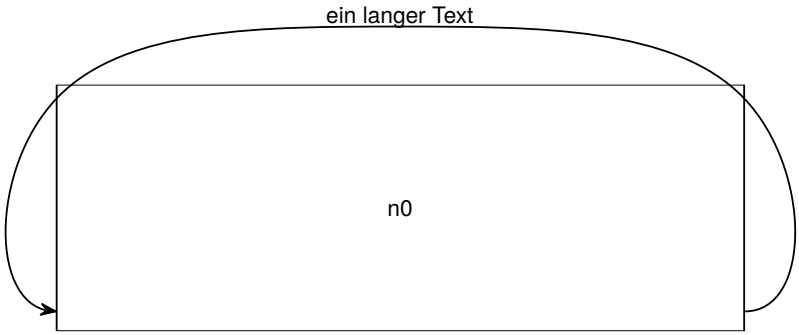


Abbildung 5.10. Bei Diametralloops kann es zu Überschneidungen mit dem eigenen Knoten kommen.

Berechnung der Höhenvorgaben wurde bereits auf die Knotengröße Rücksicht genommen. Genauer: auf den Abstand der Ports zu der Seite, auf der der Loop laufen soll. Eventuell kann hier noch eine bessere Formel gefunden werden, die solche Resultate zuverlässig verhindert.

Kapitel 6

Schlussbetrachtung

6.1. Zusammenfassung

Wir haben ein Kantenrouting mit Splines für ein port- und ebenenbasiertes Graphenlayout entwickelt. Hierbei wurde ausführlich auf die mathematischen Hintergründe der Splines eingegangen. Verschiedene Formen der Stetigkeit wurden vorgestellt und eine Weiterentwicklung der Bezier-Splines zu Beta-Bezier-Splines erörtert. Nach Betrachtung der Alternativen, haben wir uns schließlich für einen Entwurf mit nicht uniformen B-Splines entschieden.

In einem kurzer Abstecher in die Psychologie betrachteten wir mögliche Vorteile des Kantenroutings mit Kurven gegenüber einem Kantenrouting mit Geraden. Wir erarbeiteten einige Leitideen, nach welchen Kriterien wir unsere Splines routen wollen.

Wir haben verschiedene Herausforderung beim Routing betrachtet. Im Gegensatz zu den eingangs vorgestellten Arbeiten, entwarfen wir ein Layout mit Unterstützung für Ports. Auf Selfloops wurde ausführlich eingegangen und verschiedene Layoutansätze für diese speziellen Kanten entworfen. Mit Hyperkanten konnten wir die Anzahl der darzustellenden Kanten reduzieren. Es wurde weitgehend vermieden das Routing umgebungsabhängig zu gestalten. Die unterlassene Kollisionsabfrage mit Knoten oder anderen Kanten führt einerseits zu einer erhöhten Anzahl nicht erwünschter Kollisionen. Andererseits ist auf diesem Weg eine gute Performance erreicht worden.

Schließlich wurde ein konkretes Routing auf Basis von nicht uniformen B-Splines entwickelt, und in das KIELER-Projekt eingebunden. Die entworfene B-Spline Klasse bietet unter anderem Ableitungen, Nullstellensuche und das Einfügen von Knoten. Sie dürfte damit eine gute Basis für weitere Entwicklungen sein. Wir haben uns bemüht die erzielten Ergebnisse zu evaluieren, was jedoch nicht abschließend möglich war. Die Performance des entwickelten Routings scheint jedoch solide zu sein, und die erzeugten Kurven können auch schon jetzt überzeugen.

6.2. Ausblick

Aufbauend auf die hier präsentierten Ergebnisse, bietet sich eine Vielzahl von Ansatzpunkten zur Verbesserung und Weiterentwicklung des Spline-Routings.

Port-Positionierung

Wie in den vorhergehenden Punkten erwähnt, lohnt eine eingehendere Beschäftigung

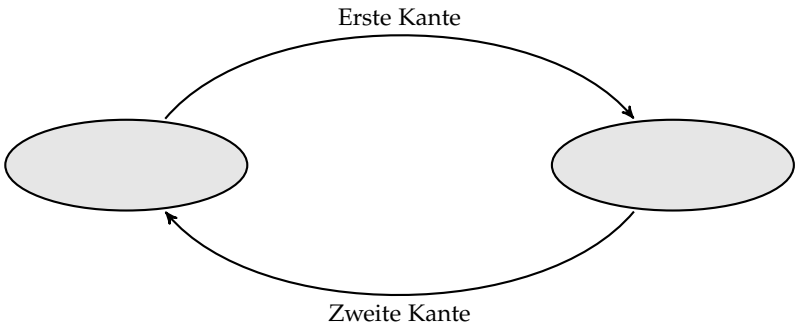


Abbildung 6.1. Idee einer verbesserten und symmetrischen Knotenplazierung für Splines.

mit der Portplazierung. Sie ist gut für Head- und Tail-Beschriftungen, Seitenloops mit Beschriftungen und Kanten an nicht rechtwinkligen Knoten. Bisher werden die Ports gleichmäßig auf den Knotenseiten platziert. Gerade für Selfloops ist eine Optimierung dieser Positionierung denkbar.

Head- und Tail-Beschriftungen an Selfloops
Wie erwähnt wurden diese Beschriftungen außer acht gelassen, da auf die Portplazierung Einfluss genommen werden müsste. Der Ansatz die Margin der Ports zu verändern mag Erfolg versprechen, dies bedarf jedoch einer weiteren Evaluierung.

Seitenloops mit Beschriftungen
Die Problematik der Seitenloops, wie in Abbildung 3.10a dargestellt, bedarf einer näheren Untersuchung, falls mehrere Seitenloops häufiger vorkommen sollten. Auch hier müsste auf die Portplazierung Einfluss genommen werden. Alternativ könnte der Kantenverlauf überarbeitet werden, so dass die Kante eine Kurve um die Beschriftung macht.

Flussrichtung anders als Ost
Das entwickelte Routing lässt sich theoretisch für alle Flussrichtungen einsetzen. Jedoch wurde es vor allem für einen West-Ost-Verlauf entwickelt. Abbildung 5.7b zeigt, dass bei anderen Flussrichtungen noch Nacharbeitungsbedarf besteht.

Kanten an nicht rechteckigen Knoten
Fall die Knotenform nicht rechteckig ist, so ergeben sich gegenwärtig unbefriedigende Anfang- und Endstücke der Kanten an diesen Knoten. Idealerweise würden die Kanten unabhängig von der Knotenform rechtwinklig auf die Knotenkante stoßen.

Portierung der Selfloops zu anderen Routings
Eine Portierung zu anderen Kantenroutings erscheint sinnvoll. So existiert für das Polylinien-Routing noch keine Implementierung für Selfloops, und auch das orthogonale Routing könnte durch ein Selfloop-Routing in der Knotenumgebung profitieren.

Spezielle Knotenplazierung
Die verfügbaren Knotenplazierungsalgorithmen nutzen die Möglichkeiten der Splines nicht aus. Als ein Beispiel sei auf Abbildung 6.1 verwiesen. Durch symmetrische Platzierung der Knoten ließe sich das Layout verbessern. Jedoch muss dann auch im Spline-Routing auf solche Sonderfälle eingegangen werden.

Hierarchische Kanten
Hierarchie-übergreifende Kanten werden noch nicht unterstützt. Eine Implementierung würde die Einsatzmöglichkeiten des Spline-Routings erweitern.

Hyperkanten über Layergrenzen
Der *HyperEdgeDummyMerger* wird zur Zeit noch nicht eingesetzt. Er ermöglicht es Hyperkanten auch über mehrere Layer zu spannen. Das Routing würde hiervon profitieren.

Literaturverzeichnis

[BBB87] Bartels, Richard H., John C. Beatty und Brian A. Barsky: *Introduction to Splines for use in Computer Graphics and Geometric Design*. Morgan Kaufmann, 1987.

[BD89] Barsky, B.A. und T.D. Derose: *Geometric continuity of parametric curves: three equivalent characterizations*. Computer Graphics and Applications, IEEE, 9(6):60–68, 1989.

[BD90] Barsky, B.A. und T.D. Derose: *Geometric continuity of parametric curves: constructions of geometrically continuous splines*. Computer Graphics and Applications, IEEE, 10(1):60–68, Jan 1990, ISSN 0272-1716.

[BH85] Blake, Randolph und Karen Holopigian: *Orientation selectivity in cats and humans assessed by masking*. Vision research, 25(10):1459–1467, 1985.

[BK02] Brandes, Ulrik und Boris Köpf: *Fast and Simple Horizontal Coordinate Assignment*. In: *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, Band 2265 der Reihe *LNCS*, Seiten 33–36. Springer, 2002, ISBN 978-3-540-43309-5.

[Boe80] Boehm, Wolfgang: *Inserting new knots into B-spline curves*. Computer-Aided Design, 12(4):199–201, 1980.

[BP97] Bangert, Claudia und Hartmut Prautzsch: *Circle and Sphere as rational splines*. Neural, Parallel & Scientific Computations, 5:153–162, 1997.

[BRSG07] Bennett, Chris, Jody Ryall, Leo Spalteholz und Amy Gooch: *The Aesthetics of Graph Visualization*. In: *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CAe'07)*, Seiten 57–64. Eurographics Association, 2007.

[CLR80] Cohen, Elaine, Tom Lyche und Richard Riesenfeld: *Discrete< i> B</i>-splines and subdivision techniques in computer-aided geometric design and computer graphics*. Computer graphics and image processing, 14(2):87–111, 1980.

[DETT94] Di Battista, Giuseppe, Peter Eades, Roberto Tamassia und Ioannis G. Tollis: *Algorithms for Drawing Graphs: An Annotated Bibliography*. Computational Geometry: Theory and Applications, 4:235–282, Juni 1994.

[DETT98] Di Battista, Giuseppe, Peter Eades, Roberto Tamassia und Ioannis G. Tollis: *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[DETT99] Di Battista, Giuseppe, Peter Eades, Roberto Tamassia und Ioannis G. Tollis: *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999, ISBN 0-13-301615-3.

[DGKN97] Dobkin, David P, Emden R Gansner, Eleftherios Koutsofios und Stephen C North: *Implementing a general-purpose edge router*. In: *Graph Drawing*, Seiten 262–271. Springer, 1997.

[GJ83] Garey, Michael R. und David S. Johnson: *Crossing Number is NP-Complete*. SIAM Journal on Algebraic and Discrete Methods, 4(3):312–316, 1983.

[GKN02] Gansner, Emden, Eleftherios Koutsofios und Stephen North: *Drawing graphs with dot*. Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, USA, Februar 2002.

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

[GNV88] Gansner, Emden R, Stephen C North und Kiem Phong Vo: *DAG—a program that draws directed graphs*. Software: Practice and Experience, 18(11):1047–1062, 1988.

[Kar72] Karp, Richard M.: *Reducibility Among Combinatorial Problems*. In: Miller, Raymond E. und James W. Thatcher (Herausgeber): *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, Seiten 85–103. Plenum Press, New York, 1972.

[KD10] Klauske, Lars Kristian und Christian Dziobek: *Improving Modeling Usability: Automated Layout Generation for Simulink*. In: *Proceedings of the MathWorks Automotive Conference (MAC'10)*, 2010.

[Mar82] Marr, David: *Vision: A computational investigation into the human representation and processing of visual information*. WH San Francisco: Freeman and Company, 1982.

[MPWG12] Marriott, Kim, Helen Purchase, Michael Wybrow und Cagatay Goncu: *Memorability of Visual Features in Network Diagrams*. IEEE Transactions on Visualization and Computer Graphics, 18(12):2477–2485, Dezember 2012, ISSN 1077-2626.

[MR07] Mørken, Knut und Martin Reimers: *An unconditionally convergent method for computing zeros of splines and polynomials*. Mathematics of Computation, 76(258):845–865, 2007.

[NRL08] Nachmanson, Lev, George Robertson und Bongshin Lee: *Drawing Graphs with GLEE*. In: Hong, Seok Hee, Takao Nishizeki und Wu Quan (Herausgeber): *Graph Drawing*, Band 4875 der Reihe *Lecture Notes in Computer Science*, Seiten 389–394. Springer Berlin Heidelberg, 2008, ISBN 978-3-540-77536-2.

[PD00] Phien, Huynh Ngoc und Nattawit Dejdumrong: *Efficient algorithms for Bézier curves*. Computer Aided Geometric Design, 17(3):247–250, 2000.

[PPP12] Purchase, Helen C., Christopher Pilcher und Beryl Plimmer: *Graph Drawing Aesthetics—Created by Users, Not Algorithms*. IEEE Transactions on Visualization and Computer Graphics, 18(1):81–92, 2012, ISSN 1077-2626.

[Ram89] Ramshaw, Lyle: *Blossoms are polar forms*. Computer Aided Geometric Design, 6(4):323–358, 1989.

[Run01] Runge, Carl: *Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten*. Zeitschrift für Mathematik und Physik, 46(224-243):20, 1901.

[San94] Sander, Georg: *Graph Layout through the VCG Tool*. Technischer Bericht A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, Oktober 1994.

[San04] Sander, Georg: *Layout of Directed Hypergraphs with Orthogonal Hyperedges*. In: *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, Band 2912 der Reihe *LNCS*, Seiten 381–386. Springer, 2004, ISBN 978-3-540-20831-0.

[Sch46] Schoenberg, Isaak Jacob: *Contributions to the problem of approximation of equidistant data by analytic functions.* Quart. Appl. Math., 4:45–99, 112–141, 1946.

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

[Sed05] Sederberg, Thomas W: *An Introduction to B-Spline Curves*. março de, Seiten 01–12, 2005.

[Son01] Sonar, Thomas: *Angewandte Mathematik, Modellbildung und, Informatik*. Springer DE, 2001.

[SSVH14] Schulze, Christoph Daniel, Miro Spönemann und Reinhard Von Hanxleden: *Drawing layered graphs with port constraints*. Journal of Visual Languages & Computing, 25(2):89–106, 2014.

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

[War99] Ware, Colin: *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers Inc., 1999.

[WPCM02] Ware, Colin, Helen Purchase, Linda Colpoys und Matthew McGill: *Cognitive measurements of graph aesthetics*. Information Visualization, 1(2):103–110, 2002, ISSN 1473-8716.