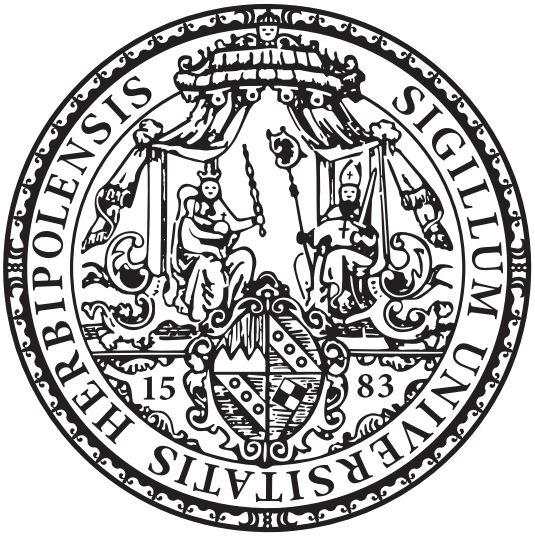
Bachelorarbeit

Generalisierung orthogonal gezeichneter Pläne

Tobias Schopka

Abgabedatum:	14. September 2022
Betreuer:	Prof. Dr. Alexander Wolff
	M. Sc. Tim Hegemann



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Zusammenfassung

Graphen eignen sich zur Modellierung vieler bekannter Netzstrukturen wie zum Beispiel U-Bahnpläne oder Rechnernetze. Zeichnungen, also Visualisierungen dieser Graphen, können je nach Netzstruktur sehr groß und somit für den Nutzer nicht mehr überschaubar sein. Generalisierung beschreibt den Prozess, eine Zeichnung zu vereinfachen mit dem Ziel, Übersichtlichkeit herzustellen.

In dieser Arbeit werden Operationen einer Generalisierung vorgestellt und implementiert. Für die Implementierung wird dabei auf ein bestehendes Framework zurückgegriffen, das für Kabel- und Netzwerkpläne entwickelt wurde. Um die durch die Generalisierung entstehenden Freiräume zu reduzieren, wird zudem ein Stauchalgorithmus entwickelt. In Verbindung mit der Generalisierung lässt sich die Zeichenfläche so, basierend auf 500 Datensätzen, um durchschnittlich 77 % reduzieren. Die Laufzeit bleibt hierbei im zweistelligen Millisekundenbereich.

Abstract

Graphs are suited for modelling various network structures, such as metro-maps or cable-plans. Drawings are visualizations of graphs and can be, depending on the underlying structure, very large and thus incomprehensible for viewers. Generalization is the process of simplifiying a drawing for better readability.

In this thesis different operations for a generalization are presented and implemented. The implemenation is based on an already existing framework, which was developed for network- and cable-plans. Furthermore a shrink-algorithm is developed for reducing the blank areas within the drawing, which result from the generalization. It is shown that, based on 500 sample-data-sets, by average the combination of generalizing and shrinking reduces the required drawing area by 77 %, while the runtime remains in a two-digit millisecond-range.

Inhaltsverzeichnis

1 Einleitung

Durch Graphen lassen sich viele existierende Netzstrukturen, wie beispielsweise Rechnernetze, Kabelpläne und U-Bahnpläne modellieren und mittels einer Zeichnung darstellen. Da diese Strukturen im industriellen Umfeld teilweise sehr groß sind, werden die Zeichnungen entsprechend unübersichtlich und somit für den menschlichen Betrachter schwerer nutzbar. Es ist also eine an den Anwendungsbereich des Betrachters angepasste Abstraktion nötig, die bestimmte Bildinhalte zusammenfasst oder entfernt und somit die Zeichnung vereinfacht. Eine solche Abstraktion wird, angelehnt an den Begriff in der Kartographie, *Generalisierung* genannt. Da eine Generalisierung nicht nur für sehr komplexe, sondern aufgrund der vielen Konfigurationsvarianten einiger technischer Produkte auch für sehr viele unterschiedliche Zeichnungen durchgeführt werden soll, ist dies händisch sehr aufwändig und sollte automatisiert werden.

Durch die Vereinfachung entstehen Freiräume in der Zeichnung, die, um die Zeichnung überschaubar zu machen, reduziert werden sollen. In der Regel sind die Freiräume nach einer Generalisierung ungleichmäßig verteilt und lassen sich somit durch eine reine Skalierung der Knoten, nach der die Bildinhalte nicht überlappen, nicht ausreichend verkleinern. Dieser Umstand macht einen aufwändigeren Stauchalgorithmus erforderlich. Jeweils ein Beispiel für eine Zeichnung, ihre generalisierte Form und ihre generalisierte und gestauchte Form liefert Abbildung 1.1.

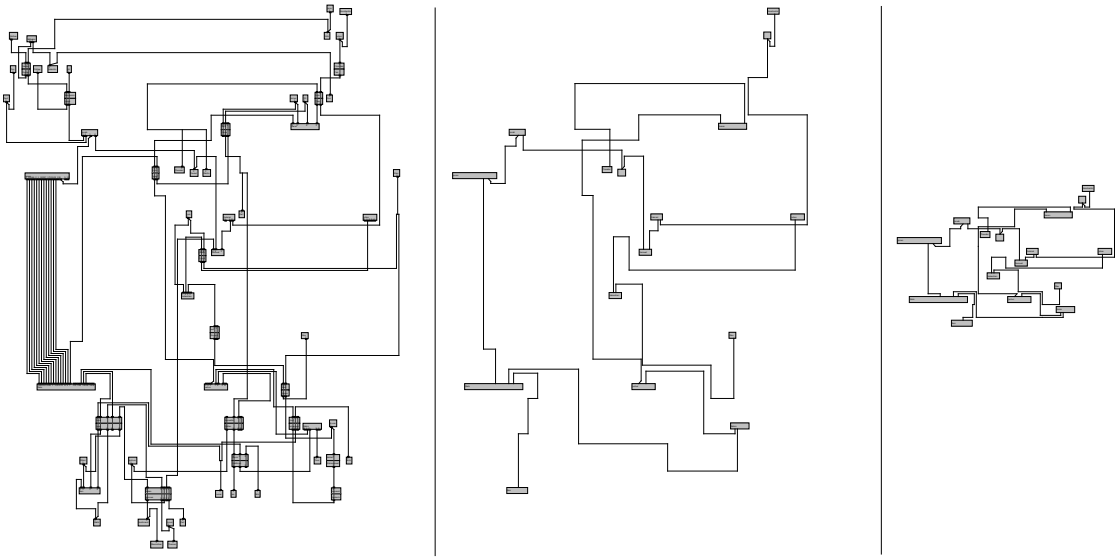


Abb. 1.1: Ursprüngliche Zeichnung (links), generalisierte Zeichnung (Mitte) und generalisierte und gestauchte Zeichnung (rechts)

1.1 Eigener Beitrag

Ziel der Arbeit ist es, die Machbarkeit der Generalisierung und eines anschließenden Stauchens einer orthogonalen Zeichnung zu klären. Die dazu notwendigen Grundlagen werden in Kapitel 2 vermittelt.

Entsprechende Algorithmen zur Generalisierung (siehe Kapitel 3) und zum Stauchen (siehe Kapitel 4) werden aufbauend auf der bereits entwickelten praline-Datenstruktur [pra20a] in Java implementiert und anschließend hinsichtlich ihrer Laufzeit (siehe Abschnitt 5.2) und Platzeinsparung (siehe Abschnitt 5.1) untersucht. Wichtig ist, dass die Zeichnung durch die Algorithmen ästhetisch (siehe Ästhetikkriterien Kapitel 2) und die relative Lage der Knoten *stabil* bleibt. Ein Knoten, der vor der Generalisierung und vor dem Stauchen rechts, links, oberhalb bzw. unterhalb eines anderen Knotens dargestellt war, soll also auch danach rechts, links, oberhalb bzw. unterhalb des anderen Knotens dargestellt werden. In der englischer Literatur trifft man diesbezüglich häufig auf den Begriff *mental-map*, der genau diese durch den Betrachter wahrgenommene Position der Bildinhalte beschreibt. Diese mental-map soll also erhalten bleiben.

Eine Diskussion der Ergebnisse (siehe Kapitel 6) schließt die Arbeit ab und zeigt offene Fragen und potentielle Optimierungen auf.

1.2 Verwandte Arbeiten

Grundlage der Generalisierung und des Stauchalgorithmus sind überlappungsfreie Zeichnungen. Einen Algorithmus zum Entfernen von Knotenüberlappungen liefern Nachmanson et al. [NNB+17]. Hierbei wird aus einer Delaunay-Triangulierung der Knotenmittelpunkte und einer vorgestellten Kostenfunktion ein minimaler Spannbaum gebildet. Ausgehend von einem zufälligen Knoten dieses Spannbaums, der Wurzel, werden dann iterativ die Überlappungen aufgelöst. Einen anderen Ansatz wählen Dwyer et al. [DMS06]. Sie modellieren das Entfernen von Überlappungen als Optimierungsproblem, in dem mit einer Menge an Randbedingungen, den sogenannten *separation-constraints*, Überlappungen verhindert werden und die neuen Positionen der Knoten möglichst nah an den entsprechenden ursprünglichen Positionen liegen sollen.

Da der Begriff Generalisierung aus der Kartographie stammt, gibt es relevante Arbeiten in diesem Bereich. Alan Saalfeld [Saa95] beschreibt die Skalierung einer Landkarte als graphentheoretische Problemstellung, gibt konkrete Beispiele für Operationen, die die Karte vereinfachen und führt Herausforderungen einer geeigneten Generalisierung auf. In der Arbeit von Mackaness und Beard [MB93] werden bestimmte Karteninhalte, wie Flüsse und Seen als Graph modelliert, um dann ausgehend von bekannten Algorithmen der Graphentheorie Regeln für die Generalisierung zu erstellen. So kann zum Beispiel ein zusammenhängendes Verkehrsnetz mit Hilfe des Algorithmus von Kruskal reduziert dargestellt werden.

Eine weitere Art der Generalisierung ist die *Fischaugen-Visualisierung*, angelehnt an die Eigenschaften eines Fischaugen-Objektives. Die Fischaugen-Visualisierung wurde bereits von einigen Arbeiten behandelt [FK96][GKN04]. Grundlegende Idee ist hier-

bei, einen Ausschnitt detailliert und die angrenzenden Bereiche vereinfacht darzustellen. Abello et al. [AKY05] verwenden für die Vereinfachung hierarchische Cluster. Jede Hierachiestufe entspricht dabei einer Abstraktionsstufe. Je nach Entfernung zum detaillierten Ausschnitt kann jedem Teilbereich nun eine Abstraktionsstufe zugeordnet werden.

2 Grundlagen

Grundlegend lässt sich diese Arbeit dem Gebiet des Graphenzeichnens zuordnen. Dazu werden im Folgenden einige grundlegende Begriffe geklärt, die zum Teil in Abbildung 2.1 veranschaulicht werden.

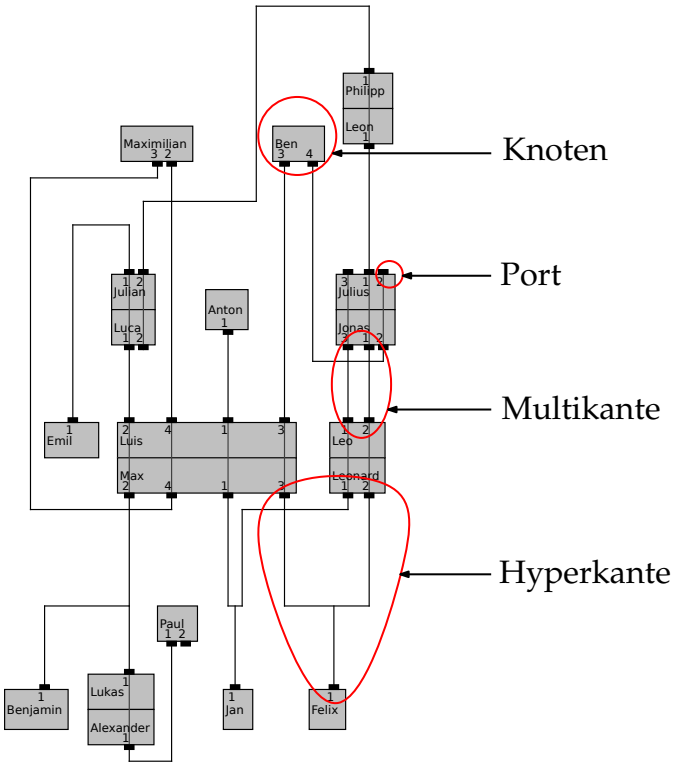


Abb. 2.1: Grundlegende Begriffe

Ein *Graph* $G = (V, E)$ ist eine Struktur, die aus einer Menge von *Knoten* V und einer Menge von Beziehungen zwischen Knoten, den sogenannten *Kanten* E besteht. Besitzen Kanten eine Richtung, spricht man von einem *gerichteten* Graphen. Als *Multikante* bezeichnet man eine Menge an Kanten, die jeweils das selbe Knotenpaar verbinden. Eine *Hyperkante* ist eine Kante, die mehr als zwei Knoten miteinander verbindet.

Eine *Zeichnung* ist eine geometrische Darstellung eines Graphen. Häufig werden hierfür Knoten als Kreise oder Rechtecke und Kanten als Linien dargestellt. In *orthogonalen* Zeichnungen bestehen die Kanten ausschließlich aus horizontalen und vertikalen Segmenten.

Um aus einem Graphen automatisiert eine Zeichnung erstellen zu können, bedarf es einem Zeichenalgorithmus (teils auch Layoutingalgorithmus genannt). Für den Zeichenalgorithmus der praline-Datenstruktur [WZBW20] wurde dazu der Sugiyama-Algorithmus [STT81] unter anderem um *Ports* erweitert. Kanten verbinden hier nicht mehr direkt Knoten sondern Ports miteinander, wobei jeder Port genau einem Knoten zugeordnet ist. Bei dem Sugiyama-Algorithmus handelt es sich um einen *lagebasierten* Zeichenalgorithmus. Hierbei werden die Knoten auf wenigen horizontalen Geraden angeordnet.

Ziel eines Zeichenalgorithmus ist es, eine möglichst ästhetische Zeichnung zu generieren. Hierzu gibt es einige zu berücksichtigende *Ästhetikkriterien*:

- Anzahl der *Kantenkreuzungen*:

  Kantenkreuzungen erschweren das Verfolgen der Kante mit dem Auge, lassen sich unter Umständen nicht oder nur schwer von Hyperkanten unterscheiden und sollten daher vermieden werden.

- Anzahl der *Kantenknicke*:

  Auch Kantenknicke erschweren die Betrachtung des Kantenverlaufs und sollten somit reduziert werden

- *Kantenlänge*:

  Hierbei sollte sowohl die Summe aller Kantenlängen, als auch die maximale Kantenlänge minimiert werden.

- *Zeichenfläche*:

  Die Zeichenfläche sollte möglichst klein sein, um eine überschaubare Zeichnung zu liefern. Insbesondere sollte die Zeichnung wenn möglich komplett auf dem entsprechenden Medium (Bildschirm, DIN-A4-Blatt) abgebildet werden. Unter diesem Aspekt spielt also auch das Seitenverhältnis eine Rolle und sollte an das entsprechende Medium angepasst werden.

- *Lesbarkeit* der Bildinhalte:

  Die letzten beiden Kriterien (Kantenlänge, Zeichenfläche) lassen sich beliebig durch Runterskalieren der Zeichnung verbessern. Dies verschlechtert jedoch die Lesbarkeit der Bildinhalte. Bildinhalte vermitteln Informationen, beispielsweise durch Form, Farbe oder Beschriftungen. Diese Informationen sollen für den Betrachter gut erkennbar sein. Dies kann durch eine ausreichende Linienstärke und große Schriftgröße gewährleistet werden.

Einige Kriterien konkurrieren untereinander, sodass eine gleichzeitige Optimierung aller unmöglich ist. Die Kriterien können allerdings helfen, objektive Vergleiche hinsichtlich der Ästhetik zwischen einer Zeichnung und der entsprechenden generalisierten Zeichnung zu ziehen.

3 Generalisierung

Prinzipiell ist eine Generalisierung abhängig vom betrachteten Anwendungsfall und der verwendeten Datenstruktur. Fragen, beispielsweise ob ein gerichteter Graph vorliegt, ob Multi- oder Hyperkanten vorhanden sein können, müssen für eine geeignete Generalisierung geklärt sein. Dementsprechend lassen sich keine allgemeingültigen Abstraktionsregeln definieren. Das praline-Datenformat wurde ursprünglich für Kabel- und Netzwerkpläne entwickelt, sodass sich die folgenden Operationen gut für ähnliche Anwendungsbereiche übertragen lassen.

In diesem Kapitel wird konkret auf die Umsetzung einer Generalisierung für das praline-Datenformat eingegangen. Ziel ist es, eine Zeichnung zu erstellen, die eine Übersicht darüber gibt, welche Geräte miteinander verbunden sind und dabei Informationen über konkrete Kabel und Anschlussstellen weglässt. Die dazu notwendigen Operationen werden nachfolgend erklärt, definiert und einige Informationen zur Implementierung gegeben.

Um eine formale Definition der Operation liefern zu können, muss jedoch zuerst auf den Aufbau eines Graphen der praline-Struktur eingegangen werden, da dieser komplexer ist, als der eines gewöhnlichen Graphen, wie er in Kapitel 2 beschrieben ist. Hier ist ein Graph ein 6-Tupel:

$$\text{Graph } G = (V, E, VG, PP, P, T)$$

mit:

$$\text{Knotenmenge } V \subseteq \mathcal{P}(P) \times T$$
$$\text{Knotentypmenge } T = \{\textsc{splice}, \textsc{plug}, \textsc{other}\}$$
$$\text{Kantenmenge } E \subseteq \mathcal{P}(P)$$
$$\text{Portpaarmenge } PP \subseteq \binom{P}{2}$$
$$\text{Knotengruppenmenge } VG \subseteq \binom{V}{2} \times \mathcal{P}(PP)$$
$$\text{Portmenge } P \subseteq V \times \mathcal{P}(E)$$

Ein Knoten besteht also aus einer Menge von Ports und einem Typ. Die Knoten aus V partitionieren die Ports aus P, dh. jeder Port ist in genau einem Knoten enthalten. Zwei Knoten können zu einer Knotengruppe zusammengefasst werden, jedoch muss nicht jeder Knoten einer Knotengruppe angehören. Die Kanten aus E verbinden mindestens zwei Ports aus P. E ist eine Multimenge, somit können mehr als ein Exemplar einer Kante enthalten sein. Ports gehören zu genau einem Knoten und enthalten beliebig viele

Kanten aus E. Zwei Ports, deren beiden zugehörigen Knoten einer Knotengruppe angehören, lassen sich zu einem Portpaar aus PP zusammenfassen.

Zusätzlich werden zwei grundlegende Funktionen definiert:

remove(x, y): Entfernt das Element x aus der Menge y
add(x, y): Fügt das Element x der Menge y hinzu

3.1 Stecker entfernen

In dieser Operation sollen die Stecker entfernt, die Kanten die durch den Stecker verlaufen jedoch erhalten bleiben. Ein Beispiel dazu liefert Abbildung 3.1.

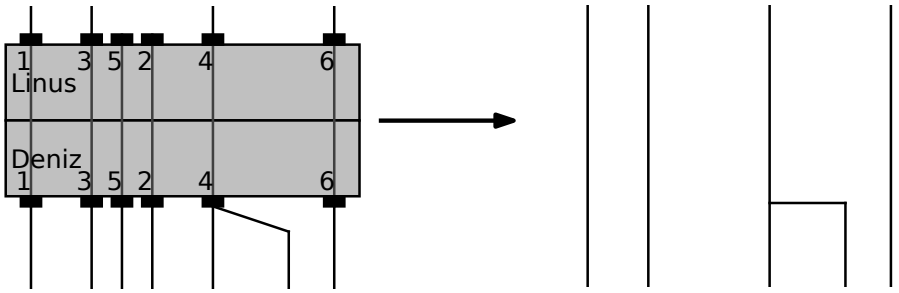


Abb. 3.1: Stecker (links) und die Kanten nachdem der Stecker entfernt wurde (rechts)

Ein *Stecker* bezeichnet im Folgenden eine Knotengruppe aus zwei Knoten des Typs PLUG und soll eine Stecker/Buchsen-Kombination darstellen. Um einen Stecker zu entfernen, müssen zuerst die Kanten von gegenüberliegenden Ports je nach Fall entfernt (Regel 2) oder angepasst (Regel 1, Regel 4) werden. Solche gegenüberliegenden Ports bilden zusammen ein Portpaar, das wiederum zu einer Knotengruppe gehört. Nachdem die Kanten eines Portpaares angepasst/entfernt wurden, wird das Portpaar entfernt. Besitzt ein Stecker kein Portpaar mehr, so kann auch er schließlich entfernt werden. Um nun alle Stecker eines Graphen zu entfernen, werden die folgenden Regeln erschöpfend angewendet:

Regel 1 Sackgassen in Hyperkanten

In Worten:
Hat genau ein Port p_1 eines Portpaars $pp = \{p_1, p_2\}$ einer Steckergruppe vg genau eine Kante e und ist diese eine Hyperkante und hat p_2 keine Kante, so wird der Port p_1 aus der Hyperkante entfernt. Die Ports p_1 und p_2 und das Portpaar pp werden anschließend entfernt.

Formal:
$(vg \in VG) \ \wedge \ (v \in vg[0]) \ \wedge \ (v[1] = \text{PLUG}) \ \wedge \ (pp \in vg[1]) \ \wedge \ (p_1 \in pp) \ \wedge \ (p_2 \in pp) \ \wedge \ (p_1 \neq p_2) \ \wedge \ (|p_1[1]| = 1) \ \wedge \ (e \in p_1[1]) \ \wedge \ (|e| > 2) \ \wedge \ (p_2[1] = \varnothing) \rightarrow \text{remove}(p_1, e) + \text{remove}(p_1, P) + \text{remove}(p_2, P) + \text{remove}(pp, vg) + \text{remove}(pp, PP)$

Regel 2 Sackgasse mit normaler Kante

In Worten:
Hat genau ein Port p_1 eines Portpaars $pp = \{p_1, p_2\}$ einer Steckergruppe vg genau eine Kante e und ist diese keine Hyperkante und hat p_2 keine Kante, so wird sie entfernt. Die Ports p_1 und p_2 und das Portpaar pp werden anschließend entfernt.

Formal:
$(vg \in VG) \ \wedge \ (v \in vg[0]) \ \wedge \ (v[1] = \text{PLUG}) \ \wedge \ (pp \in vg[1]) \ \wedge \ (p_1 \in pp) \ \wedge \ (p_2 \in pp) \ \wedge \ (p_1 \neq p_2) \ \wedge \ (|p_1[1]| = 1) \ \wedge \ (e \in p_1[1]) \ \wedge \ (|e| < 3) \ \wedge \ (p_3 \in e) \ \wedge \ (p_3 \neq p_1) \ \wedge \ (p_2[1] = \varnothing) \rightarrow \text{remove}(e, p_3[1]) + \text{remove}(e, E) + \text{remove}(p_1, P) + \text{remove}(p_2, P) + \text{remove}(pp, vg) + \text{remove}(pp, PP)$

Diese Regel lässt sich beispielsweise bei den Portpaaren 2 und 5 von Abbildung 3.1 anwenden

Regel 3 Verwaiste Ports

In Worten:
Hat kein Port eines Portpaars $pp = \{p_1, p_2\}$ einer Steckergruppe vg eine Kante, so werden die Ports p_1 und p_2 und das Portpaar pp entfernt.

Formal:
$(vg \in VG) \ \wedge \ (v \in vg[0]) \ \wedge \ (v[1] = \text{PLUG}) \ \wedge \ (pp \in vg[1]) \ \wedge \ (p_1 \in pp) \ \wedge \ (p_2 \in pp) \ \wedge \ (p_1 \neq p_2) \ \wedge \ (p_1[1] = \varnothing) \ \wedge \ (e \in p_1[1]) \ \wedge \ (p_2[1] = \varnothing) \rightarrow \text{remove}(p_1, P) + \text{remove}(p_2, P) + \text{remove}(pp, vg) + \text{remove}(pp, PP)$

Regel 4 Portpaare zu (Hyper-)Kanten

In Worten:
Haben die Ports p_1 und p_2 eines Portpaars $pp = \{p_1, p_2\}$ einer Steckergruppe vg zusammen mehr als eine Kante, so werden die Kanten entfernt und stattdessen eine neue (Hyper-)Kante e_{new} hinzugefügt die alle Ports der alten Kanten beinhaltet, außer den beiden Ports p_1 und p_2. Die Ports p_1 und p_2 und das Portpaar pp werden anschließend entfernt.

Formal:
$(vg \in VG) \ \wedge \ (v \in vg[0]) \ \wedge \ (v[1] = \text{PLUG}) \ \wedge \ (pp \in vg[1]) \ \wedge \ (p_1 \in pp) \ \wedge \ (p_2 \in pp) \ \wedge \ (p_1 \neq p_2) \ \wedge \ (|p_1[1]| + |p_2[1]| > 1) \rightarrow \text{remove}(p_1, P) + \text{remove}(p_2, P) + \text{remove}(pp, vg) + \text{remove}(pp, PP) + \text{add}(e_{new}, E) + (\forall e \in (p_1[1] \cup p_2[1]) : (\forall p \in e \backslash \{p_1, p_2\} : \text{add}(e_{new}, p[1]) + \text{add}(p, e_{new})) + \text{remove}(e, E))$

Diese Regel lässt sich beispielsweise bei dem Portpaar 4 von Abbildung 3.1 anwenden.

Regel 5 Leere Stecker

In Worten:
Besitzt eine Steckergruppe vg kein Portpaar mehr, so wird vg inklusive ihrer beiden Knoten v_1 und v_2 entfernt.

Formal:
$(vg \in VG) \ \wedge \ (v_1 \in vg[0]) \ \wedge \ (v_2 \in vg[0]) \ \wedge \ (v_1 \neq v_2) \ \wedge \ (v_1[1] = \text{PLUG}) \ \wedge \ (vg[1] = \varnothing) \rightarrow \text{remove}(vg, VG) + \text{remove}(v_1, V) + \text{remove}(v_2, V)$

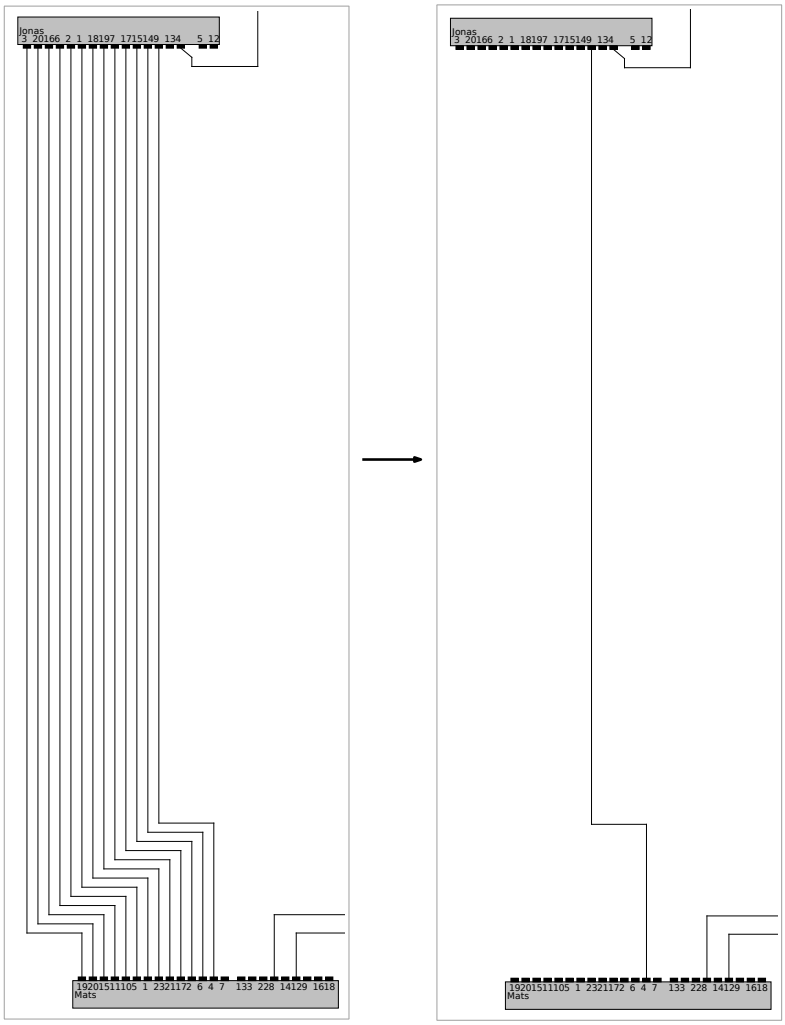


Abb. 3.2: Ausschnitt einer Zeichnung vor (links) und nach (rechts) dem Entfernen der Multikanten

3.2 Multikanten entfernen

Die grundlegende Idee zum Entfernen einer Multikante ist einfach: Verbinden mehrere Kanten das selbe Knotenpaar, so werden alle bis auf eine entfernt. Im einfachsten Fall beinhaltet die betrachtete Multikante, wie in Abbildung 3.2, ausschließlich normale Kanten, also keine Hyperkanten.

Hyperkanten lassen sich dabei allerdings nicht ohne Weiteres entfernen, wie Abbildung 3.3 veranschaulicht.

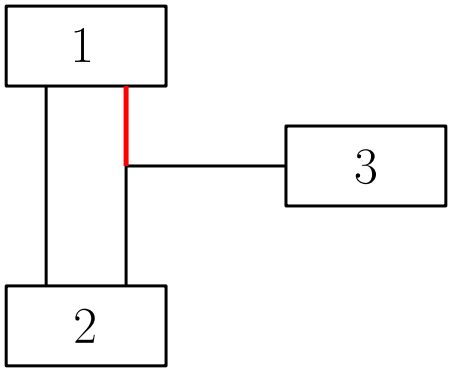


Abb. 3.3: Zeichnung mit einer Multikante, die eine Hyperkante enthält

Offensichtlich gibt es zwei Kanten, die den Knoten 1 und den Knoten 2 verbinden. Entfernt man nun die Hyperkante, oder auch nur das rote Segment, so entfernen man die einzige Verbindung zwischen Knoten 1 und Knoten 3.

Um nun eine Hyperkante e aus einer Multikante entfernen zu können, muss die Multikante eine weitere Hyperkante besitzen, die mindestens alle Knoten verbindet, die auch e verbindet. In Abbildung 3.4 verbindet die blaue Hyperkante alle Knoten, die die rote Hyperkante verbindet. Die rote Hyperkante kann somit entfernt werden.

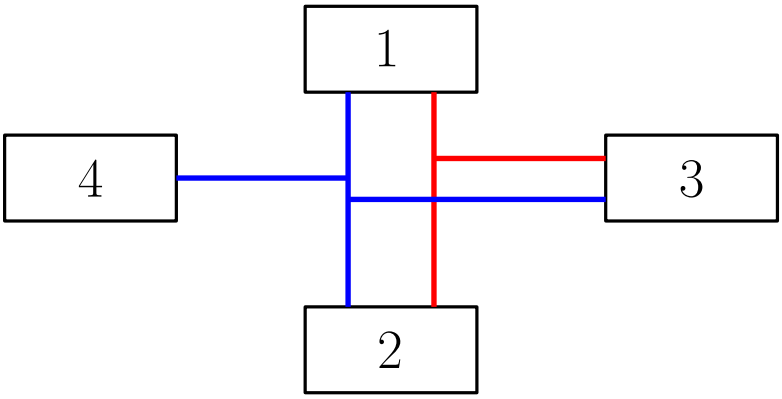


Abb. 3.4: Zeichnung mit einer Multikante, die zwei Hyperkanten enthält

An dieser Stelle muss auf noch eine Besonderheit der praline-Datenstruktur eingegangen werden, den sogenannten *Splices*. Knoten des Types SPLICE sind aus logischer Sicht keine Knoten, sondern vielmehr eine Schnittstelle innerhalb einer Kante, an die beliebig viele weitere Kanten angeschlossen werden können. Splices bieten also eine alternative Möglichkeit, Hyperkanten zu realisieren. Die drei blauen normalen Kanten aus Abbildung 3.5 sind in Verbindung mit dem Splice also logisch äquivalent zur roten Hyperkante.

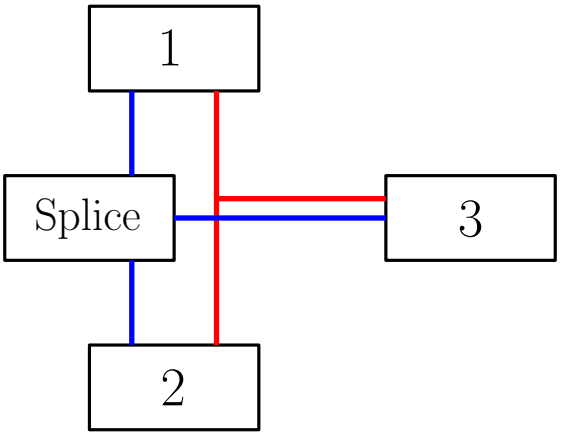


Abb. 3.5: Darstellung einer Splice-Kanten-Kombination (blau), die logisch äquivalent zur roten Hyperkante ist

Splices müssen also anders als Knoten anderen Typs behandelt werden. Entfernt man eine der beiden Kanten die in Abbildung 3.6 von dem Splice ausgehen, wie man es bei einem Knoten eines anderen Typs tun würde, so verbindet die Kombination aus Splice und Kante keine normalen Knoten mehr und ist somit unzulässig. Die Kombination aus Splice und beiden Kanten ist hingegen äquivalent zu seiner Selbstkante von Knoten 1 zu Knoten 1. Ausgehen von einem Splice müssen also Multikanten, die mehr als drei Kanten beinhalten auf eine Multikante mit zwei Kanten reduziert werden.

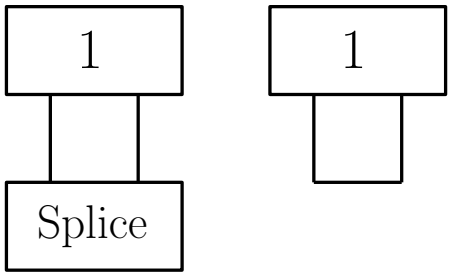


Abb. 3.6: Die Kombination aus Splice und Multikante (links) ist äquivalent zu einer Selbstkante (rechts)

Angenommen ein normaler Knoten hat eine Kante zu einem Splice, dann ist nun zu klären, mit welchen anderen normalen Knoten er logisch verbunden ist. Dazu ist in Abbildung 3.7 ein anschauliches Beispiel abgebildet.

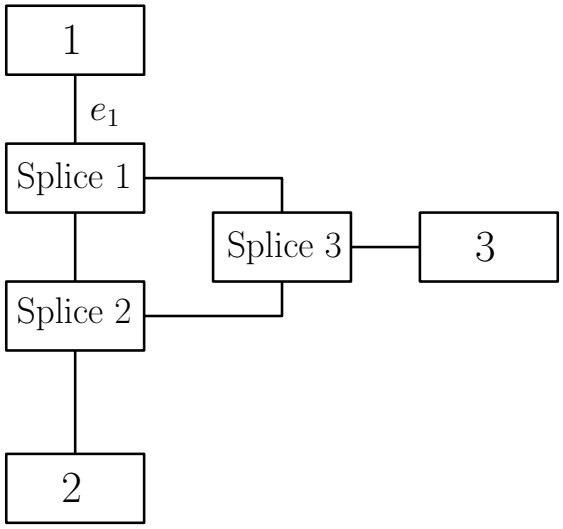


Abb. 3.7: Abbildung einer Kante mit mehreren Splices in Reihe

Prinzipiell funktioniert das gewünschte Verfahren ähnlich wie eine Tiefensuche. Ausgehend vom Knoten 1 erreichen man mit der Kante e_1 Splice 1. Über eine der beiden anderen Kanten von Splice 1, also die Kanten die ungleich e_1 sind, wird nun Splice 2 entdeckt. Im nächsten Schritt wird dann Knoten 2 und Splice 3 entdeckt. Schließlich wird Knoten 3 entdeckt. Um eine Endlosschleife zu verhindern müssen sich bereits besuchte Knoten gemerkt werden. Im Folgenden ist der Pseudocode eines solchen Algorithmus dargestellt.

Algorithmus 1:
VertexList getAdjacentVerticies(Edge e, Port p)

```
1  adjacentVerticies = new VertexList
2  alreadyVisited = new VertexList
3  alreadyVisited.add(p.getVertex())
4  foreach destinationPort ∈ e.getPorts() do
5      if destinationPort ≠ p then
6          if destinationPort.getVertex().getType() ≠ SPLICE then
7              adjacentVerticies.add(destinationPort.getVertex())
8              alreadyVisited.add(destinationPort.getVertex())
9          else
10             destinationVertex=destinationPort.getVertex()
11             adjacentVerticies.addAll(spliceDFS(destinationVertex,alreadyVisited))
12 return adjacentVerticies
```

Algorithmus 2:
VertexList spliceDFS(Vertex *currentVertex*, VertexList *alreadyVisited*)

```
1  adjacentVerticies = new VertexList
2  alreadyVisited.add(currentVertex)
3  foreach port ∈ currentVertex.getPorts() do
4      foreach edge ∈ port.getPorts() do
5          foreach destinationPort ∈ edge.getPorts() do
6              if !alreadyVisited.contains(destinationPort.getVertex()) then
7                  if destinationPort.getVertex().getType() ≠ SPLICE then
8                      adjacentVerticies.add(destinationPort.getVertex())
9                      alreadyVisited.add(destinationPort.getVertex())
10                 else
11                     destinationVertex=destinationPort.getVertex()
12                     adjacentVerticies.addAll(spliceDFS(destinationVertex,alreadyVisited))
13 return adjacentVerticies
```

Mithilfe der Funktion *getAdjacentVerticies* müssen nun folgende Regeln erschöpfend angewendet werden, um die gewünschten Multikanten zu entfernen:

Regel 1 Redundante (Hyper-)Kanten

In Worten:
Sei e_1 eine Kante, die den Port p_1 enthält, der sich wiederum dem Knoten v zuordnen lässt und sei e_2 eine andere Kante, die den Port p_2 (p_1, p_2 können, aber müssen nicht verschieden sein) enthält, der sich auch dem Knoten v zuordnen lässt. Bei v handelt es sich dabei um *keinen* Splice. Ist die auf e_1 und p_1 angewendete Funktion getAdjacentVerticies eine Teilmenge der auf e_2 und p_2 angewendeten Funktion getAdjacentVerticies, so wird e_1 entfernt.
Formal:
$(v \in V) \wedge (v[1] \neq \text{SPLICE}) \wedge (p_1 \in v[0]) \wedge (p_2 \in v[0]) \wedge (e_1 \in p_1[1]) \wedge (e_2 \in p_2[1]) \wedge (e_1 \neq e_2) \wedge (\text{getAdjacentVerticies}(e_1, p_1) \subseteq \text{getAdjacentVerticies}(e_2, p_2)) \rightarrow (\forall p_e \in e : \text{remove}(e, p_e[1])) + \text{remove}(e, E)$

Regel 2 Redundante Kanten an Splices

In Worten:
Seien e_1, e_2, e_3 drei paarweise verschiedene Kanten, die passend zu ihrem Index den Port p_1, p_2 oder p_3 (p_1, p_2, p_3 können, aber müssen nicht verschieden sein) enthalten. Die Ports p_1, p_2, p_3 lassen sich alle dem Knoten v zuordnen. Bei v handelt es sich um einen Splice. Ist die auf e_1 und p_1 angewendete Funktion *getAdjacentVerticies* eine Teilmenge der auf e_2 und p_2 angewendeten Funktion *getAdjacentVerticies* und eine Teilmenge der auf e_3 und p_e angewendeten Funktion *getAdjacentVerticies*, so wird e_1 entfernt.

Formal:
$(v \in V) \wedge (v[1] = \text{SPLICE}) \wedge (p_1 \in v[0]) \wedge (p_2 \in v[0]) \wedge (p_3 \in v[0]) \wedge (e_1 \in p_1[1]) \wedge (e_2 \in p_2[1]) \wedge (e_3 \in p_3[1]) \wedge (e_1 \neq e_2) \wedge (e_1 \neq e_3) \wedge (e_2 \neq e_3) \wedge (\text{getAdjacentVerticies}(e_1, p_1) \subseteq \text{getAdjacentVerticies}(e_2, p_2)) \wedge (\text{getAdjacentVerticies}(e_1, p_1) \subseteq \text{getAdjacentVerticies}(e_3, p_3)) \rightarrow (\forall p_e \in e : \text{remove}(e, p_e[1])) + \text{remove}(e, E)$

3.3 Alleinstehende Komponenten entfernen

Durch vorherige Operationen kann es sein, dass Knoten keine inzidenten Kanten mehr besitzen, wie zum Beispiel Knoten „Jakob“ in Abbildung 3.8. Solche Knoten sind unter Umständen unerwünscht und sollen entfernt werden.

Ebenso unerwünscht sind Splices mit nur einer inzidenten Kante, da diese Kante logisch gesehen keine normalen Knoten verbindet. Somit muss in solch einem Fall sowohl der Splice als auch die inzidente Kante entfernt werden. Ein passendes Beispiel ist der Splice ganz unten auf Abbildung 3.8.

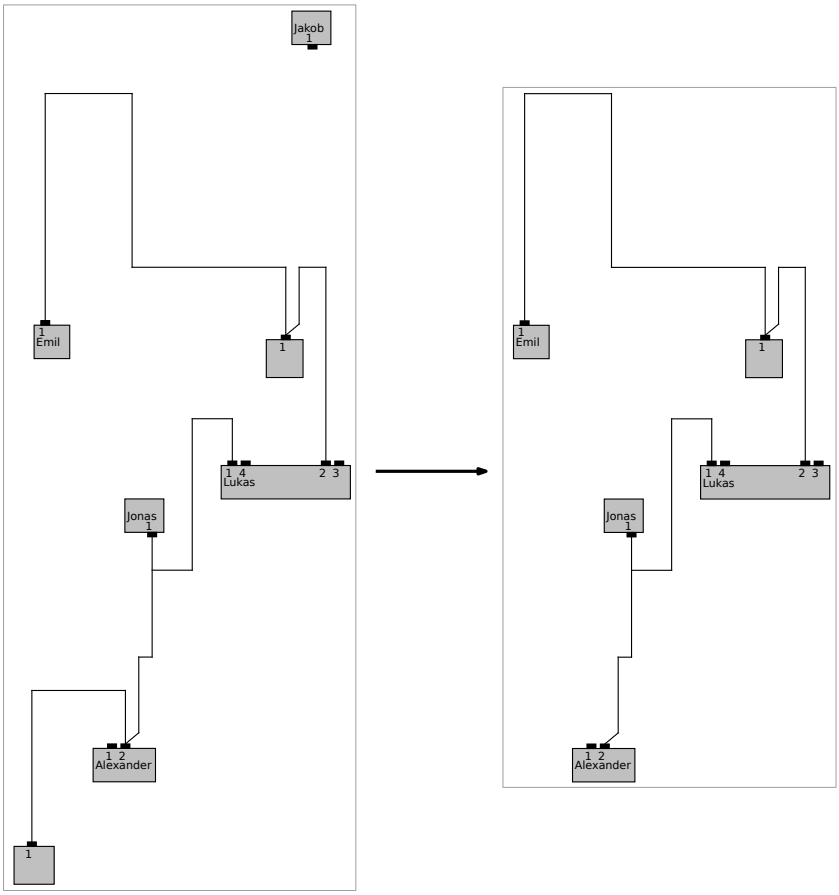


Abb. 3.8: Beispielzeichnungen bevor (links) und nachdem (rechts) alleinstehende Komponenten entfernt wurden

Um die unerwünschten Knoten zu entfernen, müssen die folgenden Regeln erschöpfend angewendet werden:

Regel 1

In Worten:
Enthalten alle Ports eines Knotens v keine Kanten, so wird dieser entfernt.
Formal:
$(v \in V) \ \wedge \ (p_i \in v[0]) \ \wedge \ ((\bigcup_{i=1}^{|v|} p_i[1]) = \varnothing) \rightarrow \text{remove}(v, V) + (\forall p \in v[0] : \text{remove}(p, P))$

Regel 2

In Worten:
Enthalten alle Ports eines Splices v in Summe eine Kante e, so werden v und e entfernt.
Formal:
$(v \in V) \ \wedge \ (v[1] = \text{SPLICE}) \ \wedge \ (p_i \in v[0]) \ \wedge \ ((\sum_{i=1}^{|v|} |p_i[1]|) = 1) \rightarrow \text{remove}(v, V) + (\forall p \in v[0] : (\forall e \in p[1] : (\forall p_e \in e : \text{remove}(e, p_e[1])) + \text{remove}(e, E)) + \text{remove}(p, P))$

3.4 Ports entfernen

Nachdem Multikanten entfernt wurden (siehe Abschnitt 3.2) liefern Ports in unserer gewünschten Zeichnung keine sinnvollen Informationen mehr und können entfernt werden. Abbildung 3.9 zeigt das gewünschte Ergebnis dieser Operation.

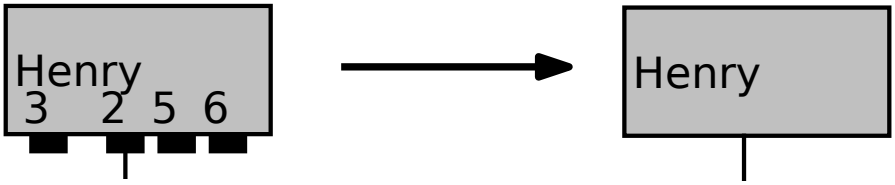


Abb. 3.9: Knoten bevor (links) und nachdem (rechts) die Ports entfernt wurden

Eine Möglichkeit dies zu erreichen erfordert eine neue Datenstruktur, in der eine Kante nicht eine Menge von Ports sondern eine Menge von Knoten ist, also:
Neue Kantenmenge $E' \subseteq \mathcal{P}(V)$
Um nun die alte Kantenmenge in die neue Kantenmenge zu transformieren, muss die folgende Regel erschöpfend angewendet werden:

In Worten:
Ersetze jede Kante e durch eine neue Kante e', die die Knoten aller Ports von e beinhaltet.
Formal:
$e \in E \rightarrow (\forall p \in e : \text{add}(p[0], e')) + \text{add}(e', E') + \mathit{delete}(e, E)$

Hinweise zur Implementierung
Bei der Implementierung wurden die Ports nicht entfernt und auch keine neue Datenstruktur erstellt, da alle nach dieser Operation angewendeten Algorithmen, wie beispielsweise der Stauchalgorithmus (Kapitel 4), darauf angepasst werden müssten. Stattdessen werden die Ports nur ausgeblendet, indem beide Seitenlängen jeweils auf 0 gesetzt werden. Zu beachten ist, dass nach dem Ausblenden der Ports die Kanten so verlängert werden müssen, dass sie wieder an die Knoten anschließen.

4 Stauchalgorithmus

Grundsätzliches Ziel des Stauchalgorithmus ist, die Zeichenfläche zu reduzieren. Als positiver Nebeneffekt werden auch einige Kanten gekürzt. Ein solcher Stauchvorgang kann nach einer Generalisierung sein ganzes Potential entfalten, da nun Freiräume in der Zeichnung vorhanden sind. Er lässt sich allerdings auch auf nicht-generalisierte Zeichnungen anwenden.

In diesem Kapitel wird ein Stauchalgorithmus für die praline-Datenstruktur vorgestellt. Anforderungen hierzu waren, dass der Algorithmus die relative Lage der Knoten stabil hält und dass er die Ästhetik der Zeichnung nicht verschlechtert. Kanten bestehen, geometrischen gesehen, in der praline-Struktur aus einem Startpunkt, einem Endpunkt und einer Liste aus Knickpunkten. Zwei aufeinander folgende Punkte bilden dabei ein *Segment*, das je nach Ausrichtung entweder horizontal oder vertikal ist.

Der Algorithmus staucht einmal in horizontaler und einmal in vertikaler Richtung. Das Stauchen lässt sich jedoch auch für beide Richtungen unabhängig voneinander anwenden. Es wird nachfolgend das Stauchen in vertikaler Richtung gezeigt. Das Stauchen in horizontaler Richtung funktioniert prinzipiell analog dazu. Alternativ lässt sich für das selbe Ergebnis die Zeichnung nach dem vertikalen Stauchen um 90° drehen, erneut vertikal stauchen und schließlich wieder zurückdrehen. Ein Stauchvorgang in vertikaler Richtung lässt sich hierbei grob in drei Teile unterteilen:

1. Vertikale Lücken zwischen Knoten suchen
2. Horizontale Segmente innerhalb der Lücken verschieben
3. Lücken minimieren

Diese drei Teile werden nun einzeln detailliert betrachtet.

4.1 Teil 1: Vertikale Lücken zwischen Knoten suchen

Ziel dieses Teils ist, vertikale Intervalle innerhalb der Zeichnung zu finden, in denen keine Knoten abgebildet sind. Grundsätzliche Idee ist, dass nach einer Minimierung dieser Intervalle (siehe Abschnitt 4.3) die relative Lage der Knoten wie gefordert stabil bleibt und die Zeichenfläche (und einige Kantenlängen) reduziert werden.

Da in Abschnitt 4.2 die Kanten nicht direkt, sondern in einem sinnvollen Abstand an den entsprechenden Knoten geschoben werden, muss dieser Abstand bereits für diesen Teil definiert werden, um die knotenfreien Lücken zu suchen. Dieser Abstand wird *Begrenzung* genannt. Ein Knoten hat jeweils eine Begrenzung für alle vier Seiten des Knotens, also eine rechte, linke, obere und untere Begrenzung. Die Begrenzung eines

Knotens hängt davon ab, ob der Knoten an der entsprechenden Seite Ports besitzt und wie viele Kanten diese Ports besitzen. Die drei unterschiedlichen Fälle werden in Abbildung 4.1 dargestellt.

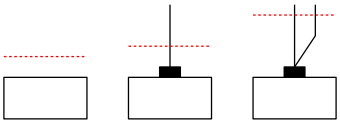


Abb. 4.1: Obere Begrenzung (rot gestrichelt) eines Knotens ohne Ports (links), mit Ports mit jeweils maximal einer Kante (Mitte) und mit mehreren Kanten innerhalb eines Ports (rechts)

Zur Initialisierung dieses Teils wird eine Liste aller Knoten erstellt. Die Knoten werden dann anhand ihrer unteren Begrenzung von unten nach oben sortiert. Das erste Element der Liste ist also der Knoten, dessen untere Begrenzung die unterste Begrenzung innerhalb der Zeichnung ist. Nun wird jeweils die bisher größte obere Begrenzung, begonnen mit der oberen Begrenzung des ersten Elements, mit der unteren Begrenzung des nächsten Knotens der Liste miteinander verglichen. Ist die obere Begrenzung unterhalb der unteren Begrenzung des folgenden Knotens, so liegt eine der zu suchenden Lücken vor. Das untere Ende des Lückenintervalls wird durch die bisher maximale obere Begrenzung und das obere Ende des Lückenintervalls durch die untere Begrenzung des folgenden Knotens bestimmt. Als Vorbereitung für Abschnitt 4.3 werden alle Knoten bis zur letzten Entdeckung einer Lücke in eine Liste (V_i in Abbildung 4.2) gepackt. Das gleiche gilt für die Kantenpunkte in diesem Intervall (P_i in Abbildung 4.2). Dieses Prozedere wird so lange durchgeführt, bis alle Knoten durchgegangen wurden.

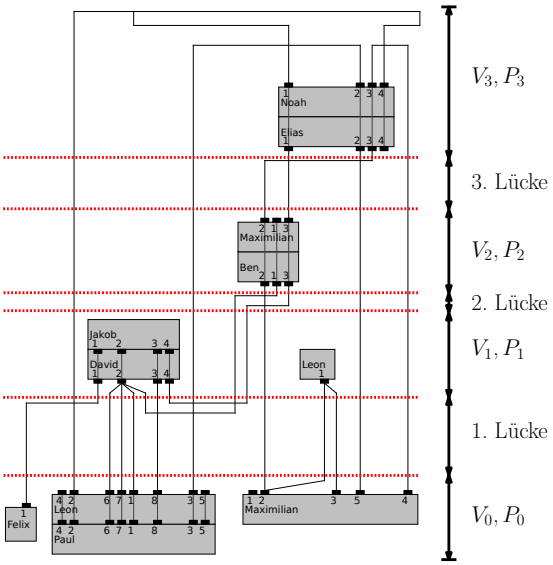


Abb. 4.2: Ziel des ersten Teils des Stauchalgorithmus: Die vertikalen Lücken wurden bestimmt. Die Knoten- und Kantenpunktemenge wurde als Vorbereitung in Gruppen unterteilt

Der Stauchalgorithmus ist wie folgt aufgebaut:

Algorithmus 3:
shrinkDrawing(Graph *graph*)

```
1  vertexList = Liste aller Knoten von graph
2  Sortiere VertexList entsprechend ihrer unteren Begrenzung von unten nach oben
3  allVertexIntervalls = new List
4  allEdgePointIntervalls = new List
5  allSegmentIntervalls = new List
6  allVerticalDisplacements = new List
7  i = 0
8  while i < vertexList.size() do
9      vertexIntervall = new List
10     vertexIntervall.add(vertexList.get(i))
11     int maxTopBoundary = topBoundary(vertexList.get(i))
12     i++
13     while i < vertexList.size() do
14         if maxTopBoundary ≥ bottomBoundary(vertexList.get(i)) then
15             if maxTopBoundary ≤ topBoundary(vertexList.get(i)) then
16                 maxTopBoundary = topBoundary(vertexList.get(i))
17             vertexIntervall.add(vertexList.get(i))
18             i++
19         else
20             segmentIntervall = Alle horizontalen Segmente zwischen
                 maxTopBoundary und bottomBoundary(vertexList.get(i))
21             allSegmentIntervalls.add(segmentIntervall)
22             allVerticalDisplacements.add(teil2(egmentIntervall, maxTopBoundary,
                 bottomBoundary(vertexList.get(i)))
23             break;
24     allVertexIntervalls.add(vertexIntervall)
25 teil3()
```

4.2 Teil 2: Horizontale Segmente verschieben

In diesem Teil werden die horizontalen Kantensegmente, die innerhalb der vertikalen Lücken liegen, soweit wie möglich an das untere Ende des zugehörigen Lückenintervalls bewegt (siehe Abbildung 4.3), um die Lücken in Abschnitt 4.3 kürzen zu können.

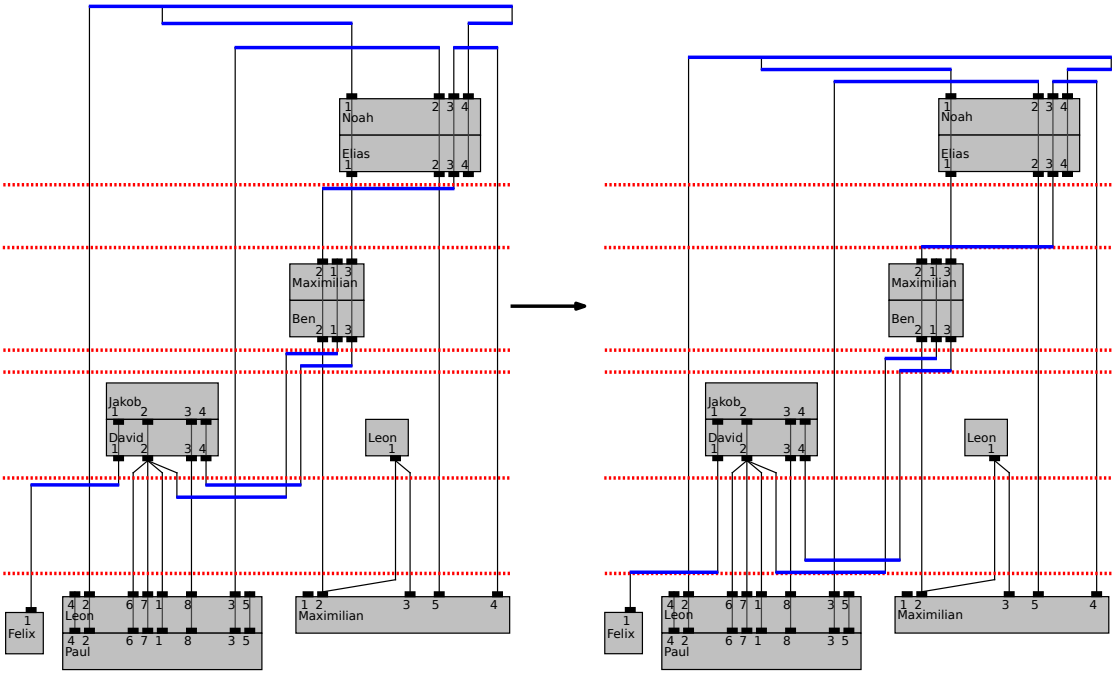


Abb. 4.3: 2. Teil des Stauchalgorithmus: Die horizontalen Segmente (blau) innerhalb der Lücken werden soweit wie möglich nach unten verschoben

Für die Umsetzung wird für jedes Segment ein Objekt einer neuen Klasse angelegt. Diese beinhaltet neben dem eigentlichen Segment zwei weitere Attribute. Das erste Attribut ist ein ganzzahliger Wert, der letztendlich die finale vertikale Position des Segments bestimmt und zu Beginn mit -1 initialisiert wird. Das zweite Attribut ist eine Liste von anderen Objekten dieser Klasse, die die Segmente enthalten, welche unter dem betrachteten Segment liegen und horizontal mit dem betrachteten Segment überlappen. Um nun die vertikale Position der Segmente zu bestimmen, werden die neu erstellten Objekte entsprechend ihrem zugeordneten Segment von unten nach oben durchgegangen. Falls die Liste des Objektes nicht leer ist, wird ihr ganzzahliger Wert auf das Maximum der in den Objekten der Liste enthaltenen ganzzahligen Werte gesetzt und anschließend in jedem Fall um Eins erhöht. Die Werte werden zuletzt mit dem gewünschten minimalen Kantenabstand multipliziert. Diese Endergebnisse liefern nun die gewünschten relativen vertikalen Abstände zwischen Segment und dem unteren Ende der Lücke.

Als Veranschaulichung dazu dient Abbildung 4.4. Der ganzzahlige Wert der Segmente s_i wird hier jeweils auf der rechten Seite des Segments angegeben. Die Pfeile auf dem rechten, oberen Bildausschnitt zeigen an, in welche Listen das Segmentobjekt geschrieben wird.

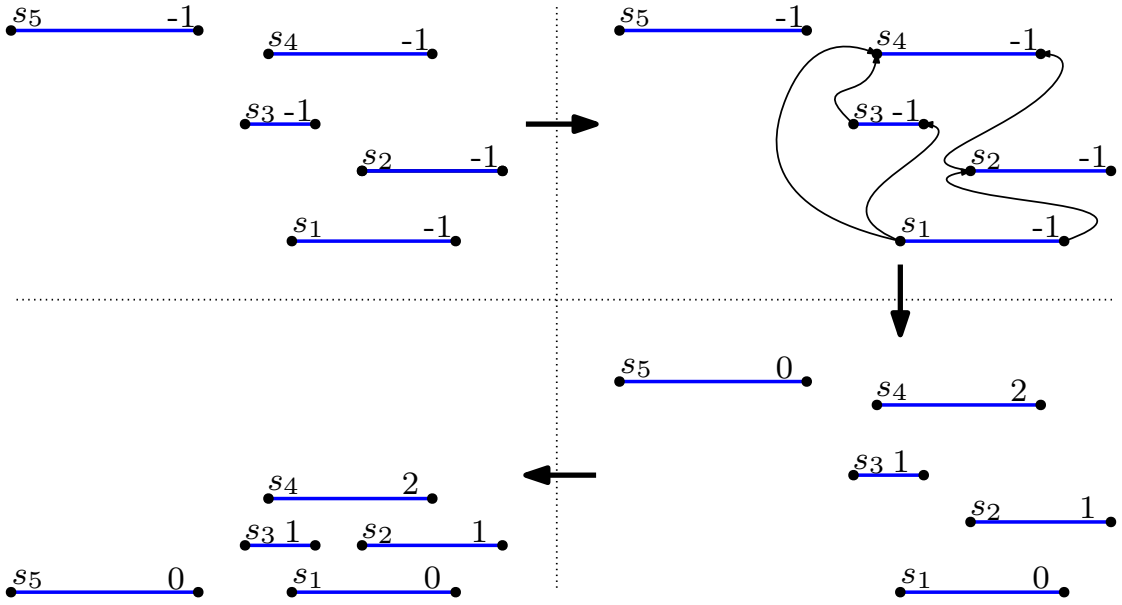


Abb. 4.4: Vorgehen zum Verschieben der Kanten

Für den folgenden Algorithmus nennen wir die neue Klasse *SegmentObject* mit den Attributen *segment*, *level* und *segmentList*. An dieser Stelle sollte noch darauf hingewiesen werden, dass der Algorithmus nur die Segmente in den Lücken zwischen Knotenmengen verschiebt. Segmente die unterhalb des untersten oder oberhalb des obersten Knotens (hier existieren einige in Abbildung 4.4) einer Zeichnung liegen, müssen, falls gewünscht, separat verschoben werden.

Algorithmus 4:
int teil2(segmentList *segmentList*, int *bottomBoundary*, int *topBoundary*)

```
1  padding = gewünschter Kantenabstand
2  segmentObjectList = new List()
3  foreach segment ∈ segmentList do
4  |   segmentObjectList.add(new SegmentObject(segment, −1,new List()))
5  |   Sortiere segmentObjectList entsprechend ihrer Segmente von unten nach oben
6  int index1 = segmentObjectList.size()−1
7  while index1 ≥ 0 do
8  |   int index2 = index1 − 1
9  |   while index2 ≥ 0 do
10 |   |   if segmentObjectList.get(index1).getSegment() überlappt mit
   |   |    segmentObjectList.get(index2).getSegment() then
11 |   |   |   segmentObjectList.get(index1).getSegmentList().add(
12 |   |   |     segmentObjectList.get(index2))
13 |   |   index2 − −
14 |   index1 − −
15 maxIntervallLevel = 0
16 foreach segmentObject1 ∈ segmentObjectList do
17 |   maxLevel = −1
18 |   foreach segmentObject2 ∈ segmentObject.getList() do
19 |   |   if segmentObject2.getLevel()> maxLevel then
20 |   |   |   maxLevel = segmentObject2.getLevel()
21 |   maxLevel + +
22 |   segmentObject2.getLevel()= maxLevel
23 |   Verschiebe segmentObject1.getSegment() an die vertikale Position
   |    bottomBoundary + maxLevel ∗ padding
24 |   if maxLevel > maxIntervallLevel then
25 |   |   maxIntervallLevel = maxLevel
26 return (topBoundary − (bottomBoundary + maxIntervallLevel ∗ padding))
```

4.3 Teil 3: Lücken minimieren

Durch das Verschieben der Kantensegmente (siehe Abschnitt 4.2) wurde erreicht, dass zwischen dem obersten Kantensegment einer Lücke und dem oberen Ende eines Lückenintervalls nur noch vertikale Geraden liegen. Die Länge dieses Bereichs wird *Displacment* genannt. Die vertikalen Geraden lassen sich nun um genau diesen Betrag kürzen, indem alle darüber liegenden Bildinhalte nach unten verschoben werden. Abbildung 4.5 zeigt beispielhaft das Ergebnis dieses Teiles und damit das Endergebnis des vertikalen Stauchens.

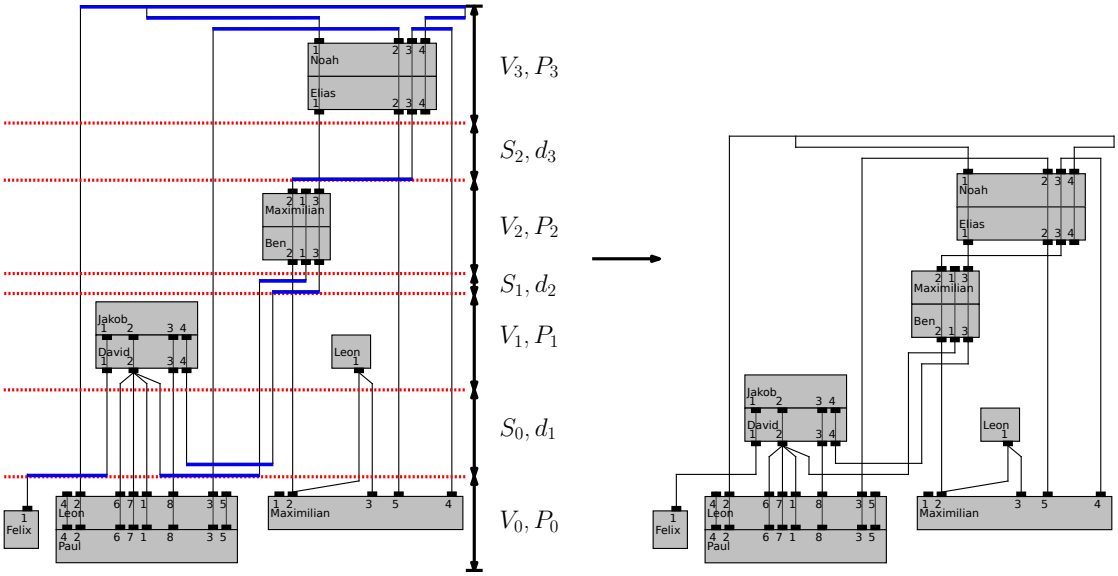


Abb. 4.5: 3. Teil des Stauchalgorithmus: Die vertikalen Lücken werden gekürzt

Außerdem werden in Abbildung 4.5 die Gruppierungen der Bildinhalte und der Displacements, also den zu kürzenden Beträgen, gezeigt, die bereits aus den vorherigen Teilen des Stauchens stammen. So bleiben die Knoten der Menge V_0, die Kantenpunkte der Menge P_0 und die Segmente der Menge S_0 an Ort und Stelle. V_1, P_1 und S_1 werden um den Betrag d_1 nach unten verschoben. V_2, P_2, S_2 werden um den Betrag $d_1 + d_2$ und V_3, P_3, S_3 schließlich um den Betrag $d_1 + d_2 + d_3$ nach unten verschoben. Der Algorithmus dazu sieht folgendermaßen aus:

Algorithmus 5:
teil3(List *allVerticalDisplacements*, List *allVertexIntervalls*, List *allEdgePointIntervalls*, List *AllSegmentIntervalls*)

```
1  cumulativeDisplacement = 0
2  for int i = 0; i < allVerticalDisplacements.size(); i++ do
3      cumulativeDisplacement += allVerticalDisplacement.get(i)
4      foreach vertex ∈ allVertexIntervalls.get(i + 1) do
5          foreach port ∈ vertex do
6              Verschiebe port um cumulativeDisplacement nach unten
7          Verschiebe vertex um cumulativeDisplacement nach unten
8      foreach point ∈ allEdgePointIntervalls.get(i + 1) do
9          Verschiebe point um cumulativeDisplacement nach unten
10     foreach segment ∈ allSegmentIntervalls.get(i + 1) do
11         foreach point ∈ segment do
12             Verschiebe point um cumulativeDisplacement nach unten
```

5 Auswertung

Grundlegendes Ziel der Generalisierung ist es, die Zeichnung zu vereinfachen und somit die Ästhetik zu verbessern. Dieses Ziel setzt voraus, dass die Ästhetikkriterien durch diese Vereinfachung nicht verschlechtert werden. Betrachtet man die in dieser Arbeit vorgestellte Generalisierung, so stellt man fest, dass sie ausschließlich Bildinhalte entfernt, bzw. in einem Fall (Regel 4 von Abschnitt 3.1) kombiniert. Durch diese Operationen können also keine der in Kapitel 2 genannten Ästhetikeigenschaften verschlechtert werden. Auch durch den Stauchalgorithmus erhöht sich weder die Anzahl der Kantenkreuzungen und Kantenknicke, noch wird die Lesbarkeit durch eine Skalierung der Bildinhalte verschlechtert.

Eine Verbesserung der Übersichtlichkeit wird insbesondere durch Reduzierung der notwendigen Zeichenfläche erreicht. Diese Platzeinsparung wird in Abschnitt 5.1 betrachtet. Um die Grenzen der Anwendbarkeit der vorgestellten Algorithmen beurteilen zu können, wird in Abschnitt 5.2 die Laufzeit untersucht. Grundlage der Auswertungen sind 500 reale industrielle Pläne [pra20b], die jedoch strukturell verfremdet und deren Bezeichnungen pseudonymisiert wurden.

5.1 Platzeinsparung

Da der verwendete Zeichenalgorithmus nicht sehr platzeffizient layoutet, lässt sich die Zeichenfläche auch ohne Generalisierung mit dem Stauchalgorithmus reduzieren (siehe Abbildung 5.1). Im Schnitt beträgt die Platzeinsparung hier ca. 41 %.

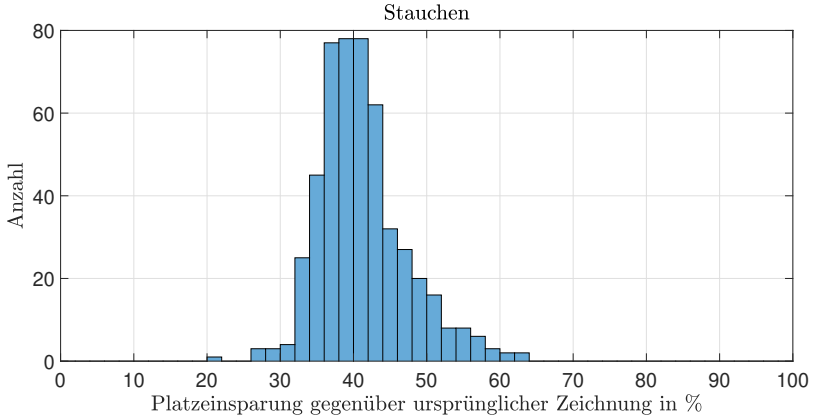


Abb. 5.1: Platzeinsparung von 500 gestauchten Zeichnungen gegenüber den entsprechenden ursprünglichen Zeichnungen in %

Diese Zeichnungen bilden jetzt die Grundlage für die Beurteilung der Kombination aus Generalisierung und Stauchalgorithmus. Schon durch den Generalsierungsprozess allein wird die Zeichenfläche kleiner, wenn Zeicheninhalte am Rand der Zeichnung entfernt werden (siehe Abbildung 5.2). Im Schnitt beträgt die Platzeinsparung gegenüber der gestauchten Zeichnung ca. 21 %. Vereinzelt werden hier sehr viele, in acht Fällen sogar alle, Zeicheninhalte entfernt, wodurch die Platzeinsparung in diesen Fällen entsprechend hoch ausfällt.

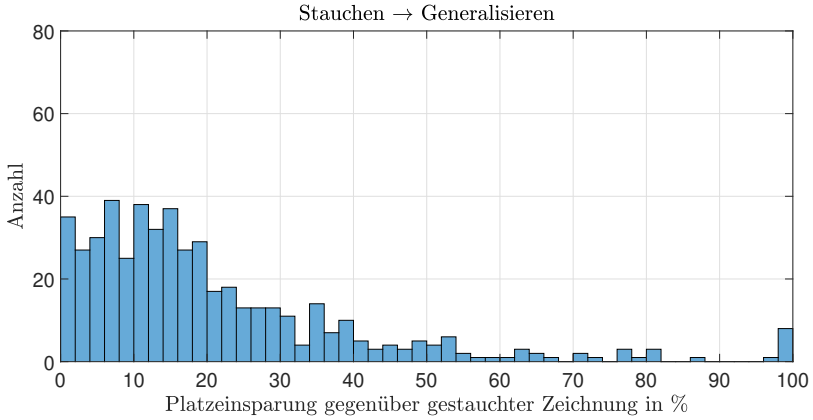


Abb. 5.2: Platzeinsparung von 500 gestauchten und anschließend generalisierten Zeichnungen gegenüber den ausschließlich gestauchten Zeichnungen in %

Zusätzliche 41 %, also insgesamt 62 % gegenüber der gestauchten Zeichnung, werden durch ein erneutes Stauchen eingespart (siehe Abbildung 5.3). Gegenüber der ursprünglichen Zeichnung wird die Zeichenfläche somit insgesamt um ca. 77 % reduziert.

Die Abfolge „Stauchen→Generalisieren→Stauchen“ liefert hierbei die selben Ergebnisse wie „Generalisieren→Stauchen“.

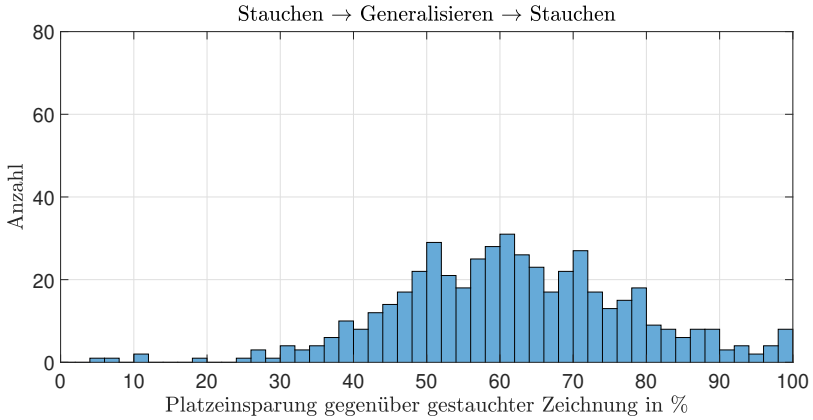


Abb. 5.3: Platzeinsparung von 500 gestauchten, anschließend generalisierten und nochmals gestauchten Zeichnungen gegenüber den ausschließlich gestauchten Zeichnungen in %

Die Platzeinsparung des Stauchens hängt auch davon ab, wie viele der Zeicheninhalte zuvor durch die Generalisierung entfernt werden (siehe Abbildung 5.4). Es gibt hierzu zwei Extremfälle: Werden durch die Generalisierung keine Inhalte entfernt, so hat ein erneutes Stauchen keine Auswirkung. Werden alle Bildinhalte entfernt, so beträgt die Platzersparnis in beiden betrachteten Abfolgen 100 %. Die Vermutung liegt nahe, dass die beiden Abfolgen „Stauchen→Generalisieren→Stauchen“ und „Stauchen→Generalisieren“ im Schnitt für einen sehr geringen und einen sehr hohen Anteil an entfernter Bildinhalte eine ähnliche Platzersparnis bezüglich der gestauchten Zeichnung liefern. Ist diese These korrekt, so lohnt sich ein Stauchalgorithmus (im Schnitt) insbesondere dann, wenn weder sehr viele, noch sehr wenige Inhalte entfernt werden. In Abbildung 5.4 lässt sich erkennen, dass sich die beiden Punktewolken für einen hohen Anteil an entfernten Knoten annähern. Leider existieren keine Datensätze, in denen die Generalisierung prozentual sehr wenige Knoten entfernt; ca. 23 % sind hier das Minimum.

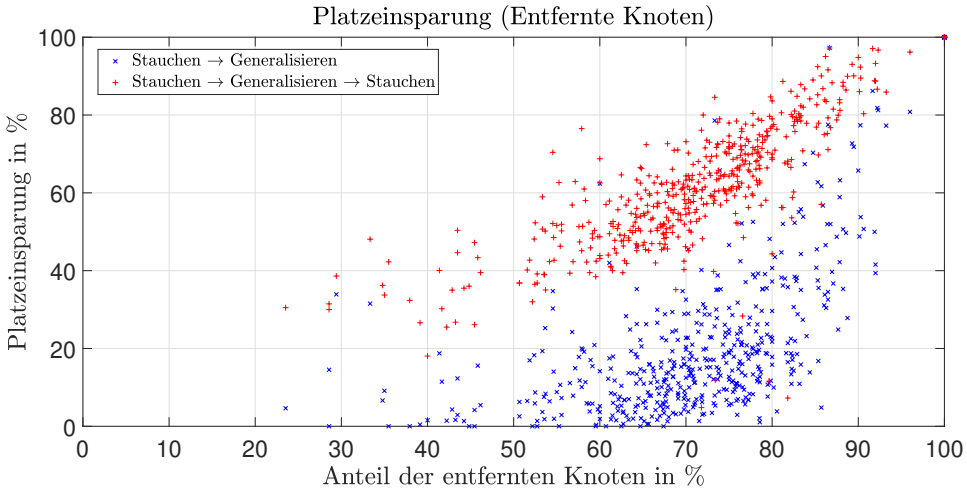


Abb. 5.4: Vergleich der Platzeinsparung bezüglich der gestauchten Zeichnung der Abfolgen „Stauchen→Generalisieren→Stauchen“ und „Stauchen→Generalisieren“ in Abhängigkeit der prozentual entfernten Knoten

5.2 Laufzeit

Die Laufzeit wurde direkt in Java mithilfe der Funktion *System.nanoTime()* bestimmt. Der dafür verwendete Rechner besitzt einen „Intel Core i5-3230M“ Prozessor und 8GB Arbeitsspeicher (1600MHz).

Generell ist der Rechenaufwand von „großen“ Graphen, also Graphen mit vielen Knoten und Kanten, höher und somit auch die Laufzeit. Da die Anzahl der Knoten stark mit der Anzahl an Kanten korreliert (KORR($|V|$,$|E|$)$= 0,9591$ für unsere Datensätze), genügt es, die Laufzeit in Abhängigkeit einer der beiden Größen zu betrachten. Sowohl die Laufzeit der Generalisierung (siehe Abbildung 5.5) als auch die des Stauchens (siehe Abbildung 5.6) streut merklich. Um zufällige Einflüsse zu reduzieren, wurde pro Messpunkt

je fünf mal gemessen und daraus der Median gebildet.

Sowohl für die Generalisierung als auch das Stauchen liegt die Laufzeit für die Testdatensätze mit weniger als 180 Knoten im zweistelligen Millisekundenbereich. Somit eignen sich die Algorithmen auch für Anwendungen, die ein unmittelbares Feedback an den Nutzer erfordern, wie beispielsweise ein Zooming-Feature innerhalb einer graphischen Oberfläche.

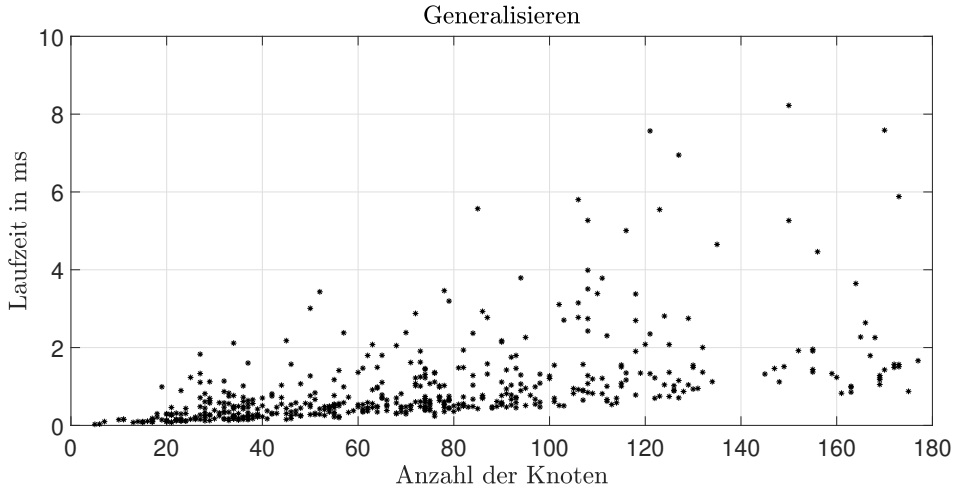


Abb. 5.5: Laufzeit der Generalisierung in Abhängigkeit der Knotenzahl

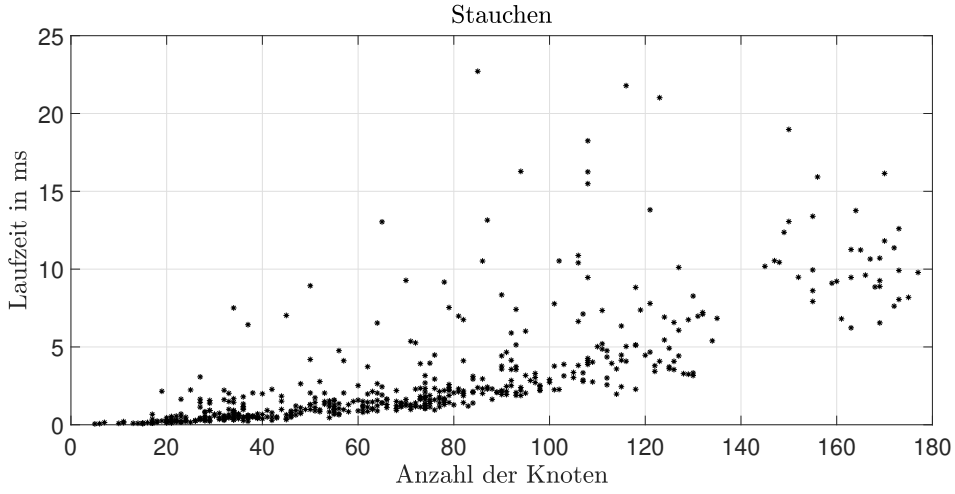


Abb. 5.6: Laufzeit des Stauchalgorithmus in Abhängigkeit der Knotenzahl

6 Fazit

In dieser Arbeit wurden ein Beispiel einer Generalisierung für Kabelpläne, sowie ein dazu passender Stauchalgorithmus vorgestellt und implementiert. Den meisten Aufwand hat hierbei die Berücksichtigung der Hyperkanten verursacht, die in der verwendeten praline-Datenstruktur unterstützt werden.

Sowohl die vorgestellten Generalisierungsoperationen, als auch das Stauchen verschlechtern keines der betrachteten Ästhetikkriterien, behalten die mental map des Betrachters bei und liefern jeweils einen Beitrag zur Reduzierung der erforderlichen Zeichenfläche um durchschnittlich 77 %. Die Laufzeit der beiden Algorithmen beträgt bei den 500 getesteten Datensätzen mit bis zu 177 Knoten jeweils weniger als 30 ms.

Eine Aufgabe, die zukünftige Arbeiten behandeln können, ist eine Verbesserung des Stauchalgorithmus. Idee hierfür ist, die Knoten (falls es Freiraum dafür gibt) soweit zu verschieben, dass die Oberkante des unteren oder die Unterkante des oberen Knotens mit dem Mittelpunkte des jeweils anderen Knotens fluchtet. Somit ließe sich noch immer erkennen, dass der eine Knoten oberhalb des anderen Knotens liegt. Die Zeichnung wäre allerdings kompakter. Abbildung 6.1 veranschaulicht die Idee.

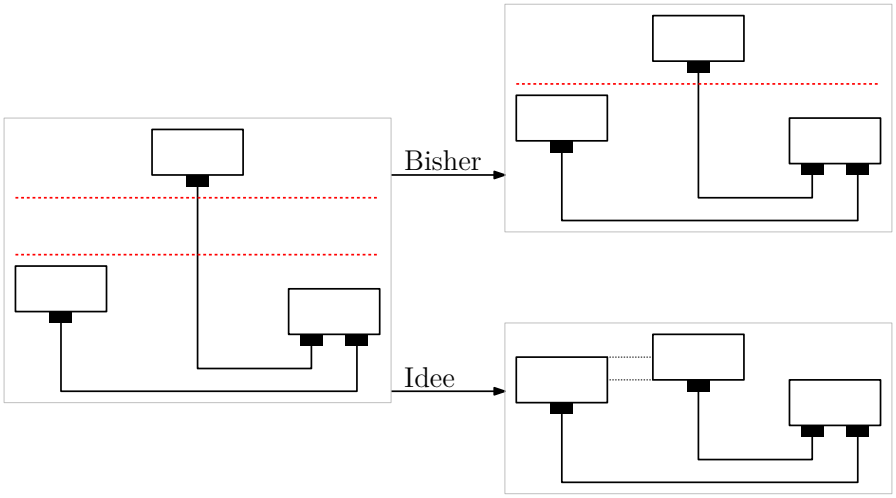


Abb. 6.1: Idee für einen verbesserten Stauchalgorithmus

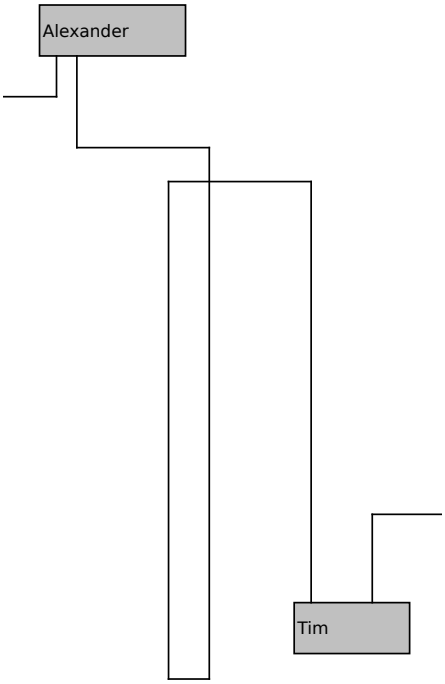


Abb. 6.2: Ausschnitt einer Zeichnung nach der Generalisierung: Die abgebildete Kante hat eine unnötige Kreuzung und mehrere unnötige Knicke

Großes Optimierungspotenzial besteht zudem beim Kantenrouting. Nach einer Generalisierung, insbesondere von Zeichnungen mit vielen Knoten, besitzen Knoten häufig unnötig viele Kantenknicke und -kreuzungen (siehe Abbildung 6.2). Hierzu könnte eine Heuristik zur Verringerung dieser Kantenknicke entwickelt werden, oder alternativ ein Zeichenalgorithmus, der die Kanten neu routet und dabei die Position der Knoten beibehält.

Literaturverzeichnis

[AKY05] James Abello, Stephen G. Kobourov und Roman Yusufov: Visualizing Large Graphs with Compound-Fisheye Views and Treemaps. In: János Pach (Herausgeber): *Graph Drawing*, Band 3383 der Reihe *Lect. Notes Comput. Sci.*, Seiten 431–441. Springer Berlin Heidelberg, 2005, 10.1007/978-3-540-31843-9$_4$4.

[DMS06] Tim Dwyer, Kim Marriott und Peter J. Stuckey: Fast Node Overlap Removal. In: Patrick Healy und Nikola S. Nikolov (Herausgeber): *Graph Drawing*, Band 3843 der Reihe *Lect. Notes Comput. Sci.*, Seiten 153–164. Springer Berlin Heidelberg, 2006, 10.1007/1161805$8_1$5.

[FK96] Arno Formella und Jörg Keller: Generalized Fisheye Views of Graphs. In: Franz J. Brandenburg (Herausgeber): *Graph Drawing*, Band 1027 der Reihe *Lect. Notes Comput. Sci.*, Seiten 242–253. Springer Berlin Heidelberg, 1996, 10.1007/BFb0021808.

[GKN04] E. Gansner, Y. Koren und S. North: Topological Fisheye Views for Visualizing Large Graphs. In: Matt Ward und Tamara Munzner (Herausgeber): *IEEE Symposium on Information Visualization*, Seiten 175–182, 2004, 10.1109/INFVIS.2004.66.

[MB93] William Mackaness und Kate Beard: Use of Graph Theory to Support Map Generalization. *Cartography and Geographic Information Science*, 20(4):210–221, 1993, 10.1559/152304093782637479.

[NNB+17] Lev Nachmanson, Arlind Nocaj, Sergey Bereg, Leishi Zhang und Alexander Holroyd: Node Overlap Removal by Growing a Tree. *Journal of Graph Algorithms and Applications*, 21(5):857–872, 2017, 10.7155/jgaa.00442.

[pra20a] PRALINE Data Structure and Layouting Algorithm, 2020. `https://github.com/j-zink-wuerzburg/praline`.

[pra20b] PRALINE Pseudo Plans Algorithm and Data Sets, 2020. `https://github.com/j-zink-wuerzburg/pseudo-praline-plan-generation`.

[Saa95] Alan Saalfeld: Map Generalization as a Graph Drawing Problem. In: Roberto Tamassia und Ioannis G. Tollis (Herausgeber): *Graph Drawing*, Band 894 der Reihe *Lect. Notes Comput. Sci.*, Seiten 444–451. Springer Berlin Heidelberg, 1995, 10.1007/3-540-58950-3$_3$98.

[STT81] Kozo Sugiyama, Shojiro Tagawa und Mitsuhiko Toda: Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981, 10.1109/TSMC.1981.4308636.

[WZBW20] Julian Walter, Johannes Zink, Joachim Baumeister und Alexander Wolff: Layered Drawing of Undirected Graphs with Generalized Port Constraints. In: David Auber und Pavel Valtr (Herausgeber): *Graph Drawing and Network Visualization (GD'20)*, Band 12590 der Reihe *Lect. Notes Comput. Sci.*, Seiten 220–234. Springer International Publishing, 2020, 10.1007/978-3-030-68766-$3_1$8.