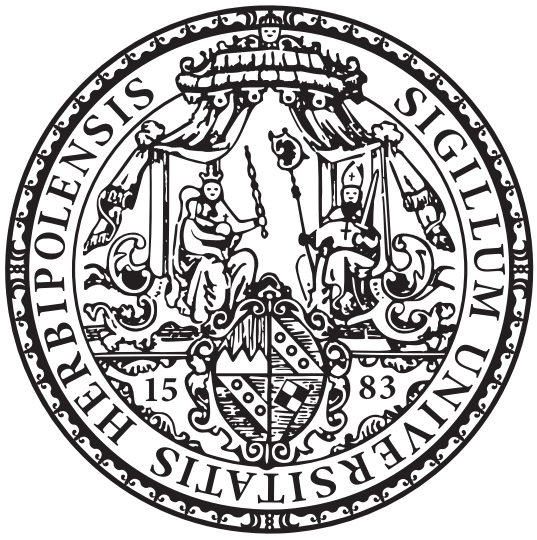
Bachelorarbeit

Stauchung orthogonaler Graphenzeichnungen mithilfe komplexer Schnitte

Sebastian Körner

Abgabedatum: 25. 07. 2023
Überarbeitet: 26. 11. 2023
Betreuer: Prof. Dr. Alexander Wolff
Tim Hegemann, M. Sc.



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

Zusammenfassung

In vielen technischen Anwendungen, wie zum Beispiel für Kabelpläne, Netzwerkstrukturen oder Kontrollflussgraphen werden große Mengen an orthogonalen Graphenzeichnungen benötigt, um komplexe Strukturen und Zusammenhänge für einen Ingenieur übersichtlich darzustellen. Die Zeichnung solcher Graphen geschieht oft automatisiert, mithilfe von Algorithmen. Diese lassen jedoch in vielen Fällen ungenutzte Freiräume, was zu einer sehr großen Zeichnung führt, die sehr unhandlich ist, da sie entweder sehr klein skaliert werden muss oder nur abschnittweise angezeigt werden kann. Beides wirkt sich negativ auf die Lesbarkeit und Interpretierbarkeit aus. In solchen Fällen können Komprimierungsalgorithmen helfen, Graphenzeichnungen übersichtlicher zu machen und das Arbeiten mit ihnen zu vereinfachen. Der Algorithmus, den wir in dieser Arbeit vorstellen, verwendet dafür Schnitte. Im einfachsten Sinne sind Schnitte orthogonale Polylinien, die durch eine Graphenzeichnung verlaufen und diese in zwei Abschnitte unterteilen. Die Schnitte werden dabei so gelegt, dass der Bereich unterhalb eines Schnittes um die sogenannte Größe des Schnittes nach oben verschoben werden kann, um somit die Zeichnung zu Stauchen. Der Vorteil dieses Verfahrens ist, dass es einfach nachzuvollziehen ist und die Struktur des Graphen nur geringfügig verändert wird.

Abstract

The aim of this research is to compact orthogonal graph drawings in a comprehensible way to increase their readability. These specialized kinds of graph drawings are prominent in many technical use cases such as cable plans, network structures and control flow graphs. As these drawings are usually needed in larger amounts and rather spontaneously, their layouting is often handled algorithmically. One problem that is common among layouting algorithms is their suboptimal use of space. The resulting graph drawings can be so large that they become hard to read or even too big to display on a screen. A compacting algorithm can be deployed as a post processing step to tackle these issues. We introduce a novel algorithm that uses cuts to eliminate unused space in a layout. A cut is an orthogonal polyline that divides a graph drawing in an upper and a lower part. The latter, which contains all elements that are located on or below the cut line, is moved upwards by the size of the cut, thereby pulling elements of opposing sides closer together and compacting the graph drawing. Due to the limited definition of cuts the resulting layouts retain a very similar structure especially regarding the edge routing.

Inhaltsverzeichnis

1. Einleitung

Bei der Entwicklung von moderner Soft- und Hardware kommen immer häufiger eine Vielzahl an Graphen zum Einsatz. Softwareseitig werden beispielsweise UML-Diagramme sowie Daten- und Kontrollflussgraphen eingesetzt, um eine Programmstruktur anschaulich darzustellen. Beispiele von der Hardware-Seite sind Pläne für elektrische oder hydraulische Systeme oder Netzwerkstrukturen. Historisch wurden solche Zeichnungen meist händisch angefertigt. Mit stetig steigender Komplexität dieser Systeme steigt jedoch auch die Nachfrage nach automatisierten Lösungen, um solche Graphen zu erstellen und zu zeichnen. Auch die Anzahl der benötigten Zeichnungen ist zum Beispiel durch die steigende Popularität von iterativen Entwicklungsprozessen und die damit verbundene Notwendigkeit von ad-hoc verfügbaren, übersichtlichen Darstellungen der sich stetig verändernden Projektstruktur gestiegen. Die algorithmisch gezeichneten Graphen sollen jedoch den von Hand erstellten in keinem Aspekt nachstehen.

Das iPraline Framework von Zink et al. [ZWBW20a] versucht diese Anforderungen zu erfüllen. Es wurde primär für die Anwendung auf industriellen Kabelplänen entwickelt. Der Zeichenalgorithmus übernimmt viele Paradigmen von händisch gezeichneten Graphen. So werden zum Beispiel Knoten immer auf festen Ebenen platziert und Kanten nur an der Ober- und Unterseite der Knoten angebracht. Ein Ingenieur ist mit dieser Art der Darstellung bereits vertraut und kann sich somit schneller in der Zeichnung orientieren. iPraline minimiert zudem die Anzahl an Knicken und Kreuzungen in der Zeichnung, da diese die Lesbarkeit des Layouts erheblich verschlechtern. Aus demselben Grund gibt es auch einen festen Mindestabstand zwischen allen Elementen der Zeichnung. All diese Restriktionen führen jedoch dazu, dass die resultierenden Zeichnungen oftmals sehr viel Platz einnehmen und viele ungenutzte Freiräume lassen. Das kann wiederum zu einem Problem werden, wenn die Zeichnung nicht mehr auf einen Bildschirm passt. Muss die Zeichnung infolgedessen skaliert werden, so kann es sein, dass komplexere Kantenstrukturen nicht mehr erkennbar sind oder Knotenbeschriftungen unlesbar werden.

In dieser Arbeit stellen wir einen neuen Stauchungsalgorithmus vor, der das iPraline Framework erweitert. Wir behandeln dabei speziell die eben beschriebene Art von orthogonalen Zeichnungen. Es sollen bei der Stauchung weitestgehend alle Layout-Eigenschaften der ursprünglichen Zeichnung beibehalten werden, um weiterhin von der Intuitivität der iPraline Zeichnung zu profitieren. Diese Vorgabe erfüllen wir mithilfe sogenannter *Schnitte*. Ein Schnitt ist eine ebenfalls orthogonale Polylinie, die eine Zeichnung in einen oberen und einen unteren Abschnitt teilt. Einen Schnitt bezeichnen wir als gültig, wenn wir alle Elemente unterhalb des Schnittes um einen festen Betrag nach oben verschieben können, sodass sich Kanten oder Knoten dadurch nicht überschneiden und der Mindestabstand weiterhin eingehalten wird. Durch Anwendung eines gültigen Schnitts können wir somit die Abstände zwischen Elementen in der Zeichnung verringern und somit das Layout

stauchen ohne die Zeichnung dabei zu beschädigen.

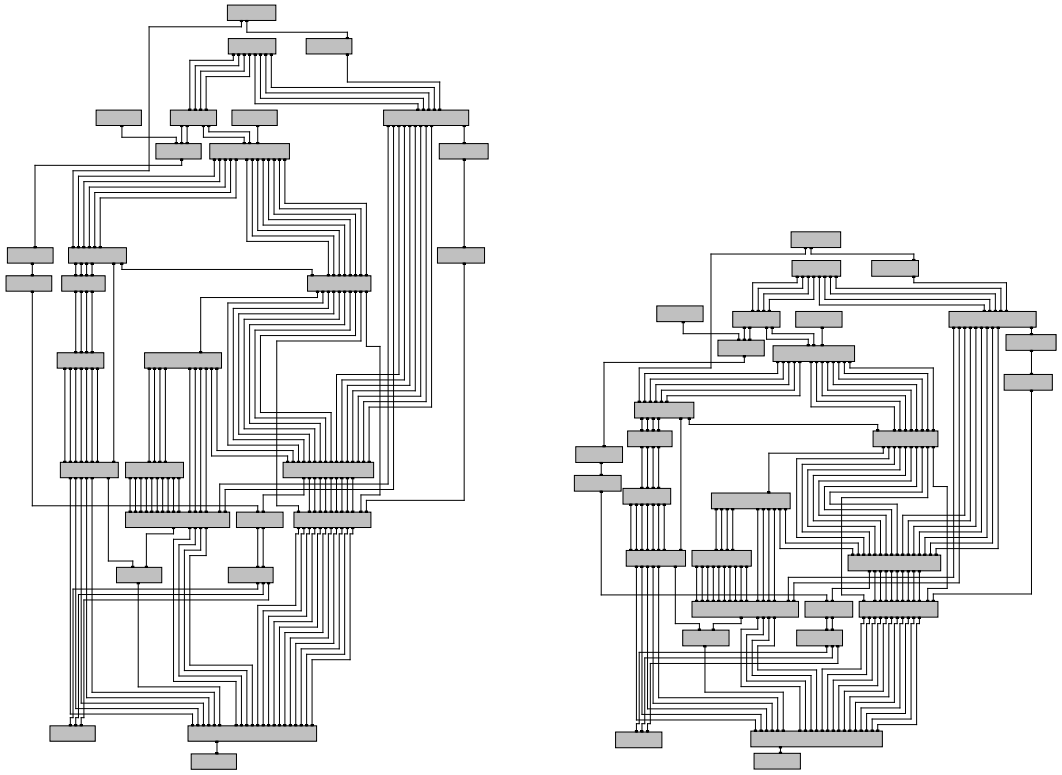
Das Erstellen orthogonaler Zeichnungen ist ein viel erforschter Teilbereich der Graphentheorie. Battista et al. [BETT94] stellen beispielsweise einige solche Algorithmen vor. Seit dessen Einführung im Jahr 2000 durch Biedl et al. [BMT97] hat sich eine aus drei Phasen bestehende Struktur für solche Zeichenalgorithmen durchgesetzt. Bei diesem Verfahren wird im dritten Schritt oftmals ein Stauchungsalgorithmus eingesetzt um die Zeichenfläche zu minimieren. Biedel et al. verwenden im speziellen einen Algorithmus von Lengauer [Len90], dessen Arbeit ein Grundstein im Entwurf für integrierte Schaltkreise ist. Dieser stellt eine Vielzahl von Stauchungsalgorithmen für die Verwendung in Kabelplänen vor. Er beschreibt dabei auch eindimensionale, graphenbasierte Algorithmen, die, ähnlich wie der von uns vorgestellte Algorithmus, die zwei Stauchungsrichtungen separat betrachten und das Problem der Stauchung auf ein Graphenproblem reduzieren. Unter dem Stichwort „Topological Compaction“ stellt Lengauer einen zweidimensionalen Stauchungsalgorithmus vor, der, wenn auch auf eine völlig andere Weise, ebenso wie unser Algorithmus Schnitte verwendet. Diese compacting Algorithmen betrachten jedoch nur strukturell Blöcke in einem Graphen und explizit keine Kanten. In diesem wichtigen Aspekt unterscheiden sich alle Stauchungsalgorithmen von Lengauer zu unserem. Ein anderer Stauchungsalgorithmus für orthogonale Zeichnungen, der, gleich zu unserem, alle Elemente im Graphen berücksichtigt und Schnitte zur inkrementellen Stauchung verwendet ist uns nicht bekannt.

Herkömmliche Stauchungsalgorithmen sind nicht in der Lage die Vielzahl an Constraints, die durch iPraline Layouts vorgegeben werden aufrecht zu erhalten. Unser Algorithmus hingegen schafft es die Wichtigsten dieser Paradigmen einzuhalten (Siehe Abbildung 1.1) und kann somit für unseren Anwendungsfall voraussichtlich bessere Resultate erzielen.

Der in dieser Arbeit vorgestellte Algorithmus ist zudem in mehrere Module unterteilt. Durch unabhängige Veränderungen in diesen Modulen kann er hinsichtlich verschiedener Heuristiken optimiert werden und so speziell für den jeweiligen Anwendungsfall angepasst werden. Ein Beispiel für eine solche Anpassung wird im Abschnitt 4.2.2 gegeben. Im Ausblick (Abschnitt 6) werden außerdem weitere genannt, um eine Weiterentwicklung des Algorithmus anzuregen.

Eigener Beitrag. Das Ziel dieser Arbeit ist es zu klären, ob die von iPraline erstellten Zeichnungen gestaucht werden können, ohne dabei die Struktur maßgeblich zu verändern. Das Ziel dabei ist es, die Lesbarkeit des Layouts zu verbessern und die Arbeit mit solchen Zeichnungen in einem industriellen Kontext zu vereinfachen. Dazu formulieren wir zunächst eine präzise Definition für Schnitte und stellen anschließend einen Algorithmus vor, der diese in einer Zeichnung im iPraline Datentyp findet. Der Algorithmus kann dabei in drei große Abschnitte unterteilt werden. Im ersten Schritt wird das Layout zunächst in elementare, rechteckige Bereiche unterteilt. Wir haben das Problem der Schnittsuche damit in ein kleines Sub-Problem zerteilt: Die Suche von Schnitten zwischen zwei Rechtecken. Der Zweite Abschnitt befasst sich damit, dieses Unterproblem zu lösen. Im Letzte Abschnitt setzten wir schließlich die kleinen Schnitte zu einem

Vollständigen, möglichst großen Schnitt durch den gesamten Graphen zusammen. Wir übersetzen die bereits gefundenen Teilergebnisse dafür in einen gerichteten st-Graphen und nutzen einen angepassten Dijkstra-Algorithmus, um einen optimalen Weg zu finden. Wir beweisen, dass auf diese Weise jeder Schnitt in einer Zeichnung gefunden werden kann. Im Anschluss definieren wir einen Stauchungsalgorithmus, der diese Schnitte iterativ findet und sie nutzt, um die Zeichnung zu stauchen. Wir werten diesen Algorithmus anhand einer Vielzahl von Metriken aus und beantworten die Frage, ob die Struktur der Ausgangszeichnung weiterhin erhalten bleibt.



(a) Ausgangszeichnung

(b) Gestauchte Zeichnung

Abb. 1.1.: Ein Graph mit 29 Knoten und 131 Kanten, der mithilfe unseres Stauchungsalgorithmus um 33% verkleinert werden kann.

2. Grundlagen

2.1. Grundbegriffe

Um die nachfolgenden Abschnitte verstehen zu können, muss zunächst das Grundvokabular von Graphen etabliert werden. Wir betrachten einen ungerichteten *Graph* G, der als 2-Tupel (V, E) definiert ist. Dabei sind V die *Knoten* in einem Graphen. Knoten bilden die grundlegenden Elemente eines Graphen. Die *Kanten* des Graphen werden als E geschrieben. Ein Kanten verbinden genau zwei Knoten miteinander. Sogenannte *Hyperkanten*, die mehr als zwei Knoten gleichzeitig verbinden werden in dieser Arbeit nicht berücksichtigt. Das iPraline Framework betrachtet außerdem nur *ungerichtete Graphen*. Das bedeutet, dass alle Kanten im Graphen ungerichtet sind, also immer in beide Richtungen zwischen den Knoten verlaufen. Wir können eine Kante somit als ungeordnetes 2-Tupel (v, w) mit $v, w \in V$ schreiben.

Die *Zeichnung* $\Gamma(G)$ (synonym Layout) eines Graphen ist dessen geometrische Darstellung, also die Zuordnung von Formen und Positionen zu allen Graphenelementen. Knoten werden im Folgenden immer als achsenparalleles Rechteck gezeichnet, das einen Text enthalten kann. Dieses Rechteck ist definiert durch eine Position $pos(v) = (x, y) \in \mathbb{N}^2$, die die Lage der oberen linken Ecke angibt, sowie eine Breite $b(v)$ und Höhe $h(v)$. Kanten werden als orthogonale Polylinien gezeichnet. Eine orthogonale Polylinie ist eine Linie, die aus mehreren geradlinigen Teilkanten besteht, die jeweils zur x- oder y-Achse parallel verlaufen. Die einzelnen Linien, aus der sich die Polylinie zusammensetzt, bezeichnen wir als *Kantenabschnitte*. Die Kantenabschnitte einer Kante verlaufen immer abwechselnd vertikal und horizontal. Eine Besonderheit des iPraline Frameworks, die auf die Anwendung für Kabelpläne zurückzuführen ist, ist die Nutzung sogenannter *Ports*. Kanten werden immer über Ports mit Knoten verbunden. Ein Port $p = (v, e)$ ist genau einer Kante und einem Knoten zuzuordnen. Die Ports werden an dem Verbindungspunkten zwischen Kanten und Knoten gezeichnet und werden als kleinere, schwarze Rechtecke dargestellt. Einen Algorithmus, der aus einem Graph eine Graphzeichung erstellt nennen wir *Zeichenalgorithmus* oder *Layoutingalgorithmus*.

2.2. iPraline Layouts

In diesem Abschnitt wollen wir etwas genauer beschreiben, welche Eigenschaften die Layouts aus dem iPraline Framework haben, um später darauf zu achten, diese nicht zu verändern. Alle hier genannten Eigenschaften sind noch einmal in Abbildung 2.1 verdeutlicht.

Wir haben bereits im vorherigen Abschnitt etabliert, dass die Zeichnung orthogonal

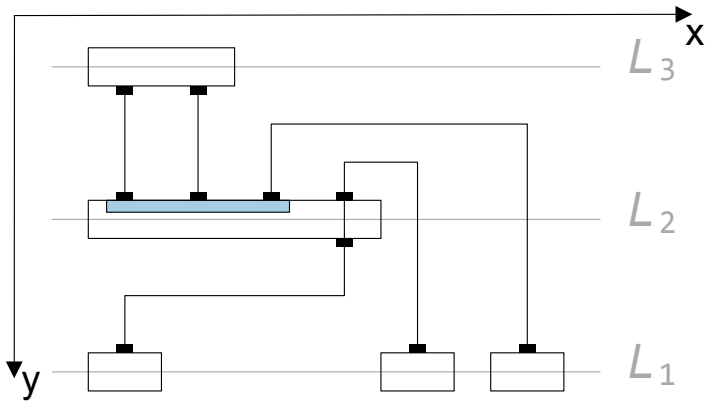


Abb. 2.1.: Eine einfache Zeichnung, die alle Besonderheiten von iPraline aufzeigt. Die Knoten sind in diesem Fall auf drei Ebenen L_1 bis L_3 verteilt. Eine Portgruppe ist blau hinterlegt. Ein Portpaar ist ebenfalls im Knoten auf Ebene zwei zu sehen. Diese Darstellung ist abgeleitet aus [ZWBW22].

ist und Knoten als Rechtecke dargestellt werden. Die Größe der Rechtecke ist fest und darf explizit nicht verkleinert werden, da die Beschriftung sonst möglicherweise nicht mehr hineinpasst. Weiterhin werden Kanten und die dazugehörigen Ports nur an der Ober- und Unterseite der Knoten angebracht. Die Ports werden in *Portgruppen* angeordnet, die die Platzierung der Ports vorgibt. Sie sind so angeordnet, dass der Betrachter verwandte Kanten schnell erkennen kann. Die Position der Ports und der Kantenenden ist somit essenziell für die Struktur der Zeichnung und darf ebenfalls nicht verändert werden. iPraline unterstützt zudem sogenannte *Port Pairings*. Die zwei Ports eines Port Pairings werden immer auf den gegenüberliegenden Seiten desselben Knotens und auf derselben y-Achse gezeichnet. Solche Port Pairings behalten wir ebenfalls bei, da wir, wie bereits festgelegt, die Position der Ports nicht verändern. Eine weitere Besonderheit des Layoutingalgorithmus ist, dass Knoten nur auf einer geringen Anzahl an Ebenen gezeichnet werden. Dies dient der Orientierung in der Zeichnung und ist, wie die meisten Charakteristiken dieses Algorithmus, von handgezeichneten Kabelplänen inspiriert. Diese Beschränkung werden wir für unseren Stauchungsalgorithmus aufheben. Würden wir dies nicht tun wäre der erzielte Stauchungseffekt minimal, da der iPraline Algorithmus solche Layouts bereits platzeffizient erstellen kann. Wir evaluieren im Auswertungs-Abschnitt (Abschnitt 5), wie sehr die Ebenenstruktur tatsächlich durch die Stauchung beeinträchtigt wird und stellen im Ausblick (Abschnitt 6) eine mögliche Kompromisslösung vor.

3. Schnitte

Nun, da wir geklärt haben, welche Gegebenheiten und Anforderungen herrschen, wollen wir genauer erläutern, was gesucht wird. Wir beginnen mit der Definition eines Schnittes. In der Graphentheorie versteht man unter einem Schnitt meist eine Unterteilung der Knotenmenge eines Graphen in zwei Teilmengen X und $V\backslash X$ mit $X \subset V$. Daraus ergibt sich eine Menge an *Schnittkanten* $A(X, V\backslash X) = \{(u, v) \in E \mid u \in X, v \in V\backslash X\}$, die zwischen den Beiden Teilmengen verlaufen [HS93]. In Graphenzeichnungen werden Schnitte meist mithilfe eine Trennlinie durch alle Schnittkanten dargestellt.

Angelehnt an diese Definition wollen wir Schnitte nun, statt für die Knotenmenge, für alle Elemente in einer Zeichnung definieren. Wir betrachten die Zeichnung dafür als Menge von Punkten $p = (x, y)$, mit $x, y \in \mathbb{N}$. Rechtecke bestehen aus vier Eckpunkten und Kanten wiederum aus einem Startpunkt, einer Menge an Knickpunkten und einem Endpunkt. Eine Zeichnung besteht somit vereinfacht nur aus einer Menge an Punkten P. Unser Schnitt teilt diese Menge an Punkten in zwei Teilmengen $X \subset P$ und $P\backslash X$, ähnlich der obigen Definition. Unsere Schnittlinie ist dabei nichts anderes als eine orthogonale Polylinie, die über die gesamte Breite des Graphen verläuft. Der Startpunkt $s(P) = (\min(\{p_x \mid p \in P\}) - 1, \max(\{p_y \mid p \in P\}) + 1)$ liegt dabei links unterhalb der Zeichnung und der Endpunkt $t(P) = (\max(\{p_x \mid p \in P\}) + 1, \max(\{p_y \mid p \in P\}) + 1)$ rechts unterhalb. Die Linie muss außerdem y-Monoton verlaufen. Das bedeutet, dass $p_{a_x} <= p_{b_x}$, wenn der Index a kleiner ist als der Index b in der Reihe aller Punkte der Polylinie $\{s, p_2, p_3, \ldots, p_{n-1}, t\}$. Eine weitere Einschränkung ist, dass Schnittlinien keine Knoten oder horizontalen Kantenabschnitte schneiden dürfen. Von einer solchen Schnittlinie auf einen Schnitt zu kommen in nun trivial. Aus der obigen Definition ergibt sich, dass jeder Punkt in der Zeichnung entweder über, auf oder unter der Schnittlinie liegt. Wir legen fest, dass alle Punkte, die auf oder unterhalb der Linie liegen zur Schnittmenge X gehören. Alle Punkte darüber gehören Folglich zu $P\backslash X$. Wie vielleicht auffällt kann X nach dieser Definition (und im Gegensatz zur Definition auf Graphen) keine beliebige Teilmenge von P sein.

Wir wollen nun die Punkte in der Zeichnung verschieben, um den Graphen schlussendlich zu verkleinern. Wir verlagern dafür alle Punkte in der Teilmenge X um einen einheitlichen Wert nach oben. Die Anzahl der Einheiten im Koordinatensystem, um die wir die Punkte verschieben, bezeichnen wir als Größe des Schnittes.

Wir unterscheiden nun weiter zwischen einem gültigen und einem ungültigen Schnitt. Die Gültigkeit beschreibt, ob die Verschiebung durch den Schnitt in einem korrekt gezeichneten Graphen resultiert. Hierfür betrachten wir nun nicht mehr nur die einzelnen Punkte der Zeichnung, sondern auch die übergeordneten Elemente. Eine Fehlerquelle haben wir bereits in der Definition für Schnitte eliminiert. Dass Knoten und Kantenabschnitte nach Durchführung eines Schnittes nicht mehr orthogonal sind haben wir

bereits durch die Bedingung, dass Knoten und horizontale Kantenabschnitte nicht geschnitten werden dürfen, sichergestellt. Dadurch können alle Punkte, der entsprechenden Gruppe immer nur zusammen bewegt werden, wodurch die horizontale Eigenschaft erhalten bleibt. Durch dieselbe Bedingung wissen wir auch, dass die Ausmaße von Knoten-Rechtecken gleichbleiben müssen. Ein Problem bleibt jedoch noch: Durch Verschiebung kann es passieren, dass der Mindestabstand zwischen Knoten oder Kantenabschnitten nicht mehr eingehalten wird oder es sogar zu Überschneidungen kommt. Genau darin besteht die Schwierigkeit einen gültigen Schnitt zu finden.

Anmerkung: Die eben beschriebenen Schnitte fallen in die Kategorie der horizontalen Schnitte, da sie über die Breite des Graphen verlaufen. Mithilfe solcher Schnitte kann eine Zeichnung nur in ihrer Höhe gestaucht werden. Wir umgehen dieses Problem in unserer Implementierung, indem wir die Zeichnung nach der horizontalen Stauchung um 90 Grad drehen, die Zeichnung erneut horizontal stauchen und sie schließlich wieder zurückdrehen. In der nachfolgenden Erklärung zum Algorithmus werden wir ebenfalls nur diese Art von Schnitten betrachten.

4. Algorithmus

Wir beschreiben nun einen Algorithmus zur Suche von Schnitten in iPraline Graphen. Wir gehen dabei nach dem divide and conquer Prinzip vor. Der Algorithmus kann somit in drei Abschnitte gegliedert werden. Die verschiedenen Schritte sind auch in Abbildung 4.1 an einem Beispiel aufgezeigt.

1. Im ersten Schritt zerteilen wir die Graphenzeichnung in möglichst große, leere, rechteckige Flächen. Wir wissen für diese Flächen, dass eine Schnittlinie sich darin frei bewegen kann, ohne eine Kante oder einen Knoten zu schneiden. Wir speichern diese Rechtecke als Knoten in einem Hilfsgraphen ab.

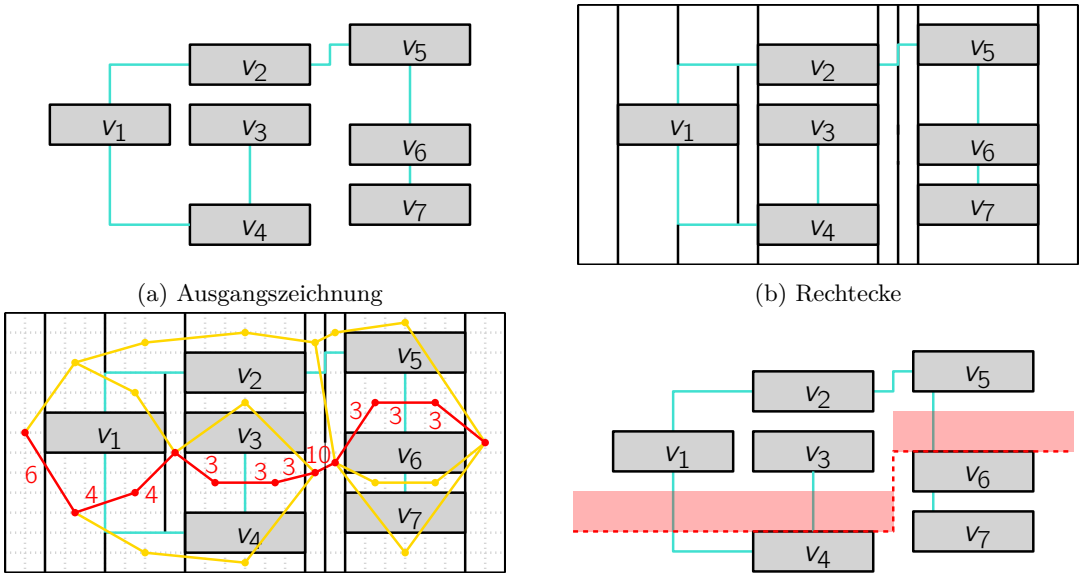
2. Anschließend suchen wir möglichst große gültige Schnitte zwischen Rechteckspaaren. Diese bilden die Kanten unseres Hilfsgraphen. Ein Schnitt, der von einem Rechteck ins Nächste verläuft, darf zum Beispiel aufgrund der Position von Kantenabschnitten zwischen den beiden Rechtecken oftmals eine gewisse Größe nicht überschreiten. Diese modellieren wir als Kantengewicht. Wir stellen zwei verschiedene Algorithmen vor, die dieses Problem mit unterschiedlichen Restriktionen lösen. Wir haben somit das Problem der Schnittsuche auf ein Graphenproblem reduziert.

3. Im dritten Schritt setzen wir die bereits gefundenen Teilschritte zu einem gültigen Schnitt für die gesamte Zeichnung zusammen. Dafür durchlaufen wir den eben aufgebauten Hilfsgraphen mit einem dynamischen Programm, das den besten Weg vom Startknoten bis zum Zielknoten findet. Der Startknoten ist dabei das Rechteck, in dem sich der Startpunkt der Schnittlinie s später befinden soll. Analog ist der Zielknoten das Rechteck, in dem der Endpunkt t liegt.

4.1. Rechtecke

Wir betrachten zunächst die Vorgehensweise zur Unterteilung der Zeichnung in Rechtecke. Dabei ist anzumerken, dass wir keinen eigenen Datentyp für die Rechtecksflächen einführen. Stattdessen unterscheiden wir diese allein durch ihre linke Kante. Ein Rechteck ist somit die Fläche von der linken Kante bis zur linken Kante des nächsten, rechts gelegenen Rechtecks.

Wir unterteilen die Zeichnung zunächst in vertikale und horizontale Linien. Diese können entweder Seiten von Rechtecken oder Kantenabschnitte sein. Wir Teilen alle vertikalen Linien, die von einer horizontalen Linie geschnitten werden, oder aus denen eine horizontale Linie nach rechts entspringt, am Schnittpunkt in zwei Teile. Eine horizontale



(a) Ausgangszeichnung

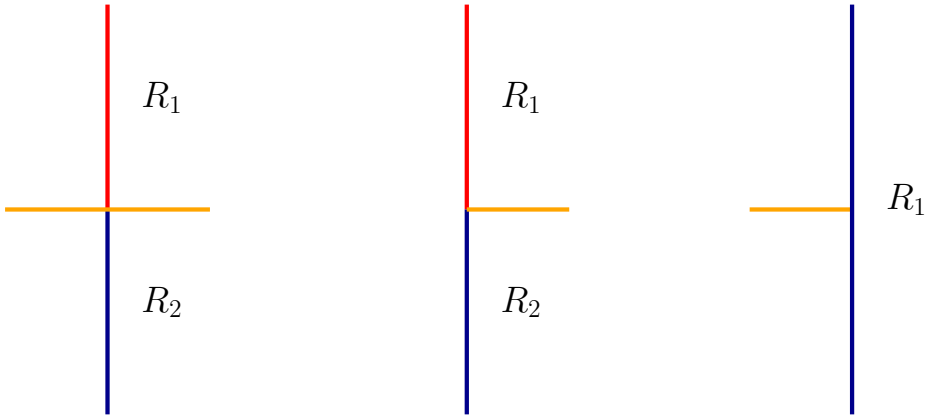
(b) Rechtecke

(c) Hilfsgraph

(d) Schnitt

Abb. 4.1.: Ein einfaches Beispiel für das Vorgehen bei der Suche eines Schnittes. In Abbildung 4.1b ist der Ausgangsgraph mit den eingezeichneten Rechtecken zu sehen. In Abbildung 4.1c ist der Hilfsgraph darübergelegt. Die Punkte repräsentieren die darunterliegenden Rechtecke. In gelb und rot hervorgehoben sind die Kanten des Hilfsgraphen. Für den roten Weg, der den größten gültigen Schnitt darstellt, sind die Kantengewichte angegeben. In Abbildung 4.1d sieht man den daraus resultierenden Schnitt, der ebenfalls in Rot gezeichnet wird. Die gestrichelte Linie ist die Schnittlinie und der semi-transparente Bereich darüber der weggeschnittene Bereich.

Linie, die nach links entspringt, wird nicht beachtet, da die Fläche rechts davon ungeteilt bleibt, und die vertikale Linie somit korrekterweise nur ein Rechteck beschreibt (Siehe Abbildung 4.2). Für die linke Seite eines Knoten fügen wir Hilfskanten hinzu, damit wir die Seite des Rechtecks nicht verlängern müssen und trotzdem sicher sein können, dass die Rechtecksflächen darüber und darunter eingefangen werden. Nun kann es jedoch sein, dass die vertikale Linie nicht die gesamte Höhe des Rechtecks umfasst. Wir müssen sie also verlängern, bis zur Grenze, an der das nächste Rechteck beginnt. Diese Grenze kann ein Knoten oder eine horizontale Linie sein. Wir fügen horizontale Linien über und unter der Zeichenfläche ein, um sicherzustellen, dass die Kanten nicht ins unendliche verlängert werden. Durch die Verlängerung kann es dazu kommen, dass es mehrere Linien mit gleicher Position gibt. Wir entfernen solche Dopplungen und führen erneut das Trennverfahren durch. Mithilfe dieses Prozesses haben wir den Graphen in Rechtecke mit maximaler Höhe eingeteilt. Wir tragen alle verbliebenen vertikalen Schnitte als Knoten in einen neuen Hifsgraphen ein. In Abbildung 4.3 ist zu sehen, wie ein Beispielgraph nach diesem Schema zerteilt wird.



(a) Linie wird geteilt (b) Linie wird geteilt (c) Linie wird nicht geteilt

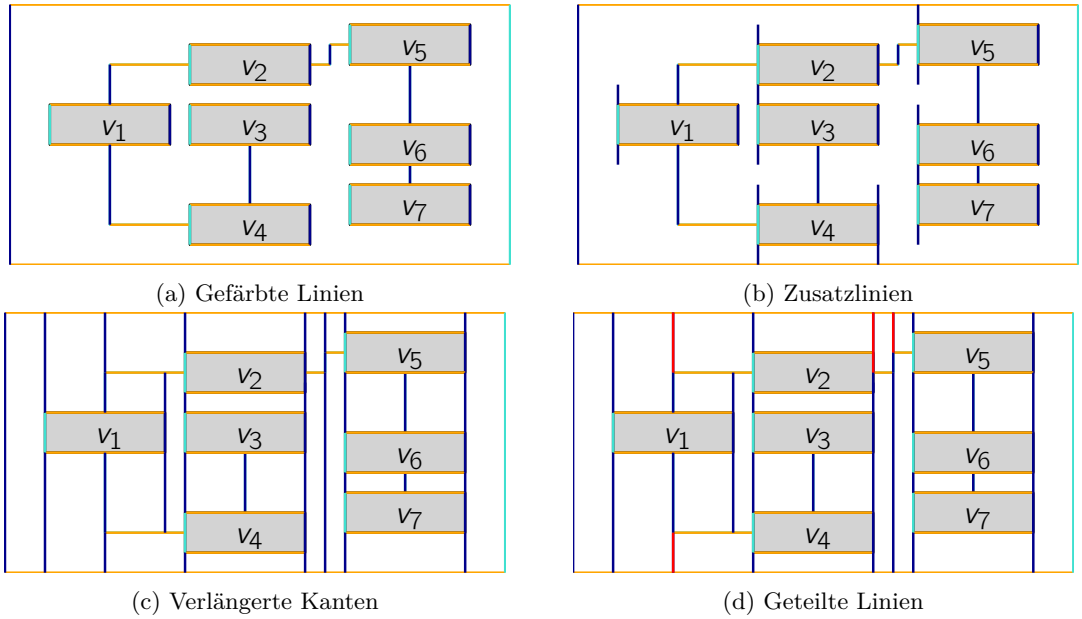
Abb. 4.2.: Die drei Möglichkeiten, wie sich eine horizontale und eine vertikale Kante schneiden können. In den beiden linken Fällen wird die vertikale Kante geteilt, da rechts davon zwei Rechtecke sein müssen. Dies ist bei der rechten Abbildung nicht der Fall, somit wird auch nicht geteilt.

4.2. Elementare Schnitte

Als nächstes suchen wir die maximale Größe, die ein gültiger Schnitt haben darf, der zwischen zwei Rechtecken verläuft. Aus diesen Werten wollen wir die Kanten unseres Hilfsgraphen bauen. Wir stellen dafür zwei unterschiedliche Verfahren vor.

4.2.1. SweepLine

Für das erste Verfahren welches wir vorstellen verwenden wir eine *SweepLine*, also eine vertikale Linie, die die Zeichnung von links nach rechts durchläuft. Die Linie hält auf allen x-Koordinaten, auf denen mindestens eine vertikale Linie liegt. Die SweepLine ist anfangs „ungefärbt“. Wenn sie auf eine vertikale Linie trifft, so „färbt“ sich die SweepLine in dem Bereich, in dem die vertikale Linie getroffen wurde. Treffen wir im späteren Verlauf des Sweeps auf eine weitere vertikale Linie, die sich mit einem eingefärbten Bereich überschneidet, so wissen wir, dass sich die zwei Rechtecke, die die vertikalen Linien repräsentieren an den Seiten überschneiden. Wir bestimmen die Höhe dieses Bereiches, indem wir die Größe der Überschneidungen zwischen der eingefärbten Fläche auf der SweepLine und dem vertikalen Abschnitt der Linie berechnen. Mit dieser Erkenntnis können wir eine gerichtete Kante in unseren Hilfsgraphen einzeichnen. Diese verläuft von der Kante, deren Farbe auf der SweepLine war, zu der, die soeben von der SweepLine getroffen wurde. Das Gewicht dieser Kante entspricht der Größe der Überschneidungsfläche. Ein Beispiel für den Ablauf des SweepLine Algorithmus ist im Anhang A zu sehen.



(a) Gefärbte Linien

(b) Zusatzlinien

(c) Verlängerte Kanten

(d) Geteilte Linien

Abb. 4.3.: Ein Beispiel für die Unterteilung eines einfachen Graphen in Rechtecke. Horizontale Kanten sind Orange hervorgehoben, vertikale Kanten in Blau. Türkis markiert vertikale Linien, die ein Rechteck abbilden, das nicht betreten werden darf. Der Unterteilung der Linien vor ihrer Verlängerung ist hier nicht nötig, da es keine Kantenkreuzungen in der Ausgangszeichnung gibt.

4.2.2. Kritische Abschnitte

Der obige Algorithmus findet bereits viele, aber noch nicht alle möglichen Schnitte zwischen Rechtecken. Bis jetzt kann ein Schnitt nur zwischen zwei Rechtecken verlaufen, wenn diese direkt nebeneinander liegen und sich überschneiden. Wie beispielsweise in Abbildung 4.4 zu sehen ist, gibt es jedoch noch weitere Schnitte, die sich über mehrere Rechtecke horizontal erstrecken und die wir noch nicht finden. Die Höhe der gefundenen Schnitte ist außerdem nicht immer optimal, da wir die Bedingung haben, dass Rechtecke, die sich vor dem Schnitt überlappen dies auch danach tun. Das bedeutet, übertragen auf die tatsächlich Graphzeichnung, dass ein vertikaler Kantenabschnitt nicht verschwinden darf. Eigentlich entscheidend in der Frage, wie groß ein Schnitt tatsächlich maximal sein darf ist jedoch, wann der vertikale Abschnitt einer Kante, die dadurch verlängert wird auf einen vertikalen oder horizontalen Kantenabschnitt trifft, der sich nicht verschiebt. Wir bezeichnen diesen Bereich als *kritischen Abschnitt*. Diese kritischen Abschnitte können wir in voraus ermitteln und dann für alle potenziellen Knotenpaare überprüfen, welche kritischen Abschnitte auf dem Weg dazwischen passiert werden. Das Minimum aus dem kleinsten kritischen Abschnitt und der Höhe des Ziel-Rechtecks beschränkt das Gewicht der Kante. Wir müssen dies nur für alle Rechteckspaare überprüfen, für die die rechte Kante des einen Rechtecks und die linke Kante des anderen auf der selben x Koordinate liegen. Mit diesem zweiten Verfahren können wir dann alle Schnitte nach

der obigen Definition finden. Die Struktur des Graphen wird jedoch potenziell stärker verändert als zuvor, was ein Nachteil sein kann.

Definition kritische Abschnitte Im Folgenden beschreiben wir die Eigenschaften der kritischen Abschnitte. Zuerst einmal stellen wir uns die Frage, wo kritische Abschnitte überhaupt auftreten können. Wir wissen, dass keine Hindernisse innerhalb der Rechtecke liegen, da diese sonst weiter unterteilt werden würden. Da Schnitte immer ohne Unterbrechung durch den gesamten Graphen verlaufen und wir keine horizontalen Linien schneiden dürfen (dazu zählen natürlich auch die oberen und unteren Kanten der Rechtecke) können Kanten im Hilfsgraphen nur von einem Rechteck zum nächsten verlaufen, wenn die rechte Kante des ersten Rechtecks und die linke Kante des zweiten auf derselben x-Koordinate liegen. Wir können unsere Suche also schon einschränken auf x-Koordinaten, an denen Rechtecke anfangen beziehungsweise enden. Alle horizontalen Kantenabschnitte in der Ausgangszeichnung, die eine solche Koordinate schneiden, bilden ein unüberwindbares Hindernis für Schnitte. Ein Schnitt kann weder von oben nach unten noch andersherum durch diese Linie, da sonst eine Hälfte der Kante verschoben würde, während die andere Hälfte sich nicht bewegt, weshalb wir die Kante durch einen vertikalen Abschnitt erweitern müssten, da ansonsten die Achsenparallelität nicht mehr gegeben wäre. Dies würde jedoch bedeuten, dass wir zwei neue Knicke in die Zeichnung einfügen, was wir aus Gründen der Lesbarkeit vermeiden wollen. Nehmen wir nun an, dass eine andere Kante der Ursprungszeichnung diese Koordinate schneidet. Dieses Mal macht sie jedoch eine Stufe auf der x-Koordinate und hat deshalb schon ein vertikales Verbindungsstück, welches die Kantenabschnitte rechts und links von der Koordinate verbindet. Hierdurch darf ein Schnitt verlaufen, der die rechte und linke Hälfte gegeneinander verschiebt. Die verbindende Linie kann einfach in ihrer Länge angepasst werden. Ein solcher Schnitt hat keine Größenbegrenzung. Befinden sich jedoch mehrere Kanten auf einer x-Koordinate, so muss sichergestellt werden, dass sich diese nicht durch Verschiebungen überschneiden und der Mindestabstand erhalten bleibt. Hierbei gibt es verschiedene Dinge zu beachten.

1. Alle kritischen Abschnitte, ausgenommen gerade Schnitte, sind gerichtet. Das bedeutet, dass es eventuell eine Rolle spielt, ob ein Schnitt den kritischen Abschnitt von oben oder von unten durchquert. Von oben bedeutet dabei, dass der linke Teil des Schnittes oberhalb des rechten ist und der Schnitt eine Stufe nach unten macht. Die andere Richtung verläuft analog.

2. Entscheidend bei der Suche eines kritischen Punktes, der einen Schnitt nach unten begrenzt, ist folgende Beschränkung: Wir betrachten einen beweglichen Kantenabschnitt, der die untersuchte x-Koordinate von links berührt. Diese Linie kann so lange nach oben verschoben werden, bis sie auf eine weitere Linie trifft die die x-Koordinate von rechts berührt. Dies gilt nur für Linienpaare die nicht durch eine vertikale Linie miteinander verbunden sind. Die Differenz der y-Positionen dieser Kantenabschnitte entspricht der maximalen Schnittgröße, die durch den kritischen Abschnitt passt. Ein solcher kritischer Abschnitt ist in Abbildung 4.7 zu sehen. Der

kritische Abschnitt gilt für alle Schnitte, die oberhalb oder auf der linken Teilkante anfangen und unterhalb der Rechten aufhören.

3. Suchen wir hingegen alle kritischen Abschnitte, die einen Schnitt nach oben beschränken, so betrachten wir alle Teilkanten, die die x-Koordinate von rechts berühren und die durch Verschiebung nach oben auf eine linke Teilkante treffen, die nicht mit ihr über einen vertikalen Kantenabschnitt verbunden ist. Die Grenze des kritischen Abschnitts ist erneut der Höhenunterschied der Beiden. Der Schnitt betrifft alle Kanten, die unterhalb der linken Teilkante eintreffen und oberhalb oder auf der Rechten weiterführen. Ein Beispiel für so einen kritischen Punkt ist in Abbildung 4.6 zu sehen.

Haben wir nun alle kritischen Punkte für eine bestimmte x-Position ermittelt, so können wir nun die Kanten zwischen den betroffenen Rechtecken ziehen. Wir legen fest, dass die Eingangshöhe des Schnittes die Unterkante des linken Rechtecks ist und, dass der Schnitt auch auf der Unterkante des rechten Rechtecks weiterläuft. Wir überprüfen für alle kritischen Abschnitte auf diese x-Koordinate, ob sie diesen Schnitt betreffen. Das Gewicht der Kante entspricht dann dem Minimum aus allen kritischen Abschnitten und der Höhe des Ziel-Rechtecks. Warum letzteres einen Einfluss auf die Größe der Kante haben kann, ist in Abbildung 4.5 zu sehen.

Variation der Kritischen Abschnitte Die obige Definition der kritischen Abschnitte maximiert die mögliche Schnittgröße. Wie wir bereits angedeutet haben, kann dies auch einen negativen Einfluss auf die Struktur der Zeichnung haben, da Kanten nun ihren Verlauf ändern. Beim SweepLine-Verfahren hat eine Kante stets ihre Knickpunkte beibehalten. Wenn eine Kante beispielsweise zuerst nach rechts geht, dann nach oben und dann wieder nach rechts, so würde eine Kante dieses Bewegungsmuster beibehalten. Nach der obigen Definition der kritischen Abschnitte könnte dieselbe Kante nach einer Stauchung zuerst nach rechts, dann nach unten und dann wieder nach rechts verlaufen. Diese Änderung kann eventuell die Lesbarkeit der Zeichnung verschlechtern. Wir können jedoch mit einer angepassten Definition der kritischen Abschnitte auch festlegen, dass die maximale Schnittgröße nur so groß ist, dass sich die gesamte Länge der Kante nicht verschlechtert oder dass sich die Richtung der Kantenknicke nicht ändert. Es wäre sogar möglich, die Beschränkung der kritischen Abschnitte so zu wählen, dass die Knicke in der Kante komplett verschwinden. Auf diese Weise könnte sich die Struktur der Zeichnung sogar vereinfachen gegenüber der Ausgangszeichnung.

4.3. Beweis

Wir wollen nun beweisen, dass wir mithilfe des eben beschriebenen Hilfsgraphen jeden gültigen Schnitt finden können. Wir beweisen zunächst, dass jeder gefundenen Pfad im Hilfsgraph einem gültigen Schnitt entspricht. Wir können eine Schnittlinie sehr einfach an der Unterseite der durchquerten Rechtecke entlangführen. Die Schnittlinie macht

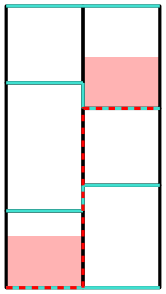


Abb. 4.4.: Ausschnitt aus einem Graphen durch den ein Schnitt verläuft, der mit dem ScanLine Algorithmus nicht gefunden wird. Die Graphkanten sind türkis, der Schnitt rot und die umliegenden Rechtecke in schwarz angedeutet.

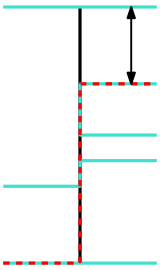


Abb. 4.5.: Ein Ausschnitt aus einer Zeichnung, der mehrere kritische Punkte enthält. Kanten in der Ausgangszeichnung sind türkis, Hilfslinien in schwarz und der Schnitt, dessen Maximalgewicht wir ermitteln wollen Rot. Der Pfeil stellt den begrenzenden Faktor des Schnittes dar. In diesem Fall die Höhe des Ziel-Rechtecks.

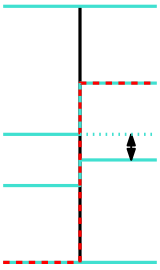


Abb. 4.6.: Verändern wir im Gegensatz zum vorherigen Beispiel den Verlauf einer Kante, so ist derselbe Schnitt nun in seiner Höhe viel eingeschränkter.

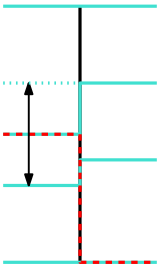


Abb. 4.7.: Verläuft in derselben Zeichnung der Schnitt nun nicht von oben nach unten, sondern von unten nach oben, so gibt es wieder einen anderen kritischen Punkt.

nur vertikale Bewegungen, wenn die Unterkanten zweier aufeinanderfolgenden Rechtecke nicht auf derselben Höhe liegen und wir diesen Unterschied ausgleichen müssen. Die so erhaltene Schnittlinie schneidet dabei wie vorgegeben keine horizontalen Kantenabschnitte oder Knoten, da wir dies durch die Definition der Rechtecke und durch die Beachtung aller kritischen Abschnitte bereits garantieren können. Dass es sich zudem um einen gültigen Schnitt handelt, wissen wir ebenfalls durch die Begrenzung der Kantengewichte mithilfe der Rechteckshöhe und der kritischen Abschnitte.

Wir zeigen nun, dass jeder gültige Schnitt durch den Hilfsgraphen dargestellt werden kann. Wir wissen bereits, dass eine Schnittlinie sich nur innerhalb eines Rechtecks horizontal bewegen kann, da alle Flächen, die nicht von einem Rechteck belegt sind, automatisch Knoten sein müssen. Wir wissen außerdem, dass eine Schnittlinie y-Monoton ist und sich somit vertikal nur zwischen zwei Rechtecken verlaufen kann, wenn das eine Rechteck auf derselben x-Koordinate endet, auf der das Nächste anfängt. Für die vertikale Bewegung der Schnittlinie unterscheiden wir zwei Fälle.

1. Wenn die Schnittlinie innerhalb eines Rechtecks eine vertikale Bewegung macht, so müssen wir diese nicht beachten, da sie keinen Einfluss auf die späteren Schnittmengen hat.

2. Für vertikale Bewegungen zwischen Rechtecken haben wir bereits über die kritischen Abschnitte alle Bewegungs-Beschränkungen aus der Definition der gültigen Schnitte erfasst.

Wir wissen somit bereits, dass wir den Verlauf der Schnittlinie im Hilfsgraphen darstellen können. Die Größe eines gültigen Schnittes kann ebenfalls nie größer sein als der besagte Weg durch den Hilfsgraphen, da wir für dessen Konstruktion mithilfe der kritischen Abschnitte und der Berücksichtigung der Rechteckshöhe alle Abstände in der Zeichnung berücksichtigen.

4.4. Wegfindung

Wir haben das Problem der Schnittsuche nun erfolgreich auf eine Pfadsuche im Hilfsgraphen reduziert. Dieser Hilfsgraph ist in beiden Fällen ein gerichteter, kreisfreier st-Graph.

Für die Wegsuche wollen wir das kleinste Kantengewicht auf einem Pfad maximieren, um einen Schnitt zu erhalten, der maximal groß ist, aber immer noch durch alle Engpässe passt. Bevor wir jedoch mit der eigentlichen Pfadsuche beginnen, entfernen wir alle Kanten aus dem Hilfsgraphen deren Gewicht kleiner als der Mindestabstand ist, da kein sinnvoller Schnitt über diese Kanten verlaufen kann.

Für die Pfadsuche verwenden wir ein dynamisches Programm, das dem Dijkstra Algorithmus ähnelt. Für jeden Knoten im Hilfsgraphen wird gespeichert, in wie vielen Schritten er erreichbar ist, wie groß die Summe der Rechteckshöhen auf dem Weg ist, wie groß das minimale Kantengewicht auf dem Weg dahin war und welcher Knoten sein Vorgänger auf dem Weg ist. Die Anzahl der Schritte wird mit ∞ initialisiert und das minimale Kantengewicht mit $-\infty$. Die einzige Ausnahme ist der Startknoten s. Die Distanz vom Startknoten ist 0, die Summe der Pfadhöhe ist $h(\Gamma)$ (also die Höhe der Zeichnung) und der größtmögliche Schnitt ist ∞. Es wird eine Prioritätsliste verwaltet, die alle Knoten nach ihrem minimalen Kantengewicht und der Pfadhöhe geordnet enthält. Am Anfang eines Zyklus wird der oberste Knoten aus der Prioritätsliste genommen und alle Kanten, die damit verbunden sind, abgelaufen. Für die Knoten, die darüber verbunden sind, wird überprüft, ob der Weg, der über den Knoten aus der Prioritätsliste läuft, besser ist als der bisher gespeicherte. Ein besserer Weg hat entweder einen höher gelegenen Schnitt (nach der durchschnittlichen Rechteckshöhe) bei gleicher Schnittgröße oder eine größere Schnittgröße. Sofern dies der Fall ist, wird der für den erreichten Knoten gespeicherte Weg aktualisiert ebenso wie die Prioritätsliste. Die Prioritätsliste wird nach diesem Schema abgearbeitet, bis keiner der Einträge eine Schnittgröße hat, die besser ist als die Schnittgröße des Zielknotens. Sobald dies der Fall ist können wir sicher sein, dass es keinen besseren Pfad mehr geben kann, da sich die Schnittgröße nur noch verschlechtern kann. Da wir einen Binärbaum für die Verwaltung der Prioritätsliste verwenden beläuft sich die Laufzeit der Pfadfindung auf $O(|E| \log(|V|) + |V|)$.

Anmerkung: Wir haben die Priorisierung von höher verlaufenden Schnitten gewählt, um zu vermeiden, dass vermehrt Rechtecke an der Unterseite der Zeichnung verwendet werden, wodurch weniger Knoten bewegt werden würden. Eine alternative Implementierung könnte zum Beispiel Schnitte bevorzugen, die möglichst mittig verlaufen. Wenn wir annehmen, dass die Knoten vertikal gleichmäßig über die Zeichnung angeordnet sind, könnten wir somit priorisieren, dass die beiden Schnittmengen etwa gleichgroß sind, wodurch sich der durchschnittliche Abstand von Knoten mehr reduzieren könnte.

Implementation Die Implementation des oben beschriebenen Algorithmus ist über die GitLab-Seite der Universität Würzburg [Kö23] mit einem gültigen Account einsehbar.

5. Auswertung

Nachdem der Algorithmus nun feststeht, wollen wir ihn anhand einiger Metriken auswerten. Wir verwenden dafür einen Datensatz aus 1140 Pseudo-Kabelplänen, die mit dem iPraline Algorithmus gezeichnet wurden. Es handelt sich dabei um denselben Datensatz, der für die Auswertung von iPraline verwendet wurde, mit dem Unterschied, dass alle Hyperkanten durch Sterngraphen ersetzt wurden. Die Pseuto-Kabelpläne bestehen aus 380 echten Kabelplänen, die aus einem großen, deutschen Maschinenbauunternehmen stammen und die wiederum zu 1140 Pseudo-Plänen verfremdet wurden. Die genaue Vorgehensweise bei diesem Prozess wird von Zink et al. [ZWBW22] beschrieben. Der originale Datensatz ist bei GutHub [ZWBW20b] verfügbar und der Datensatz mit entfernten Hyperkanten ist im GitLab-Projekt zu dieser Arbeit hinterlegt [Kö23]. Die verwendeten Graphen haben im Durchschnitt 107 Knoten und 277 Kanten. Die minimale Knotenanzahl ist 2 und die Maximale 386. Die Kantenzahl liegt in einem Bereich von 3 bis 1380. Kanten in den Graphen haben durchschnittliche $1,7$ Kantenknicke und ein Graph hat durchschnittlich 918 Kantenkreuzungen.

Alle Tests wurden in Java auf einem Intel© Core™ i5-7300U mit 4 (parallel genutzten) Threads und 16GiB Arbeitsspeicher unter Linux durchgeführt und haben etwa 4 Stunden beansprucht. Wir verwenden für alle Tests die Schnittfindung mittels SweepLine. Von den beiden vorgestellten Algorithmen ist dieser der restriktivere, der die geringere Stauchung liefert. Die Zeichnung wurde, sofern nicht anders angegeben, immer zuerst in der Höhe und danach in der Breite gestaucht.

5.1. Zeichenfläche

Platzersparnis Das wichtigste Maß zur Beurteilung der Funktionalität eines Stauchungsalgorithmus ist die eingesparte Zeichenfläche. Diese soll möglichst groß sein, um die Zeichnung so weit wie möglich zu verkleinern. Wir betrachten dafür die *Bounding Box* der Zeichnung. Diese ist definiert als das kleinste achsenparallele Rechteck, das den Graphen vollständig enthält. Wir messen die Stauchung des Algorithmus anhand der relativen Verkleinerung der Fläche der Bounding Box. Formal definieren wir die *Platzersparnis* als

$$pe(\Gamma, \Gamma') = 1 - \frac{w'_{bb} \cdot h'_{bb}}{w_{bb} \cdot h_{bb}},$$

wobei Γ die Zeichnung vor der Stauchung und Γ' die Zeichnung danach ist. w_{bb} ist die Breite der Bounding Box und h_{bb} die Höhe. Eine weitere Alternative ist, anstatt dem kleinsten Rechteck die kleinste konvexe Hülle zu betrachten. Hiermit kann die tatsächlich

von der Zeichnung eingenommene Fläche genauer beschrieben werden. Wir haben uns jedoch bewusst gegen diese Variante entschieden, da sie wenig Aussagekräftig ist, wenn es darum geht, ob ein Layout auf einem Bildschirm angezeigt werden kann.

Im Abbildung 5.1 sieht man, dass die Verkleinerung der Bounding Box von der Anzahl der Knoten im Graphen abhängig ist. Bei weniger als 50 Knoten ist die Stauchung noch tendenziell niedriger, da in kleineren Graphen meist weniger Freiraum ist, der über den Mindestabstand hinaus geht. Im Bereich von 50 bis 120 Knoten ist die Platzeinsparung am größten mit durchschnittlich etwa 35%. Bei noch größeren Graphen sinkt der Wert hingegen wieder. Dieser Effekt kommt vermutlich von der höheren Komplexität der Kanten, die das Finden von Schnitten erschwert. Die Anzahl der Kantenkreuzungen, die ein gutes Maß für die Komplexität der Kanten ist, steigt proportional zur Kantenanzahl mit einer Korrelation von $0,996$. Das Arithmetische Mittel der Platzersparnis ist 31%.

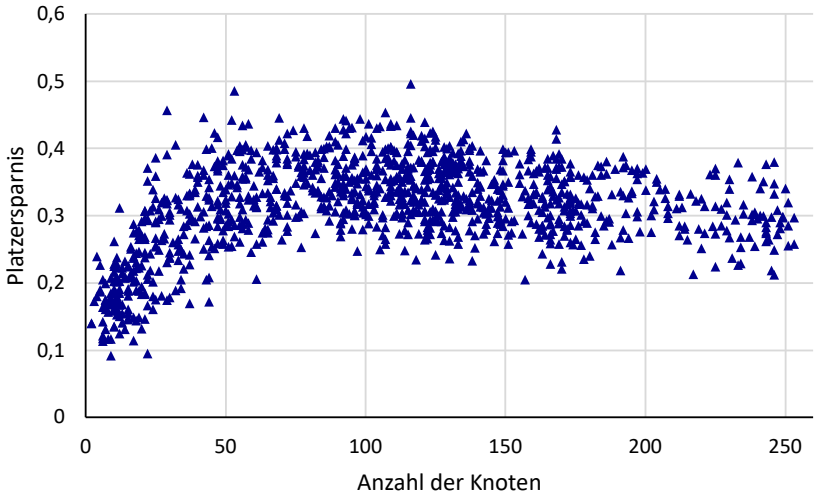


Abb. 5.1.: Platzersparnis in Relation zur Knotenanzahl.

Breite und Höhe Wir unterscheiden nun zwischen der Stauchung in Höhe und Breite. Es zeichnet sich dabei grundsätzlich derselbe Verlauf der Werte ab. Was jedoch auffällt ist, dass die Stauchung in der Höhe (∅21%) deutlich größer ist als die Stauchung in der Breite (∅13%). Dafür gibt es zwei Erklärungen. Zum einen ist der horizontale Abstand von Knoten in der Ausgangszeichnung geringer als der vertikale. Das liegt an der Ebenenstruktur der Layouts. Knoten, die auf derselben Ebenen liegen werden meist relativ nah aneinander gezeichnet. Der vertikale Abstand zwischen zwei Ebenen ist hingegen höher, da dort alle Kanten untergebracht werden müssen. Außerdem können bei allen iPraline Zeichnungen die Kanten nur an der Ober- und Unterseite eines Knotens angebracht werden. Dadurch passiert es häufiger, dass zwei Knoten mit einer gerade vertikalen Linie verbunden sind. Horizontale Schnitte können diese Kanten problemlos schneiden, vertikale hingegen nicht.

Eine weitere Theorie ist, dass die asymmetrische Stauchung durch die Reihenfolge der

Stauchung bedingt ist. Um dies zu verifizieren haben wir den Algorithmus angepasst, sodass er zuerst in der Breite und anschließend in der Höhe staucht. Wir konnten dabei jedoch keine Veränderung der Breiten- und Höhenstauchung feststellen. In der Knotenausrichtung und Ebenengröße konnten minimale Verbesserungen festgestellt werden. Diese sind jedoch vernachlässigbar klein (1%).

Die asymmetrische Stauchung erscheint auf den ersten Blick womöglich als Nachteil. Es kann jedoch auch vorteilhaft sein, wenn die Zeichnung ein breiteres Format einnimmt. Da übliche Seitenverhältnisse für digitale Anzeigegeräte meistens Breitbildformate wie 16:9 (1,77:1) oder 16:10 (1,6:1) sind können breitere Graphenzeichnungen in vielen Fällen besser angezeigt werden. Der iPraline Algorithmus strebt ein Seitenverhältnis von 1:1 an. Eine Untersuchung der Ausgangszeichnungen ergibt, dass das durchschnittliche Seitenverhältnis tatsächlich 1,19:1 ist. Nach der Stauchung ist der Mittelwert 1,31:1. Allein durch diese Veränderung der Seitenverhältnisse können wir die Zeichnung durchschnittlich 10% größer auf einem 16:9 Bildschirm anzeigen, was die Lesbarkeit bereits verbessern kann.

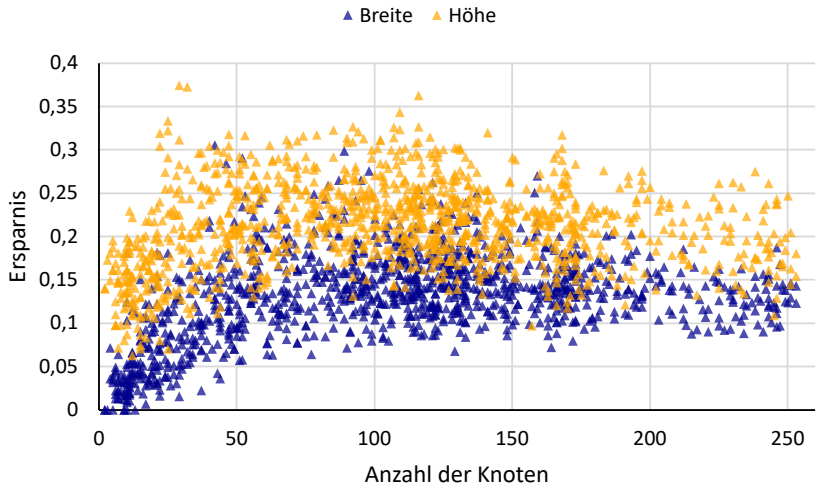


Abb. 5.2.: Stauchung in Höhe und Breite in Relation zur Knotenanzahl.

5.2. Kantenstauchung

Als nächstes betrachten wir die relative Veränderung der durchschnittliche Kantenlänge. Kürzere Kanten können für eine übersichtlichere Graphenzeichnung sorgen, da ein Betrachter, der den Weg einer Kante nachverfolgt eine geringere Strecke zurücklegen muss und somit schneller den Zielknoten findet. Wir nennen diese Metrik *Kantenstauchung* und definieren sie als

$$ks(\Gamma, \Gamma') = 1 - \frac{\sum_{e' \in E'} |e'|}{\sum_{e \in E} |e|},$$

wobei $|e|$ die Länge der Kante e und $|E|$ die Anzahl der Kanten in der Kantenmenge ist. Es werden möglichst große Werte angestrebt, um die Kantenlänge zu minimieren. Wir sehen in Abbildung 5.3, dass sich hier ein ähnlicher Verlauf wie bei der Bounding Box abzeichnet. Was jedoch auffällig ist, ist dass die Korrelation zwischen Kantenstauchung und der Platzersparnis mit $0,57$ kleiner ist als erwartet. Für eine Stauchung des Graphen ist schließlich eine Stauchung der Kanten zwingend notwendig. Ein Beispiel, warum die beiden Messwerte nicht direkt zusammenhängen, ist in Abbildung 5.4 zu sehen.

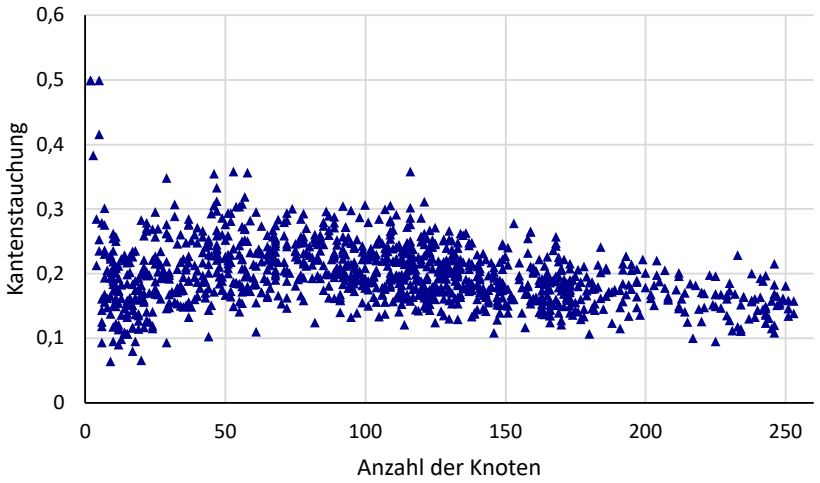


Abb. 5.3.: Kantenstauchung in Relation zur Knotenanzahl

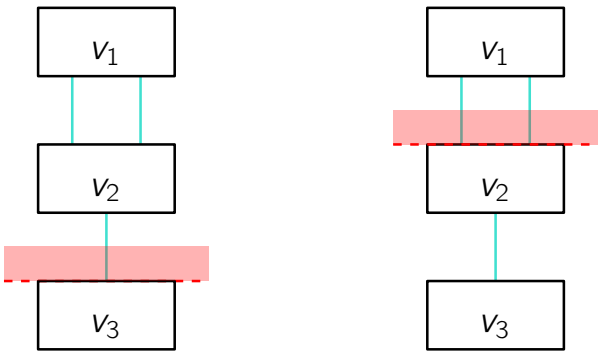


Abb. 5.4.: Zwei gültige Schnitte durch eine Zeichnung die dieselbe Stauchung verursachen aber einen unterschiedlichen Einfluss auf die durchschnittliche Kantenlänge haben.

5.3. Ebenen

Die nachfolgenden Metriken befassen sich mit der Erhaltung der Ebenen-Struktur der iPraline Zeichnungen.

Knotenausrichtung Wir stellen uns zunächst die Frage, welcher Anteil der Knoten in der Zeichnung mit mindestens einem anderen Knoten auf einer gemeinsamen Ebene liegen. Wir nennen solche Knoten *ausgerichtet*. Wir wollen eine möglichst große Anzahl an ausgerichteten Knoten erreichen, da eine Einteilung der Knoten in Ebenen zu übersichtlicheren Zeichnungen führt. Der Kehrwert aus dieser Metrik ist die Anzahl der Knoten, die keiner Ebene zugewiesen sind. Wir definieren die Knotenausrichtung als

$$ka(\Gamma) = \frac{\sum_{v \in V} L(v)}{|V|}$$

$$L(v) = \begin{cases} 1, & \text{falls ein anderer Knoten auf derselben y-Koordinate wie } v \text{ liegt} \\ 0, & \text{ansonsten} \end{cases}$$

In der Abbildung 5.5 sehen wir, dass der Wert für die Ausgangszeichnungen sehr häufig dem Idealwert von 1 entspricht. Nach der Stauchung hingegen fällt der Wert von einem Median von $0,98$ auf $0,79$. Die Ebenenstruktur wird bei größeren Graphen eher beibehalten als bei kleineren. Dafür gibt es eine einfache Begründung: Die Wahrscheinlichkeit, dass ein zufällig verschobener Knoten mit einem anderen Knoten auf derselben Ebene liegt, steigt mit der Anzahl der Knoten, da die Knotenanzahl pro Zeichenhöhe steigt (Siehe Abbildung 5.6). Ein weiterer Grund ist, dass bei größeren Zeichnungen auch die Anzahl der Knoten pro Ebene größer ist. Wird eine Ebene mit vielen Kanten in der Mitte geschnitten, so haben die beiden Hälften immer noch mehrere Knoten. Mehr dazu im nachfolgenden Abschnitt.

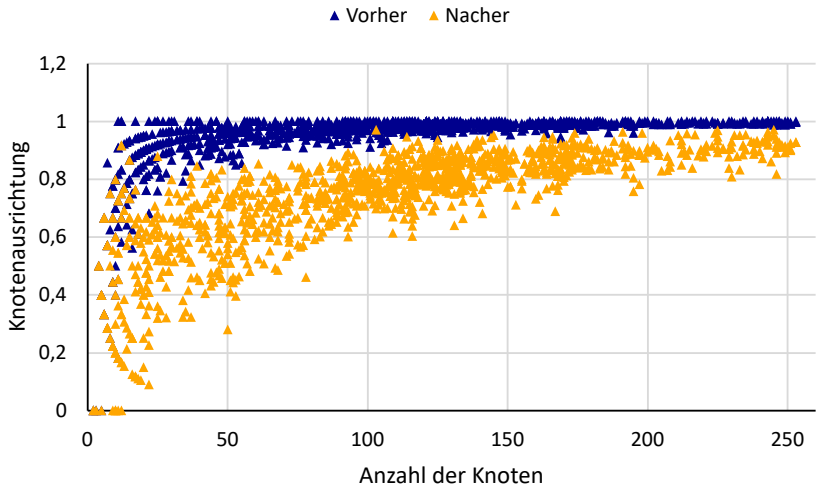


Abb. 5.5.: Knotenausrichtung in Relation zur Knotenanzahl.

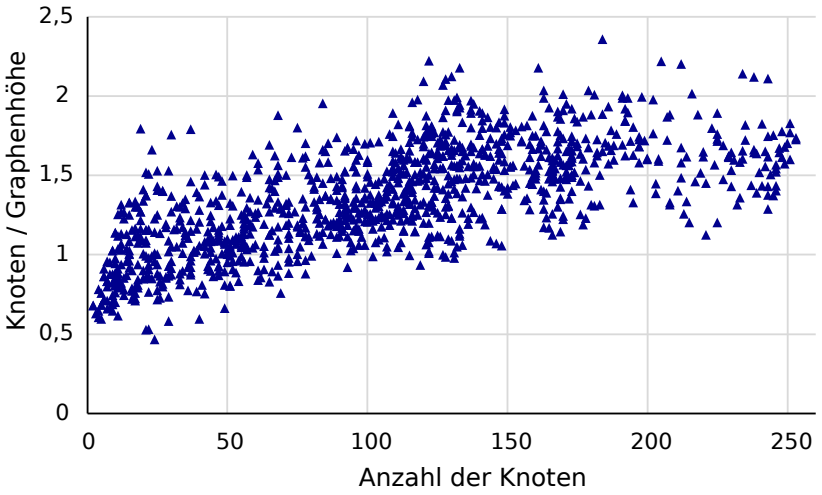


Abb. 5.6.: Die Knotenanzahl geteilt durch die Höhe der Zeichnung in Vielfachen der Knotenhöhe in Relation zur Knotenanzahl.

Ebenengröße Wir wollen nun betrachten, wie sich die durchschnittliche Anzahl der Knoten pro Ebene verändert. Eine große durchschnittliche Anzahl von Knoten pro Ebene bedeutet im Umkehrschluss eine niedrige Anzahl an Ebenen, was gut für die Lesbarkeit der Zeichnung sein kann. Wir nennen diese Metrik eg (Ebenengröße) und definieren sie als

$$eg(\Gamma) = \frac{|V|}{|Ebenen(V)|}$$

Die Größe der Ebenen wird durch die Stauchung drastisch verkleinert. Im vorherigen Abschnitt haben wir jedoch gesehen, dass die Anzahl der Knoten, die sich auf Ebenen befinden immer noch bei 79% liegt. Der Grund dafür ist, dass Knoten oftmals nicht einzeln aus einer Ebene verschoben werden. Stattdessen teilt sich beispielsweise eine Ebene mit vier Knoten in zwei Ebenen mit je zwei Knoten. Dadurch verändert sich die Knotenausrichtung nicht, aber die Anzahl der Knoten pro Ebene sinkt.

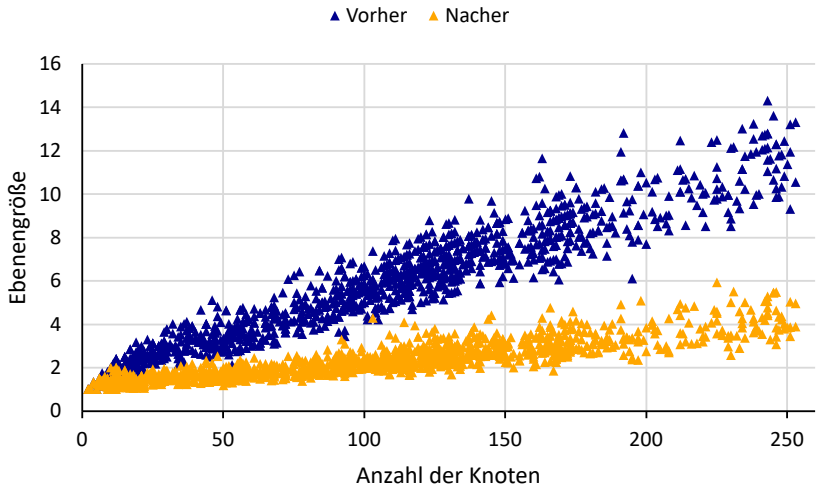


Abb. 5.7.: Ebenengröße in Relation zur Knotenanzahl.

Ausrichtungs-Abweichung Wie wir bereits gesehen haben behalten viele der Knoten nicht ihre Position in der Anfangsebene bei. Wir untersuchen nun, wie weit sich die Y-Koordinaten eines Knotenpaares durchschnittlich unterscheiden, wenn sie vor der Stauchung auf einer Ebene waren. Da der Versatz in Pixelanzahl wenig anschaulich ist teilen wir den Wert durch die Knotenhöhe. Diese ist in iPraline Zeichnungen für alle Knoten gleich. Wir nennen diese Metrik *Ausrichtungs-Abweichung* und definieren sie als

$$aa(\Gamma, \Gamma') = \frac{\sum_{i=1}^{|V|} \sum_{j=1}^{|V|} L(v_i, v_j) \cdot |v_i' - v_j'|}{\sum_{i=1}^{|V|} \sum_{j=1}^{|V|} L(v_i, v_j)} \div Knotenhöhe(\Gamma)$$

$$L(v_1, v_2) = \begin{cases} 1, & \text{falls } v_1 \text{ auf derselben y-Koordinate wie } v_2 \text{ liegt} \\ 0, & \text{ansonsten} \end{cases}$$

Im Diagramm 5.8 sehen wir, dass in mehr als 58% der getesteten Graphen die Ausrichtungs-Abweichung kleiner als zwei Knotenhöhen ist und in etwa 19% der Fälle sogar geringer als eine Knotenhöhe ist. Besonders in diesen Zeichnungen vermuten wir nur eine geringe Beeinträchtigung der Wiedererkennbarkeit von Strukturen. Im Abschnitt 6 gehen wir auf Möglichkeiten ein, die Ausrichtungs-Abweichung weiter zu verbessern.

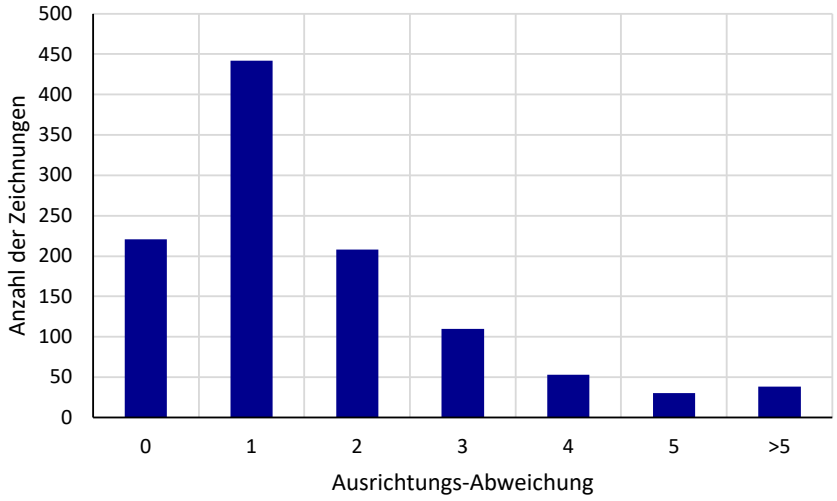


Abb. 5.8.: Ausrichtungs-Abweichung in Relation zur Knotenanzahl.

5.4. Laufzeit

Bei den folgenden Angaben ist zu beachten, dass der Algorithmus noch nicht hinsichtlich der Laufzeit optimiert wurde. Die Laufzeit wurde über die zugeteilte CPU-Zeit der einzelnen Threads berechnet (Pro Thread wird genau eine Zeichnung bearbeitet). Der gemessene Median der Laufzeit beträgt $8,5$ Sekunden. Die meiste Zeit verbringt der Algorithmus damit, die Wege durch den Hilfsgraphen zu finden. Eine weitere Beobachtung zur Laufzeit, die wir machen konnten, ist, dass die Suche nach Wegen am Ende des Algorithmus mit sinkender Schnittgröße erheblich länger werden kann. Dafür gibt es eine einfache Erklärung: Wie im Abschnitt 4.4 beschrieben ist bricht der Suchalgorithmus ab, sobald es keine Knoten in der Prioritätsliste gibt, deren Schnittgröße gleich oder größer ist als der beste bis dahin gefundene Schnitt. Da es besonders in großen Zeichnungen auch sehr viele kleine Schnitte gibt muss der Algorithmus auch mehr Knoten durchlaufen, bis er abbrechen kann. Mit der aktuellen Laufzeit ist der Algorithmus noch für wenige Anwendungsfälle brauchbar. Durch die Optimierung des vorhandenen Codes und die Implementierung der in Abschnitt 6 vorgeschlagenen Laufzeitverbesserungen könnte sich dies jedoch sehr schnell ändern.

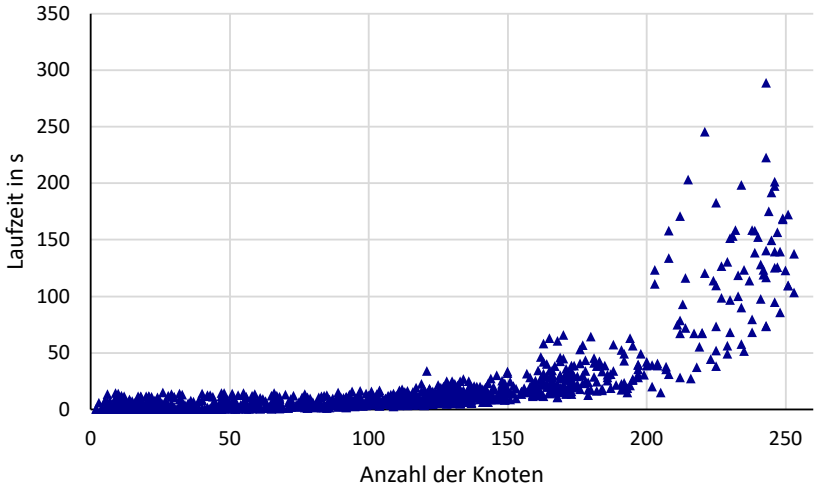
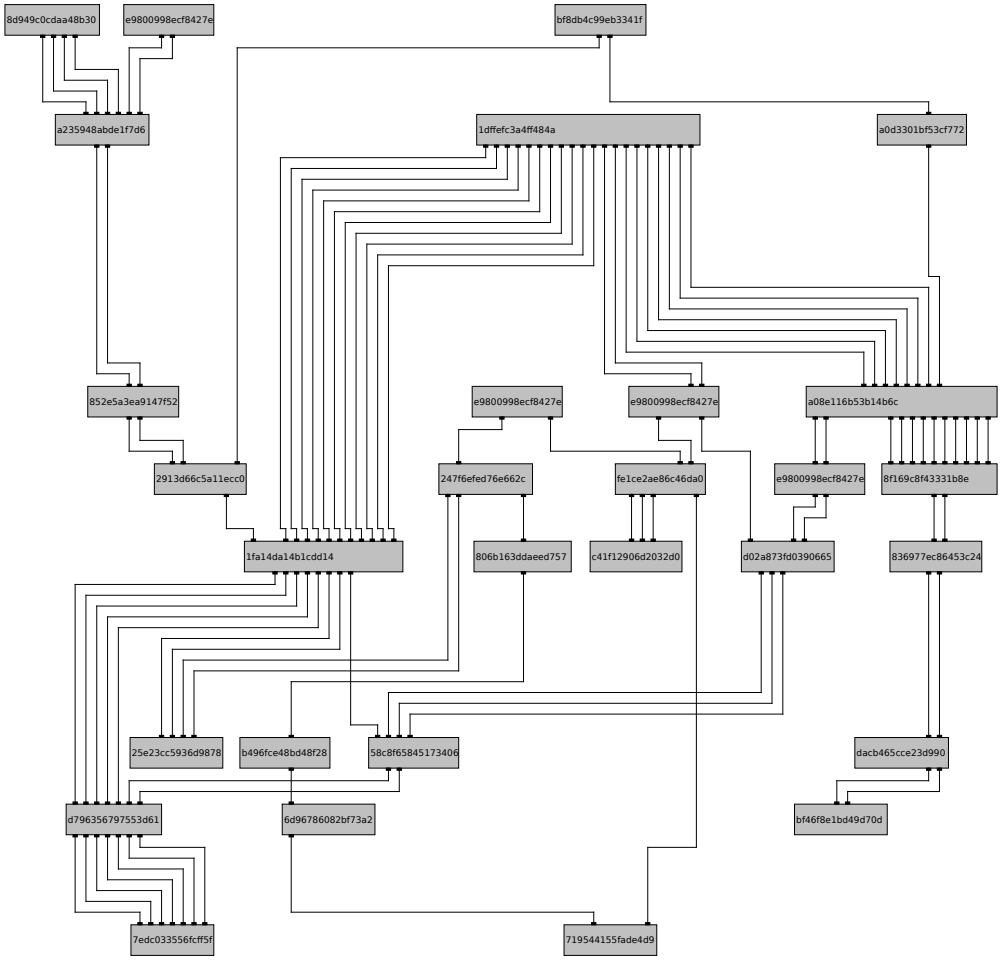


Abb. 5.9.: Die Laufzeit des Algorithmus gemessen in Sekunden.

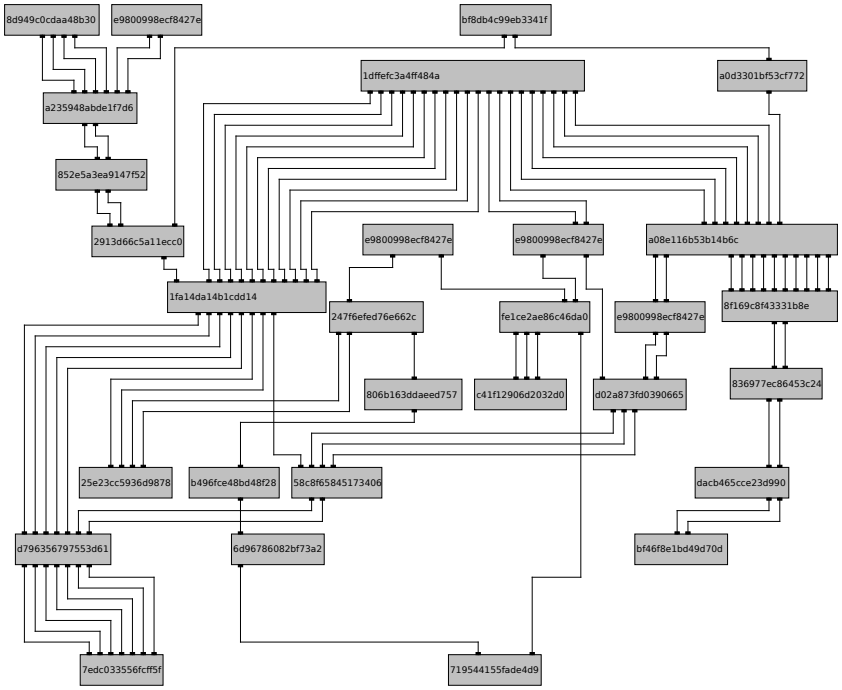
5.5. Visuelle Evaluation

Wir haben bereits die wichtigsten Kennzahlen des Algorithmus vorgestellt. Bei einem Algorithmus, der für die Zeichnung von Graphen eingesetzt wird ist eine visuelle Evaluation jedoch genau so wichtig. Wir betrachten einen Graphen mit 29 Knoten und 261 Kanten. Er ist damit einer der kleineren Graphen im Datenset. Die Platzersparnis durch die Stauchung ist 40%. Er wird in der Höhe um 28% und in der Breite um 16% gestaucht und liegt damit über dem Durchschnitt. Die Kantenlänge wird um 30% verringert, was vor allem daran liegt, dass die großen Multikanten[1] sehr stark gestaucht werden. Die Knotenausrichtung ist zuvor 1 und nachher $0,79$. Die Ebenengröße sinkt von $3,6$ auf $2,0$. Das liegt daran, dass nach der Stauchun 5 Knoten keiner Ebene angehören. Die Ebenenabweichung beträgt $1,7$. Unser subjektive Eindruck der Ebenenstruktur ist jedoch, dass sie sich nicht wesentlich verschlechtert hat. Der Grund dafür ist, dass die Knoten in der Mitte der Zeichnung, die zuvor auf einer Ebene lagen, danach kaum verschoben sind. Die Knoten oben links, die sich dagegen viel mehr von ihrer Ursprungsebene wegbewegt haben, hatten auch davor keinen logischen Zusammenhang zu ihrer Ebene und sind an ihrer neuen Position mindestens genau so übersichtlich platziert. In diesem Beispiel erzielt der Stauchungsalgorithmus ein gutes Ergebnis und es ist nach wie vor sehr einfach sich in der Zeichnung zu orientieren. Höchstens die Kantenstruktur könnte noch vereinfacht werden. Dabei könnte wiederum die Verwendung des Algorithmus mit kritischen Abschnitten helfen. Einige beispielhafte Schnitte in diesem Beispielgraphen sind in Abbildung B.8 zu sehen.

[1] Multikanten sind Kanten, die mehrfach zwischen zwei Knoten verlaufen.



(a) Die iPraline Zeichnung



(b) Die gestauchte Zeichnung

5.6. Sonstige

Abgesehen von den bereits genannten Metriken haben wir die Varianz der Kantenlänge untersucht. Hier konnten wir keine signifikante Veränderung zum Ausgangswert feststellen. Wir haben uns außerdem dagegen entschieden die orthogonale Ordnung der Knoten genauer zu betrachten, da die Ergebnis-Werte ohne einen Referenzwert kaum informativ sind. Den wichtigsten Kern der orthogonalen Ordnung beschreiben wir außerdem bereits durch den Ebenen-Anteil in Abschnitt 5.3.

6. Ausblick

Im Verlauf der Arbeit und besonders im letzten Abschnitt haben wir die Stärken und Schwächen unseres Algorithmus ausgearbeitet. Wir wollen im Folgenden mögliche Lösungsansätze für einige der gefundenen Probleme vorstellen. Die geschichtete Struktur des Algorithmus macht es uns möglich durch das Austauschen oder Erweitern einiger Funktionalitäten einfach maßgebliche Änderungen am Algorithmus vorzunehmen. Die folgenden Vorschläge sind in mehrere Abschnitte gegliedert:

Constraints Wie in der Auswertung bereits erwähnt, behält der Algorithmus nur schlecht bis gar nicht die Ebenen in der Zeichnung bei. Die Ebenenstruktur vollständig während der Stauchung zu erzwingen wäre jedoch kontraproduktiv, da das Ausmaß der Stauchung dadurch minimal wäre. Eine mögliche Alternative wäre es, nur in wichtigen Fällen entsprechende Beschränkungen zu setzten. Dies könnte manuell oder nach einer festgelegten Heuristik geschehen. Weiterhin wäre auch eine iterative Implementierung denkbar. Man könnte die Zeichnung zunächst ohne Constraints stauchen und anschließend problematische Knotenpaare identifizieren. Beispielsweise solche, deren Ausrichtungs-Abweichung sehr stark über dem Durchschnitt liegt. Zwischen solchen Knoten könnte man ein Constraint einfügen und den Algorithmus erneut durchlaufen lassen, bis die Ausrichtungs-Abweichung einem gewünschten Zielwert entspricht.

Möglichkeiten solche Constraints umzusetzen sind bereits im Algorithmus vorhanden. Wir können uns zu Nutze machen, dass der Algorithmus keine neuen Kantenknicke hinzufügt. Horizontale Kanten können deshalb nicht geschnitten werden. Wir müssen folglich nur eine gerade, horizontale Hilfskante zwischen zwei Knoten oder Bereichen legen, die nicht gegeneinander verschoben werden sollen. Ein großer Nachteil dieser Variante ist jedoch, dass ein Schnitt in dem Bereich zwischen den beiden Knoten entweder nur über oder nur unter der Hilfskante verläuft. Damit schränken wir die möglichen Schnitte eventuell stark ein. Besser wäre es, wenn nur die beiden Knoten auf derselben Ebene bleiben würden und der Schnitt ansonsten frei verlaufen kann.

Dies könnte mit einem angepassten Wegfindungs-Algorithmus umgesetzt werden. Hier wäre ein möglicher Ansatzpunkt für die Weiterentwicklung des Algorithmus.

Laufzeit Wie wir im Abschnitt 5.4 bereits etabliert haben, ist die Laufzeit ein weiterer Schwachpunkt des Algorithmus. Als Ursache dafür haben wir den Wegfindungs-Algorithmus identifiziert. Wir können den Algorithmus auf zwei Weisen verbessern:

1. Indem wir die Anzahl der Iterationen verringern. Wir schlagen dafür zwei Möglichkeiten vor: Der Algorithmus könnte erlauben, dass sich Schnitte in zwei Wege aufteilen und wieder zusammensetzen können. Weiterhin könnte der Algorithmus

mehrere unabhängige Wege gleichzeitig finden. Für beide dieser Ansätze müssten jedoch sehr häufig die Wege durch den Graphen zurückverfolgt werden, was sich wieder negativ auf die Laufzeit des Algorithmus auswirken würde. Eine Möglichkeit, diesen Prozess etwas zu vereinfachen wäre es, statt nur dem Vorgängerknoten im gefundenen Weg, den gesamte Weg für jeden Knoten zu speichern. Dadurch würde sich wiederum der Speicherverbrauch des Algorithmus stark erhöhen. Es bleibt zu untersuchen, in welchem Maß diese Erweiterungen des Algorithmus die Gesamtlaufzeit verbessern, oder ob sie sich nicht sogar negativ darauf auswirken würden.

2. Indem wir die Laufzeit pro Iteration verbessern. Durch die Verwendung eines dynamischen Programms mit frühzeitiger Abbruchbedingung haben wir uns erhofft, im Vergleich zu einer Implementiertung via Breitensuche, eine schnellere Laufzeit in reellen Anwendungsfällen zu bekommen, da die eingesparte Laufzeit durch den früheren Abbruch die zusätzliche Laufzeit durch die Prioritätsliste übertrifft. Die tatsächlich gemessene Laufzeit lässt uns diese Vermutung anzweifeln. Es wäre besonders im Bezug auf die Worst-Case-Laufzeit sehr interessant, die zwei Verfahren zu vergleichen.

Stauchung Um eine noch höhere Stauchung des Graphen zu erreichen, könnten wir einführen, dass ein Schnitt auch über zusätzliche Kanten mit unendlicher Größe verlaufen kann, die unterhalb der Zeichnung angeordnet sind. Damit könnte das Problem umgangen werden, dass die Summe aller Schnittgrößen durch ein Bottleneck in der Zeichnung (also einen kleinsten Schnitt im Hilfsgraphen) beschränkt wird. Solche Engpässe könnten durch diese Zusatzkanten umgangen werden. Solch eine Kante wäre mit dem aktuellen Pfadsuch-Algorithmus nicht kompatibel, da dieser ausschließlich die Hilfskante benutzen würde. Ein einfacher Lösungsansatz dafür wäre es, immer den zweitbesten Weg durch die Zeichnung zu nehmen. Das Problem dabei ist jedoch, dass ein Schnitt immer auf dieser Linie startet und sie dann erst nach dem Engpass verlassen kann. Grund dafür ist, dass kein Weg, der oberhalb der Zusatzkanten verläuft, die Hilfskante betreten kann, da ihr größter Schnitt immer größer sein wird. Diese Lösung würde folglich eine sehr ungleichmäßige Zeichnung ergeben, da eine Seite der Zeichnung kompakter wäre als die andere. Der Pfadsuch-Algorithmus müsste also weiter angepasst werden, um eine solche Hilfskante zu unterstützen. Eine Annäherung an eine solche Kante, die mit dem aktuellen Suchalgorithmus umsetzbar wäre, ist es, die Kantengewichte zwischen zwei Rechtecken, die an der unteren Kante der Zeichnung verlaufen nicht zu reduzieren, nachdem ein Schnitt durch diese verlaufen ist. Wir müssen bei diesen Rechtecken nicht befürchten, dass ein unterhalb liegendes Objekt sich irgendwann mit einem anderen überschneidet, da es unterhalb solcher Rechtecke keine Elemente mehr gibt. In Fällen, in denen das Bottleneck nicht dem höchsten Bereich der Zeichnung entspricht, können wir somit den Engpass umgehen. Auch wenn wir dafür eventuell mehr Schnitte brauchen als im Optimalfall mit einer dedizierten Hilfskante.

Nicht-Orthogonale Graphen Wie bereits mehrfach erwähnt ist der in dieser Arbeit vorgestellte Algorithmus speziell für orthogonale Graphen entworfen. Das bedeutet jedoch nicht, dass er nicht auch auf nicht-orthogonale Graphen erweitert werden kann. Eine naheliegende Erweiterung wären Kanten, die in einem 45 Grad Winkel verlaufen. Diese Kanten sind ebenfalls recht weit verbreitet in der Zeichnung von Kabelplänen. Die Zeichenfläche müsste dafür zunächst in komplexere Polygone unterteilt werden. Außerdem müssten Schnitte statt nur in horizontale und vertikale auch in zwei diagonale Richtungen unterschieden werden. Es dürften dabei wieder nur Kanten derselben Ausrichtung gekürzt werden, damit keine zusätzlichen Ecken hinzugefügt werden müssen. Dies könnte sich jedoch als schwer erweisen, falls es nur wenige diagonal verlaufende Kantenabschnitte in der Zeichnung gibt. Außerdem könnte sich die Einhaltung der Mindestabstände als schwierig erweisen. Es bleibt also zu testen, ob eine derartige Erweiterung sinnvoll wäre.

Eindimensionalität Der bisher beschriebene Algorithmus behandelt horizontale und vertikale Stauchung komplett unabhängig voneinander. Dies kann in seltenen Fällen dazu führen, dass der Mindestabstand zwischen zwei Objekten im Graphen nicht eingehalten wird (Siehe Abbildung B.2). Der Grund dafür ist, dass der Zeichenalgorithmus von iPraline im Gegensatz zu anderen Implementierungen nicht alle Elemente auf einem Raster anordnet, dessen Größe dem Mindestabstand entspricht. Dies könnte durch einen Vorverarbeitungsschritt auf der Ausgangszeichnung jedoch geändert werden. Die Zeichnung würde sich dabei womöglich um einen nicht zu vernachlässigenden Anteil vergrößern. Durch die nachfolgende Stauchung sollte sich dies jedoch relativieren.

7. Fazit

Mit dieser Arbeit haben wir die Realisierbarkeit eines Stauchungsalgorithmuses gezeigt, der die Kantenstruktur in iPraline Zeichnungen bewahrt. Dies haben wir erreicht, indem wir einen konzeptionell neuen Algorithmus für orthogonale Graphenzeichnungen auf dem iPraline-Datentyp entwickelt haben. Wir haben dafür eine präzise Definition von Schnitten in Zeichnungen ausgearbeitet. Das entwickelte Verfahren ist in drei modulare Schritte unterteilt. Diese können unabhängig voneinander angepasst werden, womit sich eine Vielzahl an Möglichkeiten ergibt, den Algorithmus zu erweitern und zu verbessern.

Doch auch die bereits vorhandene Implementierung liefert, wie sich in der Auswertung gezeigt hat, bereits zufriedenstellende Ergebnisse. Die Zeichenfläche und die Kantenlänge können maßgeblich verkürzt werden. Der Verlauf der Kanten sowie deren Positionierung an den Knoten bleibt dabei völlig unverändert, was ein großer Vorteil unseres Algorithmus ist. Die größten Nachteile sind die beeinträchtigte Ebenenstruktur und die schlechte Laufzeit. Für beide Probleme haben wir im Abschnitt 6 jedoch mögliche Lösungsansätze genannt. Diese bilden mögliche Ansatzpunkte für die Weiterentwicklung des Algorithmus. Durch einen direkten Vergleich mit anderen Stauchungsalgorithmen könnte man außerdem weitere Stärken und Schwächen des Algorithmus identifizieren und somit eindeutigere Ziele für solche Weiterentwicklungen abstecken.

Literaturverzeichnis

[BETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia und Ioannis G. Tollis: Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994, https://doi.org/10.1016/0925-7721(94)00014-X.

[BMT97] Therese C. Biedl, Brendan P. Madden und Ioannis G. Tollis: The three-phase method: A unified approach to orthogonal graph drawing. In: Giuseppe DiBattista (Herausgeber): *Graph Drawing*, Seiten 391–402. Springer Berlin Heidelberg, 1997, https://doi.org/10.1007/3-540-63938-1$_8$4.

[HS93] D. A. Holton und J. Sheehan: *The Petersen Graph*. Australian Mathematical Society Lecture Series. Cambridge University Press, 1993, 10.1017/CBO9780511662058.

[Kö23] Sebastian Körner: Stauchungsalgorithmus und Graphen-Dataset. `https://gitlab.informatik.uni-wuerzburg.de/s411537/bachelorarbeit-koerner`, 2023.

[Len90] Thomas Lengauer: Combinatorial Algorithms for Integrated Circuit Layout. Jan 1990, 10.1007/978-3-322-92106-2.

[ZWBW20a] Johannes Zink, Julian Walter, Joachim Baumeister und Alexander Wolff: iPraline Data Structure and Layouting Algorithm. `https://github.com/j-zink-wuerzburg/praline`, 2020.

[ZWBW20b] Johannes Zink, Julian Walter, Joachim Baumeister und Alexander Wolff: iPraline Pseudo Plans Algorithm and Data Sets. `https://github.com/j-zink-wuerzburg/pseudo-praline-plan-generation`, 2020.

[ZWBW22] Johannes Zink, Julian Walter, Joachim Baumeister und Alexander Wolff: Layered drawing of undirected graphs with generalized port constraints. *Computational Geometry*, 105-106:101886, 2022, 10.1016/j.comgeo.2022.101886.

Anhang

A. SweepLine

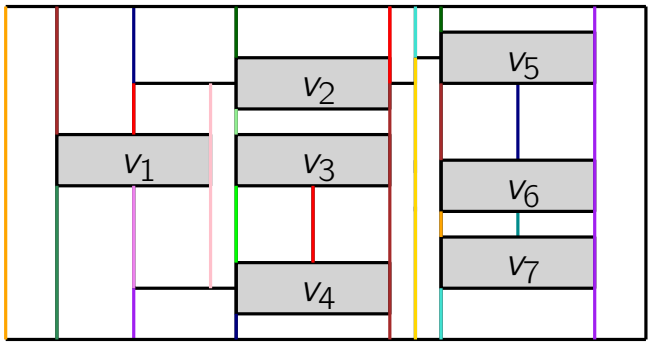


Abb. A.1.: Ein Beispielgraph, in dem die vertikalen Linien farblich unterschieden sind. Schwarze Linien sind die linke Seite eines Knotens und werden daher in der Schnittsuche ignoriert.

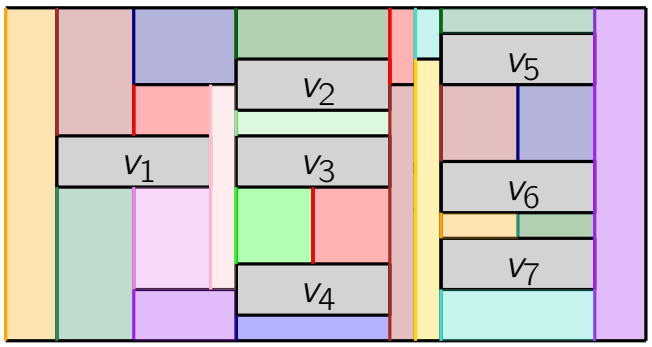
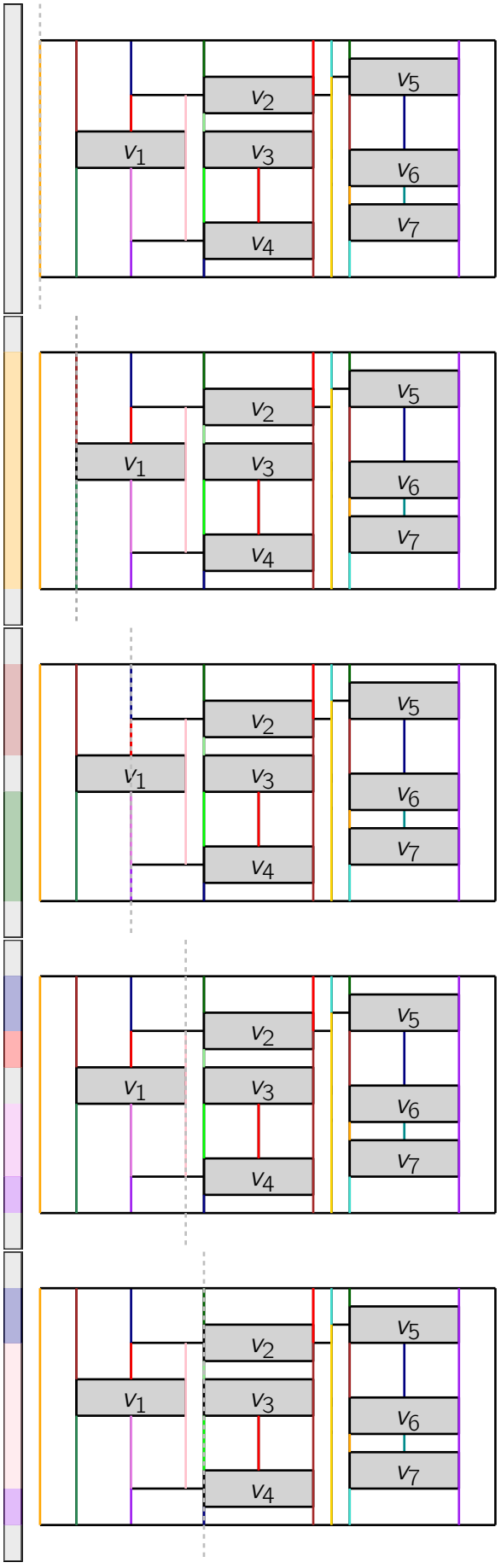
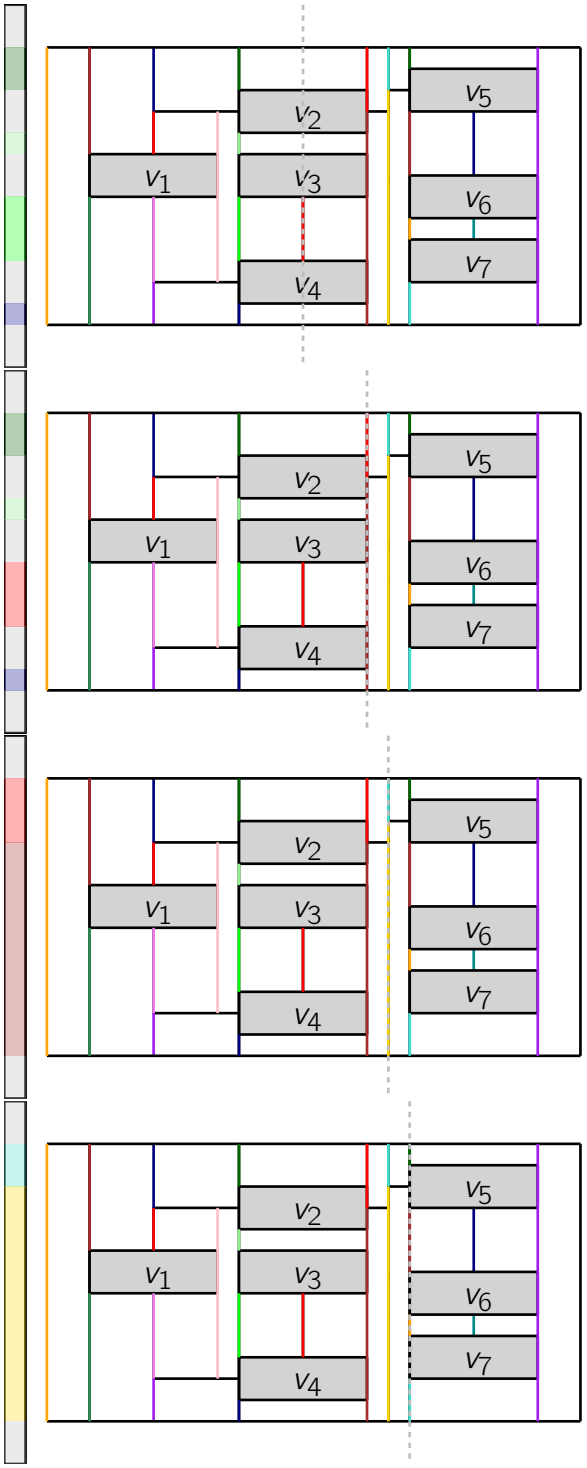


Abb. A.2.: Die zugehörige Rechtecksfläche zu jeder vertikalen Linie in gleicher Farbe markiert.


v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_1
v_2
v_3
v_4
v_5
v_6
v_7


v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_1
v_2
v_3
v_4
v_5
v_6
v_7

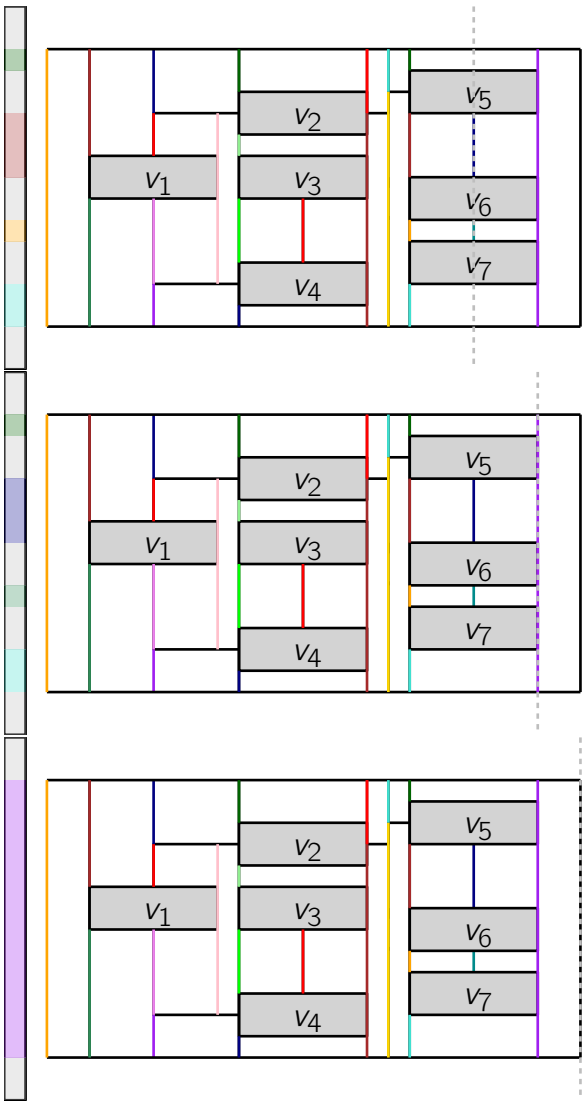
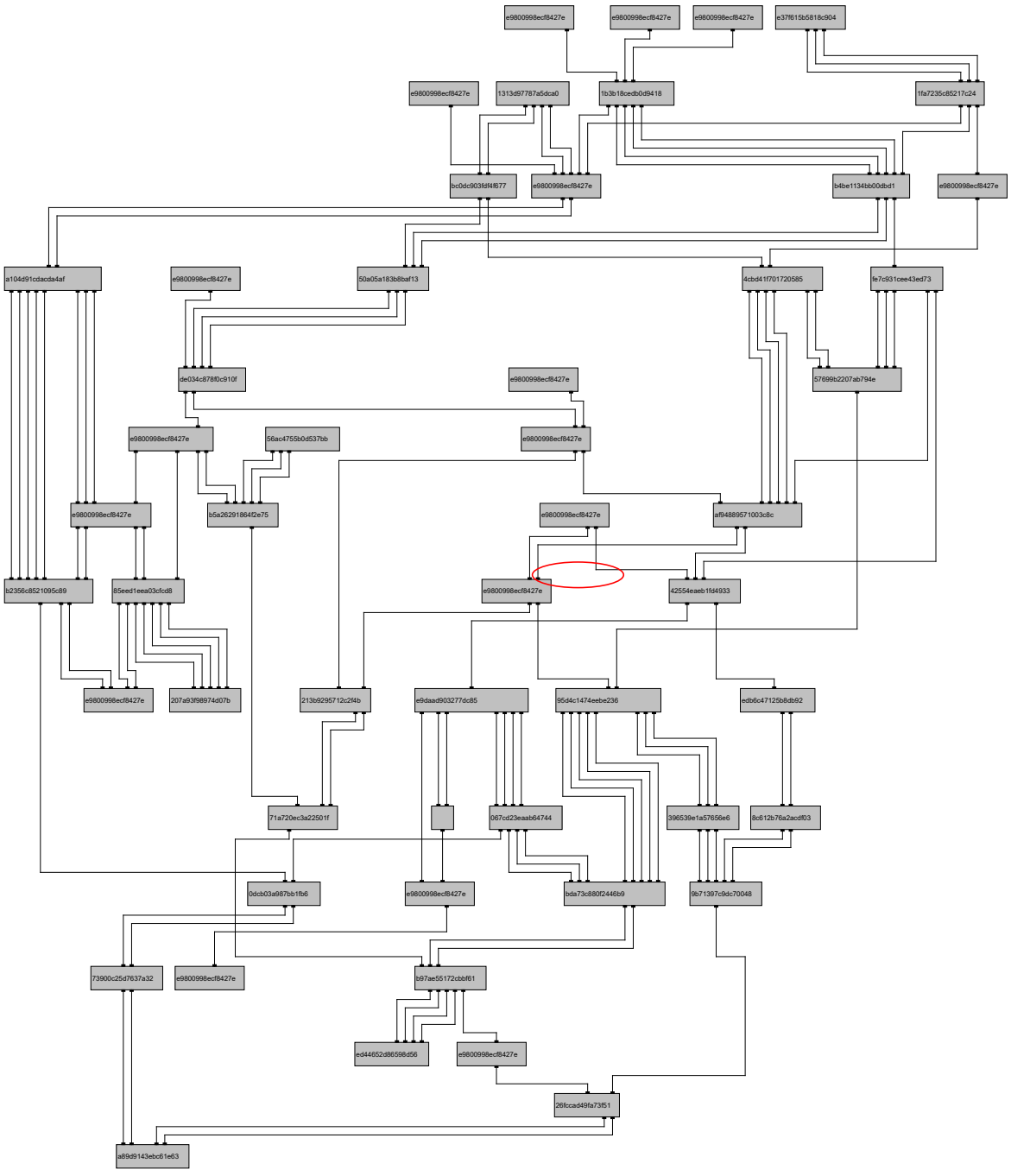
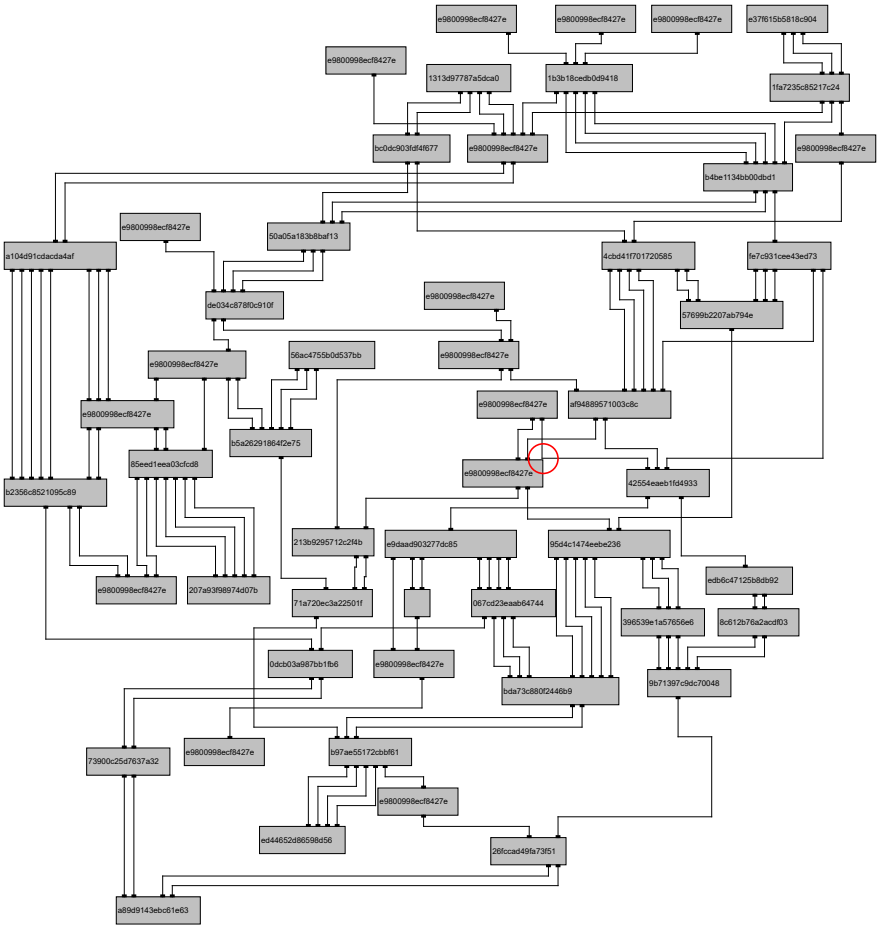


Abb. A.3.: Verlauf der ScanLine durch den Graphen. Die ScanLine wird durch die gestrichelte, graue Linie dargestellt. Der linke Balken zeigt die „Farben“ der ScanLine an. Ungefärbte Bereiche sind grau. Trifft ein gefärbter Bereich der Linie auf eine neue vertikale Linie, so wird eine Kante im Hilfsgraphen zwischen den beiden Rechtecken eingezeichnet und die ScanLine anschließend umgefärbt.

B. Zusatzgraphen

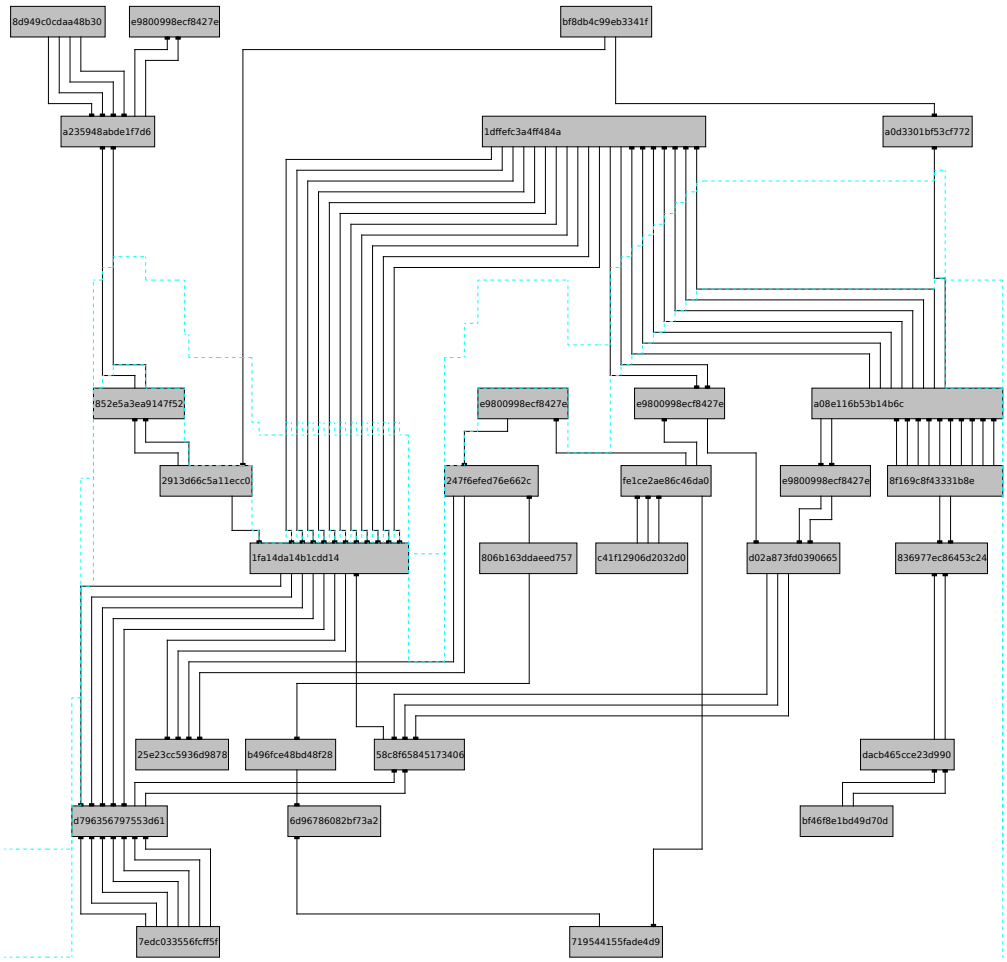


(a) iPraline Layout

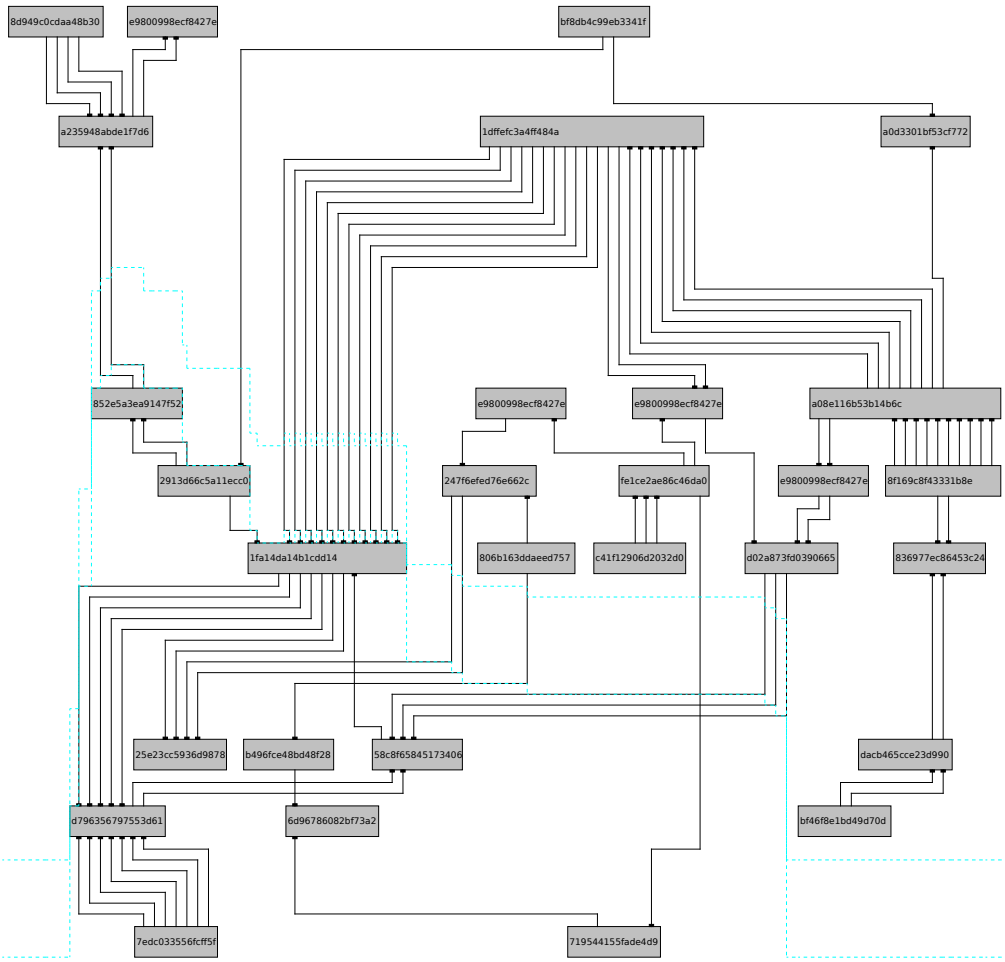


(b) Gestauchtes Layout

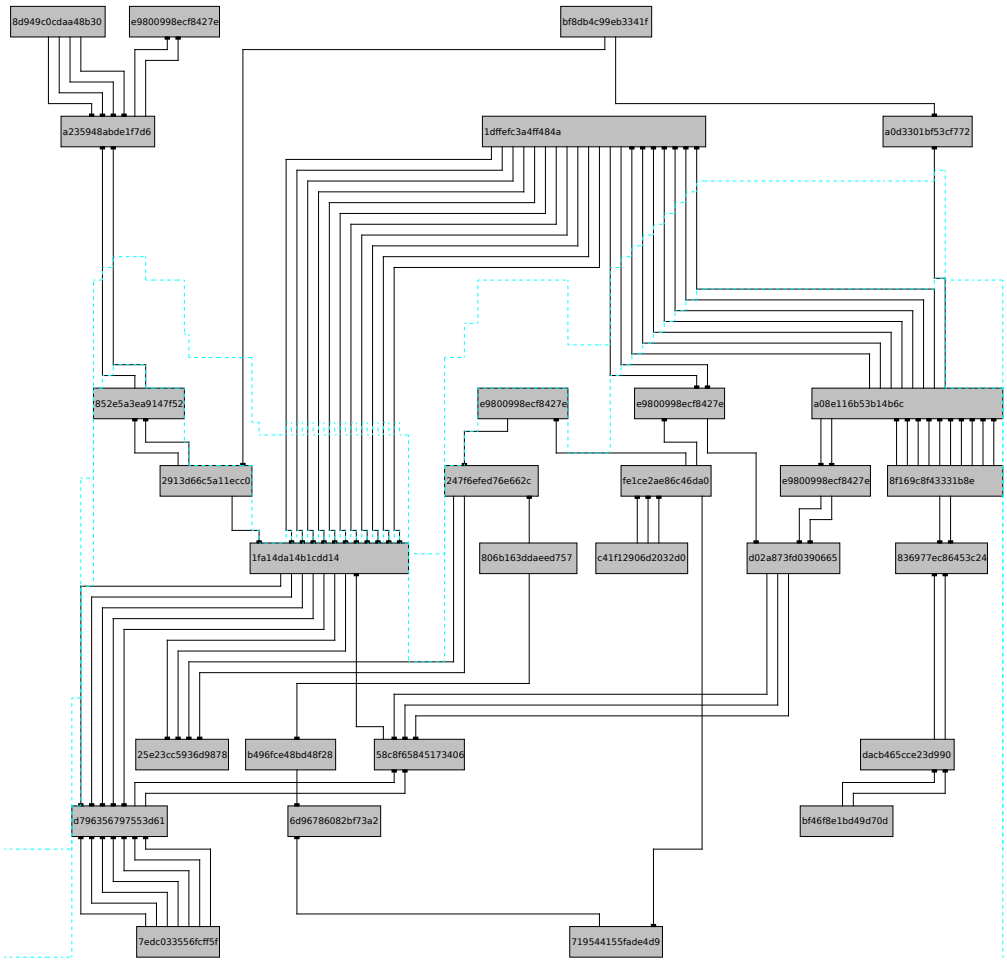
Abb. B.2.: Ein von iPraline gezeichneter Graph aus dem Datenset mit 53 Knoten und 132 Kanten. In den rot eingezeichneten Bereichen sieht man, dass der Mindestabstand aufgrund der eindimensionalen unseres Algorithmus nicht eingehalten wird.



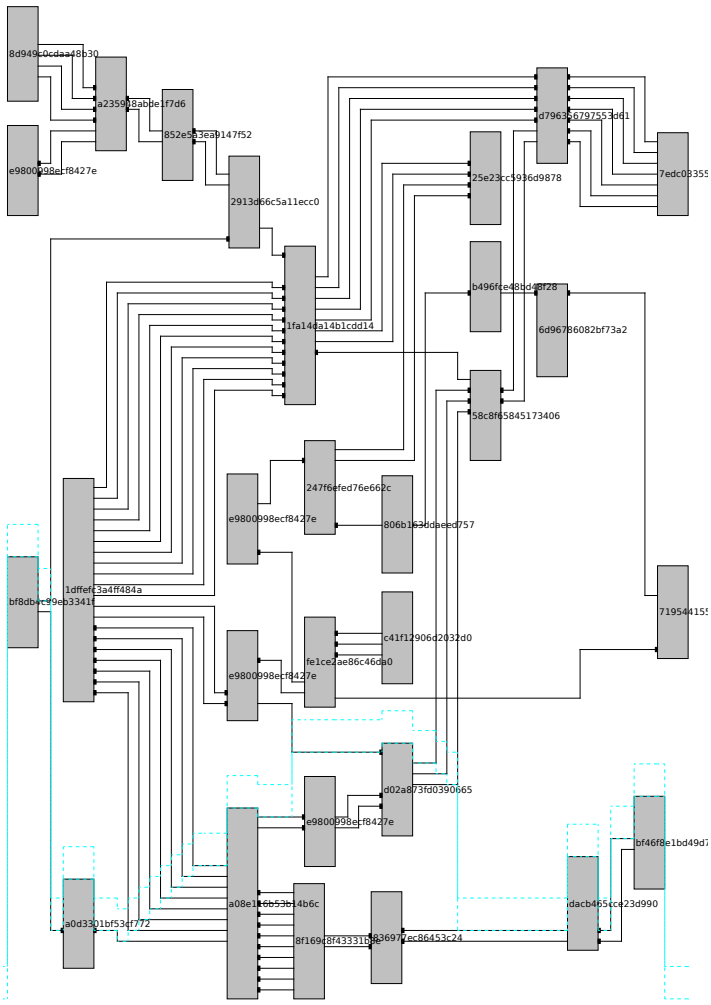
(a) Erster horizontaler Schnitt



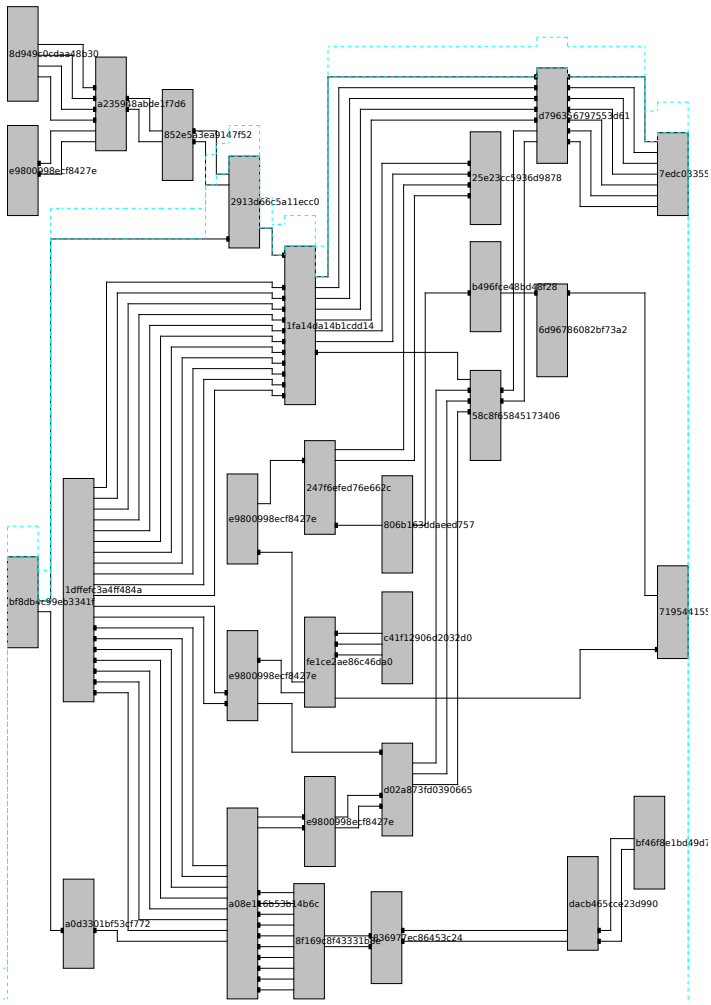
(b) Zweiter horizontaler Schnitt



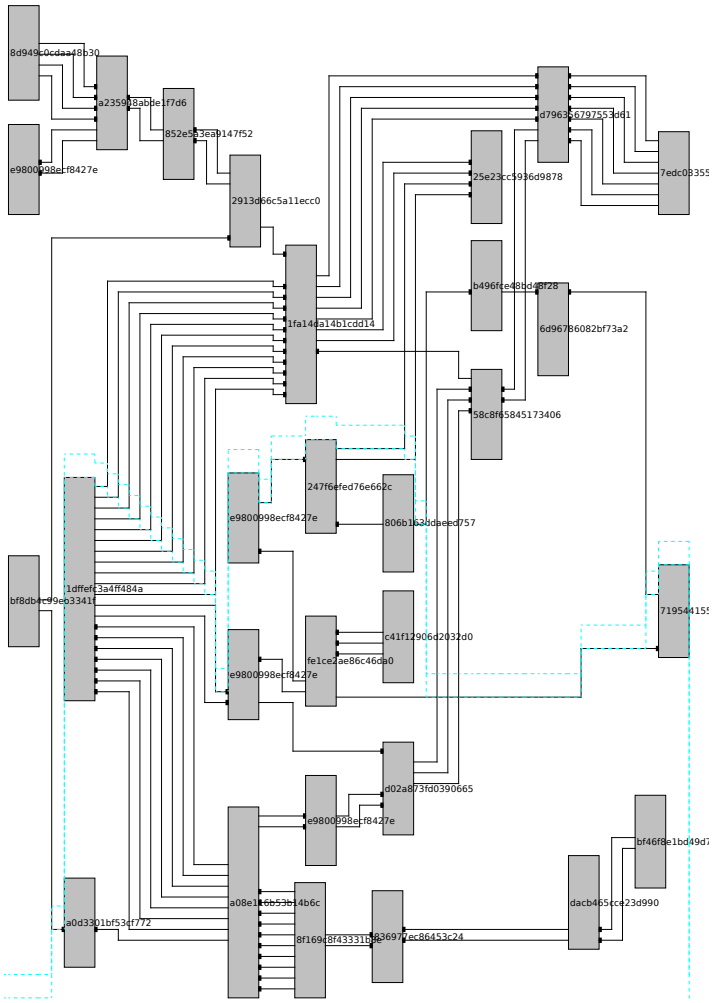
(c) Dritter horizontaler Schnitt



(d) Erster vertikaler Schnitt



(e) Zweiter vertikaler Schnitt



(f) Dritter vertikaler Schnitt

Abb. B.8.: Eine Visualisierung der ersten drei Schnitte in horizontaler und vertikaler Richtung für den Beispielgraphen aus Abschnitt 5.5. Die untere türkisfarbene Linie entspricht der Schnittlinie. Die obere türkisfarbene Linie ist die Position der Linie nach der Stauchung mithilfe dieses Schnittes. Der Abstand der Beiden ist die Größe des Schnittes und der Abschnitt dazwischen der weggeschnittene Bereich.

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Versicherung zur selbständigen Leistungserbringung

Titel der Bachelorarbeit :

Stauchung orthogonaler Graphenzeichnungen mithilfe komplexer Schnitte

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):

Prof. Dr., Alexander, Wolff, Lehrstuhl I für Informatik

Eingereicht durch (Vorname, Nachname, Matrikel):

Sebastian, Körner, 2597126

Ich versichere, dass ich die vorstehende schriftliche Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

☒ Mit dem Prüfungsleiter bzw. der Prüfungsleiterin wurde abgestimmt, dass für die Erstellung der vorgelegten schriftlichen Arbeit Chatbots (insbesondere ChatGPT) bzw. allgemein solche Programme, die anstelle meiner Person die Aufgabenstellung der Prüfung bzw. Teile derselben bearbeiten könnten, eingesetzt wurden. Die mittels Chatbots erstellten Passagen sind als solche gekennzeichnet.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Leinach, 24.07.2023, Sebastian Körner

Ort, Datum, Unterschrift

Ref. 2.3 / Stand: 15.06.2023