Karlsruher Institut für Technologie

Die Auswirkungen neuer Kanten auf den CCH-Algorithmus

Bachelor Arbeit von

Benedikt Müller

an der Fakultät für Informatik
Institut für Theoretische Informatik (ITI)

Erstgutachter:	T.T.-Prof. Dr. Thomas Bläsius
Zweitgutachter:	Prof. Dr. Bernhard Beckert
Betreuer:	Michael Zündorf
	Adrian Feilhauer

13.12.2024 – 17.03.2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Quellen und Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, **17.03.2025**

.....

(Benedikt Müller)

Zusammenfassung

Damit Routingalgorithmen schnell viele Anfragen, auch Queries genannt, in kurzer Zeit beantworten können, werden häufig Vorberechnungen verwendet. Da sich die Topologie des Graphen nur selten verändert, sind auch aufwendigere Vorberechnungen akzeptabel. Der Customizable Contraction Hierarchies (CCH) Algorithmus berechnet dazu in einem ersten, teuren Schritt eine Ordnung, die in den weiteren, günstigeren Schritten verwendet wird. Fügt man eine neue Kante in den Graphen ein, ändert sich seine Topologie und es müssen alle Vorberechnungen erneut durchgeführt werden. In dieser Arbeit beschäftigen wir uns daher mit der Frage, wie der CCH-Algorithmus besser auf lokale Änderungen in der Topologie reagieren kann. Unser Ziel ist es daher, die Ordnung effizient und lokal an die neue Kante anzupassen. Dafür stellen wir drei Lösungsansätze vor. Die erste Möglichkeit ist, die alte Ordnung zu übernehmen, dies optimiert die Vorberechnungen. Alternativ kann lokal eine optimale Ordnung berechnet werden, was zu schnellen Queryzeiten führt. Als Kompromiss können wir nur die Ordnung von einem Endpunkt der Kante erhöhen, dadurch erhalten wir sowohl in der Vorberechnung, als auch in den Queries schnelle Ergebnisse. Im abschließenden experimentellen Vergleich der Varianten stellt sich heraus, dass der CCH-Algorithmus durch die zuletzt genannte Variante effizient auf neue Kanten reagieren kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Kürzeste Wege in einem Straßennetz zu finden, ist ein alltägliches Problem und es gibt diverse Anwendungen, die es lösen [DGJ]. Um die Nutzerfreundlichkeit zu gewährleisten, müssen die verwendeten Routingalgorithmen in der Lage sein, auf viele Anfragen innerhalb kürzester Zeit zu reagieren und verwenden dazu oft Vorberechnungen. Ein Beispiel für einen Routingalgorithmus mit einer Vorberechnungsphase ist der Contraction Hierarchies Algorithmus (CH) [GSSV12]. Damit sich der Aufwand für die Vorberechnungen rentiert, darf sich der zugrundeliegende Straßengraph nur selten ändern, da die Vorberechnungen andernfalls oft wiederholt werden müssen. In der Praxis beobachtet man aber, dass sich vor allem die Reisezeit zwischen verschiedenen Orten, und damit der zugrundeliegende Graph, mehrmals täglich ändert. Das führt beim CH-Algorithmus dazu, dass seine Vorberechnungen häufig neu berechnet werden müssen, worunter seine Effizienz leidet [DSW16]. Dieses Problem wird durch den Customizable Contraction Hierarchies Algorithmus (CCH) gelöst, indem die Vorberechnungen in mehrere Schritte aufgeteilt werden. Ändert sich die Topologie des Graphen, müssen alle Vorberechnungen wiederholt werden, bei Änderungen an den Kantengewichten dagegen nur die zweite Phase [DSW16].

Wird nun eine neue Straße gebaut, ändert sich die Topologie des Straßennetzes. Also muss der CCH-Algorithmus alle Vorberechnungen erneuern und kann somit nur langsam auf neue Straßen reagieren. Dabei können wir eine neue Straße formal durch das Einfügen einer neuen Kante in den zugehörigen Graphen modellieren. In der praktischen Routenfindung im Straßenverkehr können wir allerdings beobachten, dass sehr viele Straßen nur einen sehr lokalen Einfluss haben [Blä+25]. Eine Straße in einem Wohngebiet ist in der Regel auch nur für Strecken in diesem Wohngebiet relevant. Für Routen in einer anderen Stadt wird diese Straße dagegen meistens nicht benötigt. Selbst eine Autobahn ist für Reisen, die sehr weit von dieser Region entfernt sind, oftmals unbedeutend [SS15]. Eine neue Straße beeinflusst also das Straßennetz an weit entfernten Orten nur geringfügig. Dennoch kann sie in einem Routingalgorithmus zentrale Änderungen bewirken.

Aus diesem Grund beschäftigen wir uns in dieser Arbeit mit der Frage, welche Auswirkungen eine neue Kante auf den CCH Algorithmus hat. Dafür betrachten wir zunächst den Algorithmus genauer und beschreiben seinen Ablauf. Anschließend stellen wir verschiedene Optimierungen vor, durch die der Algorithmus schneller auf eine neue Kante reagieren kann. Diese Ansätze evaluieren wir schließlich und vergleichen sie experimentell miteinander.

2 Allgemeine Definitionen und Notationen

Im Folgenden führen wir wichtige Notationen und Begriffe ein, die wir in dieser Arbeit verwenden.

Sei V eine Knotenmenge und $E_u \subseteq \big\{\{u, v\} \mid u, v \in V\big\}$ eine Kantenmenge, dann nennen wir $G_u = (V, E_u)$ einen ungerichteten Graphen. In einem gerichteten Graph $G_g = (V, E_g)$ besteht die Kantenmenge $E_g \subseteq \big\{(u, v) \mid u, v \in V\big\} = V \times V$ aus Tupeln anstatt Mengen. Wenn eine Gewichtsfunktion $c : E \to \mathbb{R}$ existiert, die den Kanten Gewichte zuordnet, nennen wir den Graphen zusätzlich gewichtet. Sowohl ein gerichteter, als auch ungerichteter Graph kann gewichtet sein.

In einem ungerichteten Graphen sind zwei Knoten $u, v \in V$ benachbart, falls die Kante $\{u, v\} \in E$ existiert. Alle Knoten, mit denen $v \in V$ in G benachbart ist, werden zu seiner Nachbarschaft $N_G(v)$ zusammengefasst. Wenn in einer Knotenteilmenge $C \subseteq V$ alle Knoten paarweise benachbart sind, ist C eine Clique. Eine nichtleere Teilmenge $X \subsetneq V$ definiert einen Schnitt $(X, V \setminus X)$ auf dem Graphen. Ein Schnitt teilt einen Graphen in alle Knoten, die in X enthalten sind, und alle übrigen Knoten.

In einem gerichteten Graph heißt eine Folge von Knoten $P = (v_1, \ldots, v_n) \subseteq V$ Pfad, falls Kanten zwischen jeweils aufeinanderfolgende Knoten v_i und v_{i+1} existieren, also $(v_i, v_{i+1}) \in E$ gilt. Für zwei Knoten $s, t \in V$ nennen wir den Pfad von s nach t einen s–t–Pfad. Die Länge eines Pfades ist definiert, als die Anzahl seiner Knoten. Ein Zyklus ist ein s–t–Pfad, der Länge mindestens zwei, bei dem $s = t$ gilt. Ein gerichteter, kreisfreier Graph (*DAG*) ist ein gerichteter Graph, in dem keine Zyklen existieren. Drei Knoten $u, v, w \in V$ bilden ein Dreieck, falls die Kante $(u, w) \in E$, sowie der Pfad (u, v, w) im Graphen existieren. Falls zusätzlich eine Totalordnung π auf den Knoten existiert, nennen wir das Dreieck $u, v, w \in V$ mit $\pi(u) < \pi(v) < \pi(w)$ ein unteres Dreieck von $(v, w) \in E$, weil die Ordnung von u kleiner ist als die von v und w. Analog nennen wir in diesem Fall das Dreieck ein mittleres Dreieck von (u, w) und ein oberes Dreieck von (u, v).

Ein Straßennetz kann als gerichteter Graph modelliert werden. Jeder Straßenabschnitt wird dabei als gerichtete Kante dargestellt und die Endpunkte der Segmente als die Knoten. Eine Straße, die in beide Richtungen befahrbar ist, wird als zwei Einbahnstraßen in entgegengesetzten Richtungen modelliert, es existiert dann also eine Kante und ihre Gegenkante.

Ein zusammenhängender DAG $T = (V, E)$ mit einem Wurzelknoten $w \in V$ heißt Baum, falls w Endpunkt keiner Kante ist, und alle übrigen Knoten $v \in V \setminus \{w\}$ Endpunkt genau einer Kante in E sind. Für einen Baum wird seine Tiefe definiert als die maximale Pfadlänge ab Wurzel w. Knoten, die keine ausgehenden Kanten haben, werden Blätter genannt.

3 CCH-Algorithmus

Der Customizable Contraction Hierarchies (CCH) Algorithmus ist eine Technik, kürzeste Wege in einem Straßengraphen effizient für viele Anfragen in kurzer Zeit zu berechnen [DSW16]. Dazu bekommt der Algorithmus als Eingabe einen gerichteten und gewichteten Graphen $G = (V, E, c)$. Bei $c : E \to \mathbb{R}_{\geq 0}$ handelt es sich um eine Kostenfunktion auf den Kanten. Für eine gegebene Metrik auf den Knoten sollen die Kosten einer Kante $(u, v) \in E$ dem Abstand zwischen den Knoten u und v in der Metrik entsprechen. In dieser Arbeit verwenden wir als Metrik die Reisezeit zwischen Knoten, es sind aber auch andere Metriken wie z. B. der euklidische Abstand möglich.

In der Praxis beobachtet man, dass sich die Topologie von Straßennetzen nur selten ändert. Allerdings variiert die Reisezeit zwischen zwei Orten zu verschiedenen Tageszeiten zum Teil erheblich. Aus diesem Grund besteht der CCH-Algorithmus aus insgesamt drei Phasen: Die erste Phase ist Metrik-unabhängig, in ihr werden Vorberechnungen auf der Topologie des Graphen ausgeführt. Nachdem sich die Topologie nur selten ändert, wird diese Phase nur selten durchlaufen und hat daher geringen Einfluss auf die durchschnittliche Laufzeit. In einer parallelen Implementierung dieser Phase liegen übliche Laufzeiten auf großen Straßengraphen im Bereich von mehreren Minuten, bei einer sequentiellen Implementierung wird dagegen deutlich mehr Zeit benötigt [DSW16 | Blä+25]. In dieser Arbeit verwenden wir DIMACS Straßengraphen von Europa, mit ca. 18 000 000 Knoten, er dient uns als Beispiel einer großen Instanz [DGJ]. Die zweite Phase, die *Customization*, ist eine Metrik-abhängige Vorberechnungsphase und soll schnell auf Änderungen in der Metrik reagieren können. Da die Reisezeit, welche wir als Metrik nutzen, sich häufig ändert, wird diese Phase oft ausgeführt. Aus diesem Grund ist eine kurze Laufzeit in dieser Phase wichtiger als in der ersten Phase. Selbst in sequentiellen Implementierungen liegen hier übliche Laufzeiten auf großen Instanzen im Bereich einiger Sekunden [DSW16 | Blä+25]. Abschließend werden in der Query-Phase die Kosten von s–t–Pfaden berechnet. Auch sequentiell sollte ein Query in unter einer Millisekunde berechnet werden können [DSW16 | Blä+25].

3.1 Struktur des Algorithmus

In diesem Abschnitt beschreiben wir grob die Struktur des CCH-Algorithmus in Anlehnung an Bläsius et al. [Blä+25]. In den folgenden Abschnitten dieses Kapitels erläutern wir dann weitere Details zu den einzelnen Phasen des Algorithmus.

Ziel der Vorberechnungen ist es, Kanten so einzufügen, dass es für jeden s–t–Pfad ausreicht, nur Knoten mit höherer Ordnung als s oder t zu betrachten. Dadurch wird die Anzahl an Knoten, welche wir in der Query-Phase besuchen müssen, stark reduziert. Die in den Vorberechnungen neu eingefügten Kanten nennen wir *Shortcuts* und den resultierenden Graphen G' *Augmented Graph*. Die Queries bearbeiten wir dann auf dem neu berechneten Augmented Graph G'.

Die Metrik-unabhängige Phase zu Beginn lässt sich in zwei Aufgaben unterteilen: Zunächst berechnen wir eine Totalordnung $\pi : V \to \{0, \ldots, |V| - 1\}$ der Knoten. Diese heißt *Nested Dissection Ordnung* und wird von uns im Folgenden kurz als Ordnung bezeichnet [SS15]. In

dieser Reihenfolge werden die Knoten in den weiteren Schritten des CCH-Algorithmus abgearbeitet. In der *Contraction* wird anschließend für jeden Knoten $v \in V$ seine Nachbarschaft mit höherer Ordnung als v zu einer Clique vervollständigt. Danach werden in der Customization neue Kantengewichte für den Augmented Graph berechnet.

Eine Kante $(u, v) \in E$ heißt aufwärts gerichtet, falls $\pi(u) < \pi(v)$ gilt, andernfalls ist sie nach unten gerichtet. Ausgehend von G' definieren wir dann die Auf- und Abwärtsgraphen $G^{\uparrow}$ bzw. $G^{\downarrow}$, als die DAGs, in denen alle Kanten aus G', die aufwärts bzw. abwärts gerichtet sind, übernommen werden. Um die Kosten eines s–t–Pfades zu berechnen, ist es dann ausreichend, eine Aufwärtssuche in $G^{\uparrow}$ von s und eine in $G^{\downarrow}$ von t ausgehend durchzuführen. Um die Queries effizient berechnen zu können, benötigen wir einen *Elimination Tree*. Dieser speichert für jeden Knoten $v \in V$ seinen *Parent*. Der Parent von v wird dabei definiert als der eindeutige Knoten p mit der niedrigsten Ordnung unter allen Nachbarn von v in G' mit höherer Ordnung als v. Wir schreiben dies als $\text{parent}(v) = p$ und es gilt $\text{parent}(v) = \text{argmin}\{\pi(w) \mid w \in N_{G'}(v), \pi(w) > \pi(v)\}$.

In Abschnitt 3.5 stellen wir fest, dass die Ordnung einen großen Einfluss auf die Struktur des Elimination Tree und die Performance des gesamten Algorithmus hat. Damit wir sowohl die Vorberechnungen als auch die Queries effizient berechnen können, müssen wir eine Ordnung finden, die zu einem balancierten Elimination Tree führt.

3.2 Metrik-unabhängige Vorberechnungen

Wie oben erwähnt besteht die Metrik-unabhängige Vorberechnungsphase aus zwei Teilen: Zunächst berechnen wir eine Totalordnung π der Knoten, danach fügen wir in der Contraction die Shortcuts ein. Dabei interessieren wir uns in der gesamten Phase nur für die Topologie des Graphen. Daher sind hier die Richtungen und Gewichte der Kanten nicht relevant [DSW16 | Blä+25].

Die Ordnung hat einen großen Einfluss auf die Performance der folgenden Schritte. Bei einer schlechten Ordnung kann der Elimination Tree zu tief werden, wodurch die Query-Berechnung ineffizient wird. Außerdem können die Vorberechnungen deutlich länger dauern, da zu viele Shortcuts eingefügt werden. Des Weiteren kann es durch die vielen Kanten auch zu Speicherprobleme kommen [Blä+25]. In Abschnitt 3.2.1 wird eine Variante beschrieben, um eine gute Ordnung zu berechnen.

Auffällig ist, dass das Berechnen der Ordnung deutlich teurer ist, als die Contraction. Eine gute Ordnung zu berechnen dauert mehrere Minuten, die Contraction ist im Gegensatz dazu nach mehreren Sekunden abgeschlossen [Blä+25].

3.2.1 Inertial Flow

Der Inertial-Flow-Algorithmus ist eine Technik, um gute Ordnungen effizient zu berechnen [SS15]. Eine Beobachtung aus der Praxis ist, dass nicht jede Straße gleich wichtig ist: Eine Autobahn oder große Brücke ist z. B. deutlich wichtiger als ein Feldweg. Wenn eine Route beispielsweise einen Fluss quert, so muss sie eine der Brücken über den Fluss nutzen. Die Fragen, wie man zu der Brücke kommt, und wie man danach weiterfährt, sind unabhängig. Daraus können wir ableiten, dass manche Knoten in einem Straßengraphen wichtiger sind als andere. Kanten, die eine Route partitionieren, nennen wir wichtige Kanten und deren Endpunkte sollen hohe Ordnungen bekommen. Man kann nun so lange wichtige Knoten löschen, bis der Graph in mehrere Zusammenhangskomponenten zerfällt. Die gelöschten Knoten bilden einen Separator $S \subsetneq V$ und alle Knoten aus S erhalten die höchsten Ordnungen.

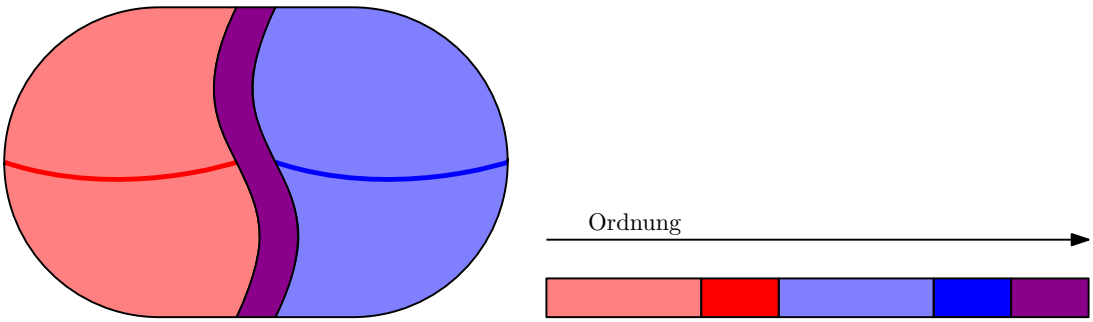

Abbildung 3.1: Schematisches Beispiel für die Berechnung einer Nested Dissection Ordnung bis zum zweiten Rekursionsschritt. Links schematisch ein Graph mit einem Separator auf höchster Ebene in violett, der den Graphen in einen roten und einen blauen Teil spaltet. Die Separatoren in den jeweiligen Teilen sind wieder in kräftigem rot bzw. blau gefärbt. Rechts wird gezeigt, wie die einzelnen Teile in die Ordnung zusammengefügt werden. Der rote Teil hat die niedrigsten Ordnungen, anschließend der rote Separator die nächst höheren, anschließend der gesamte blaue Teil gefolgt von seinem Separator. Der violette Separator auf höchster Ebene erhält die höchsten Ordnungen.

In jeder der neu entstandenen Zusammenhangskomponenten $V_1, \ldots, V_k \subsetneq V \setminus S$ kann dann rekursive eine Ordnung für die einzelnen Komponenten berechnet werden. Da die Komponenten unabhängig voneinander sind, ist es für den Algorithmus irrelevant, auf welche Arten die Teilordnungen zusammengefügt werden [SS15]. Zur besseren Übersicht werden wir in dieser Arbeit die einzelnen Teile strikt hintereinander einfügen. Die erste Komponente erhält also die niedrigsten Ordnungen, die zweite die nächst höheren und so weiter, dies wird schematisch auch in Abbildung 3.1 gezeigt.

Es gilt also, kleine, balancierte Separatoren auf dem Graphen zu finden. Dieses Problem ist im Allgemeinen allerdings NP-vollständig [GJS76]. Straßennetze haben jedoch natürliche Barrieren wie Flüsse oder Gebirge. Solche Grenzen bilden kleine, balancierte Separatoren, sodass die Berechnung auf realistischen Straßengraphen effizienter ist [SS15].

Zur Berechnung eines solchen Separators wird ein kleiner Schnitt $C \subsetneq V$ auf dem Graphen berechnet. Daraus kann anschließend die Schnittkantenmenge $X \subsetneq E$ berechnet werden, die genau die Kanten $(u, v) \in E$ enthält, bei denen $u \in C$ und $v \notin C$ gilt oder umgekehrt. Aus X können schließlich zwei Separatoren $S_1, S_2 \subsetneq V$ berechnet werden, indem für jede Kante aus X der Endpunkt, welcher in C liegt, in S_1 eingefügt wird und der andere in S_2 [SS15].

Um den Schnitt C zu erhalten, wird ein Flussnetzwerk gelöst. Dafür wird der Graph auf eine Gerade projiziert und ein maximaler Fluss entlang der Geraden gesucht. Dabei handelt es sich bei der Quelle um den ersten und bei der Senke um den letzten Knoten in der Projektion. Zwischen diesen Knoten wird dann ein maximaler Fluss mittels Dinics Algorithmus [Din70] oder eines anderen Max-Flow-Algorithmus berechnet. Kanten aus dem Flussnetzwerk, die keinen weiteren Fluss transportieren können, also saturiert sind, entsprechen dann den wichtigsten Kanten im Straßengraphen. Der Schnitt C enthält schließlich alle Knoten, die von der Quelle aus über nicht saturierte Kanten erreichbar sind. Nach dem Max-Flow-Min-Cut-Theorem handelt es sich bei C um einen minimalen S–T-Schnitt, also ein Schnitt, bei dem sich die Quelle in C und die Senke in $V \setminus C$ befindet [EFS56].

Das oben beschriebene wird für verschiedene Geraden durchgeführt, z. B. für die Nord-Süd- und Ost-West-Achsen, und aus allen Optionen wird der kleinste Separator gewählt. Auf jeder Komponente, in die der Graph durch Herausschneiden des Separators zerfällt, wird rekursiv

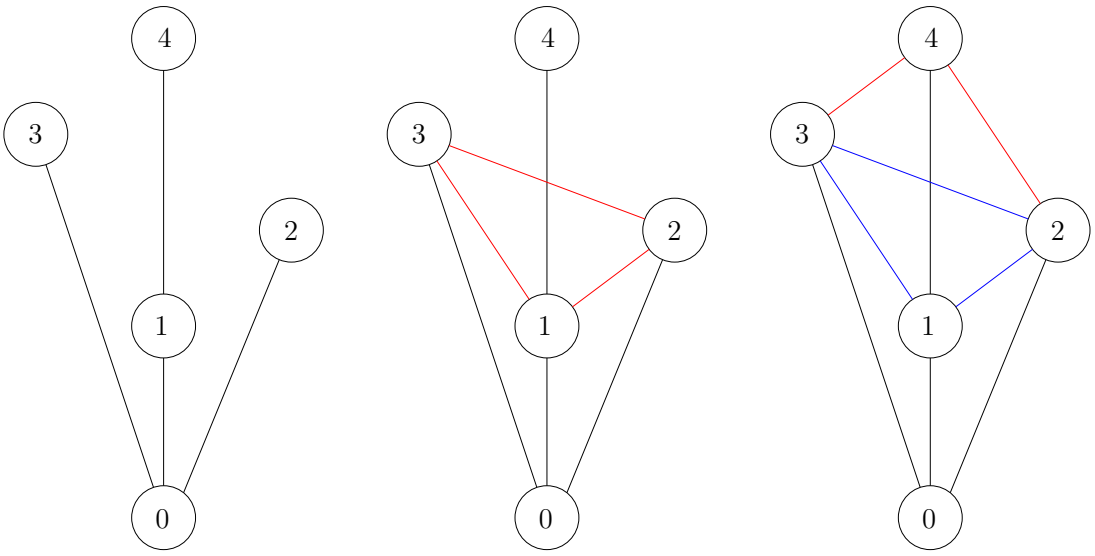

Abbildung 3.2: Beispiel einer Contraction, der Knotenname entspricht auch seiner Ordnung ($\pi = id$). Links der Originalgraph, in der Mitte wurde der Knoten „1“ fertig kontrahiert und rechts der Graph nach der Abarbeitung von Knoten „2“. Letzterer ist gleichzeitig auch der fertig kontrahierte Graph. Im aktuellen Schritt neu eingefügte Shortcuts werden rot dargestellt, alte Shortcuts blau.

wieder ein solcher Separator berechnet. Schlussendlich werden die Ordnungen der Teilgraphen zusammengefügt. Dies geht z. B. indem die erste Komponente die niedrigsten Ordnungen bekommt, anschließend erhält die nächste Komponente die nächst höheren Ordnungen und so weiter, bis alle Teilordnungen zusammengefügt wurden. Als Letztes werden den Separatorknoten die höchsten Ordnungen zugewiesen. Damit der oben berechnete maximale Fluss nicht durch die geringen Knotengrade des ersten und letzten Knotens beschränkt ist, fasst der Algorithmus die ersten und letzten 30 % der Knoten zur Quelle bzw. Senke zusammen. Auf diesem Graphen wird dann der maximale Fluss zwischen Quelle und Senke berechnet.

3.2.2 Contraction

In der Contraction fügen wir die Shortcuts in den Graphen ein und berechnen somit die Topologie des Augmented Graphs G' [DSW16]. Am Ende soll für jeden s–t–Pfad $P = (v_1, \dots, v_k)$ in G' mit $s = v_1$ und $t = v_k$ ein $i \in \{1, \dots, k\}$ existieren, sodass $\pi(v_1) < \cdots < \pi(v_i)$ sowie $\pi(v_i) > \cdots > \pi(v_k)$ gilt.

Wie in Abbildung 3.2 werden dazu die Knoten in aufsteigender Reihenfolge gemäß der Ordnung durchlaufen. Für jeden Knoten $v \in V$ wird seine Nachbarschaft mit höherer Ordnung als v zu einer Clique vervollständigt. Dabei können Shortcuts auch zu weiteren Shortcuts führen. Es macht also einen Unterschied, in welcher Reihenfolge die Knoten durchlaufen werden. In Abbildung 3.2 wird z. B. der Shortcut $(2, 4)$ nur wegen des Shortcuts $(1, 2)$ eingefügt. Wir bezeichnen mit S die Menge aller Shortcuts und definieren anschließend die neue Kantenmenge $E' = E \cup S$ für den Augmented Graph G'. Insgesamt kann die Contraction in $\mathcal{O}(|V| + |E'|)$ berechnet werden [DSW16].

Der in dieser Arbeit implementierte Algorithmus verwendet verschiedene, optimierte Implementierungsdetails. Diese wurden von Bläsius et al. [Blä+25] zusammengefasst.

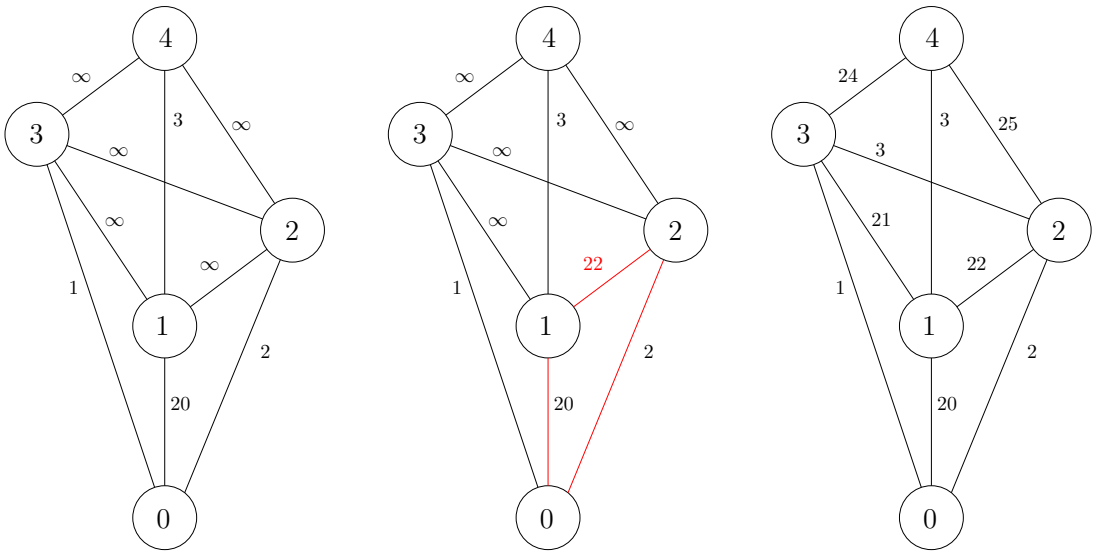

Abbildung 3.3: Beispiel einer Customization, der Knotenname entspricht auch seiner Ordnung ($\pi = id$). Links ist der kontrahierte Originalgraph. In der Mitte wird das Dreieck 0, 1, 2 bearbeitet, das aktuelle Dreieck sowie das sich ändernde Kantengewicht ist rot eingefärbt. Rechts der fertige Augmented Graph.

3.3 Customization

In der Customization berechnen wir nun die neuen Kantengewichte $c' : E' \to \mathbb{R}_{\geq 0}$. Dabei werden folgende drei Bedingungen an die neuen Gewichte gestellt: Das neue Gewicht einer Kante $(u, v) \in E$ soll mindestens dem Abstand zwischen u und v gemäß unserer Metrik entsprechen. Für alle Originalkanten darf das neue Gewicht höchstens so hoch wie das alte Gewicht sein. Und für jedes Dreieck $\{u, v, w\} \subseteq V$ soll die untere Dreiecksungleichung gelten [Blä+25]. Für eine Kante $(u, v) \in E$ wird dazu $c'((v, w))$ auf das Minimum von den ursprünglichen Kosten $c((v, w))$ und den Kosten über u, also $c'((v, u)) + c'((u, w))$ gesetzt. Auch hier ist die Reihenfolge relevant, in der die Dreiecke durchlaufen werden, da ansonsten Dreiecke mehrfach betrachtet werden müssen. Es gibt jedoch eine Reihenfolge, in der jedes Dreieck höchstens einmal betrachtet werden muss [Blä+25]. Nach der Customization können wir den Augmented Graph definieren als $G' = (V, E', c')$. Die Laufzeit der Customization liegt dann in $\mathcal{O}(|V| \cdot d_v^2)$, wobei d_v den maximalen Knotengrad in G' bezeichnet.

Auch hier verwenden wir wieder verschiedene optimierte Implementierungsdetails, die Buchhold et al. [BSW19] beschrieben haben.

3.4 Queries

Nachdem die Vorberechnungen abgeschlossen sind, können Nutzende konkrete s–t–Pfad Anfragen stellen und erhalten als Rückgabe die Kosten des Pfades. Die schnellste Variante dafür ist die Elimination-Tree-Query-Technik [DSW16]. Dies geschieht wie oben schon erwähnt auf dem Augmented Graph G' und als zusätzliche Datenstruktur benötigen wir nur der Elimination Tree. Diesen können wir einmal am Ende der Contraction berechnen, da er sich durch die Customization oder Query-Berechnung nicht ändert [Blä+25].

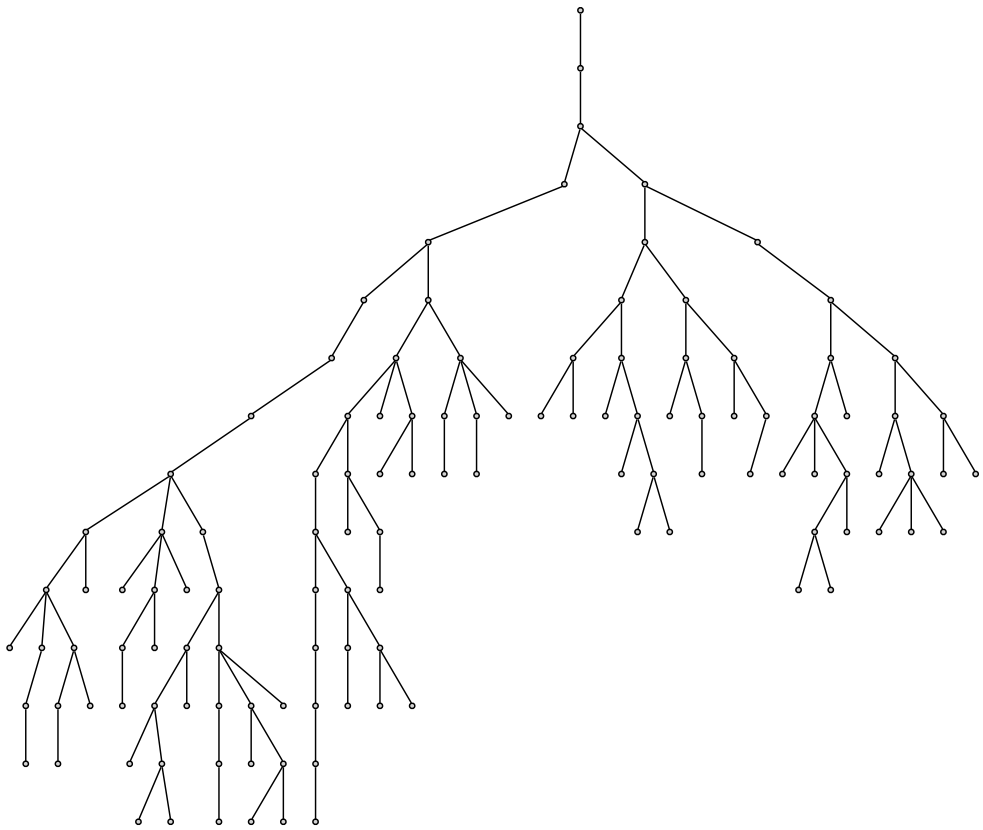

Abbildung 3.4: Beispiel eines Elimination Trees. Die Wurzel hat die höchste Ordnung.

Für die Berechnung eines s–t–Pfades wird der Elimination Tree von s und t nach oben bis zur Wurzel durchlaufen. Für jeden Knoten v auf dem Pfad zur Wurzel werden alle ausgehenden Kanten (v, u) von v in $G^{\uparrow}$ bzw. $G^{\downarrow}$ relaxiert und die Kosten zu von s bzw. t nach u gespeichert.

Buchhold et al. [BSW19] haben auch für Queries einige optimierende Implementierungsdetails vorgestellt, die wir in dieser Arbeit verwenden.

3.5 Elimination Tree

Wie bei der Beschreibung der Queries in Abschnitt 3.4 erwähnt, ist der Elimination Tree eine wichtige Datenstruktur für die Query-Berechnung. Daher werden wir uns in diesem Abschnitt einige Eigenschaften des Elimination Trees ansehen. Abbildung 3.4 zeigt beispielhaft den Elimination Tree des Karlsruher Stadtteils Stupferich. Für einen Knoten $x \in V$ fassen wir alle Knoten, die im Elimination Tree auf dem Pfad von x zur Wurzel liegen, zum Knotensuchraum von x zusammen und schreiben diesen als $S_V(x) \subseteq V$. Alle Kanten, die auf dem Weg von x zur Wurzel relaxiert werden, fassen wir zum Kantensuchraum $S_E(x) \subseteq E$ von x zusammen. Die Suchräume sind ein Maß für die Effizienz der Queries, je größer die Suchräume sind, desto langsamer wird die Berechnung der Queries [Blä+25].

Wir nennen einen Knoten *Kreuzungsknoten*, falls er im Elimination Tree mindestens zwei Kinder hat. Alle Knoten von der Wurzel bis zum ersten Kreuzungsknoten im Elimination Tree sind Teil des ersten Separators. Danach spaltet sich der Elimination Tree in die jeweiligen Zusammenhangskomponenten, in die der Graph durch den Separator zerfällt. Dieses Schema setzt sich fort: Die Kinder eines Kreuzungsknotens bis zum nächsttieferen Kreuzungsknoten gehören zu je einem Separator [Blä+25]. Um die Kosten eines s–t–Pfades zu berechnen, muss der Elimination Tree, wie in Abschnitt 3.4 beschrieben, von s und t bis zur Wurzel durchlaufen werden [Blä+25].

Je tiefer der Elimination Tree ist, desto größer werden die Knotensuchräume von s und t, somit dauert die Berechnung eines Query länger. Fügt man in der Contraction dagegen viele Shortcuts hinzu, wachsen die Kantensuchräume. Dies beeinträchtigt die Querylaufzeiten ebenfalls. Um die Laufzeit der Queries möglichst gering zu halten, dürfen also sowohl die Knoten- als auch die Kantensuchräume nicht zu groß werden [DSW16 | Blä+25]. Ein anderes Problem zu vieler Shortcuts ist, dass die Vorberechnungen deutlich länger dauern können. Außerdem kann es, selbst auf Hochleistungsrechnern, durch die vielen Kanten zu Speicherproblemen kommen [Blä+25].

4 Lösungsansätze

Grundsätzlich ist der CCH-Algorithmus, wie er in Kapitel 3 beschrieben wurde, bekannt. Ziel dieser Arbeit ist es aber, den Einfluss von neuen Kanten auf den CCH Algorithmus zu diskutieren. Wenn wir eine neue Kante (x, y) in den Graphen einfügen, ändern wir die Topologie des Graphen und nach der Definition des CCH-Algorithmus müssen wir alle Phasen neu durchlaufen [DSW16]. Eine neue Ordnung zu berechnen ist aber sehr teuer, sodass wir diesen Schritt vermeiden möchten. Die restlichen Vorberechnungen, die Contraction und Customization, sind dagegen günstig und eine erneute Ausführung dieser Schritte hat asymptotisch keine Auswirkungen auf die Gesamtlaufzeit [Blä+25 | DSW16].

Die Ordnung beeinflusst zwar nicht die Korrektheit der Queries, wohl aber die Effizienz des Algorithmus [Blä+25]. Das heißt auf jeder Ordnung, und insbesondere der ursprünglichen, werden die Queries korrekt berechnet. Allerdings leidet die Performance des gesamten Algorithmus und besonders der Queries unter einer schlechten Ordnung [Blä+25].

In diesem Kapitel beschreiben wir verschiedene Varianten, wie die Ordnung an die neue Kante angepasst werden kann. Anschließend führen wir die Contraction und Customization gemäß der neuen Ordnung durch. In Kapitel 5 analysieren wir schließlich die Auswirkungen von unseren Optimierungen. Wir bezeichnen mit (x, y) stets die neu eingefügte Kante und mit π die alte Ordnung.

4.1 Shortcut-Kante einfügen

Sei (x, y) ein Shortcut. Wenn wir diese Kante in den Ausgangsgraphen neu einfügen, ist die alte Ordnung für den neuen Graphen nach Lemma 4.1 ebenfalls optimal. Aus diesem Grund übernehmen wir immer, wenn eine Shortcut-Kante in den Graphen eingefügt wird, die alte Ordnung und verfahren gemäß Abschnitt 4.3.

Lemma 4.1: *Für einen Graphen $G = (V, E)$ sei $G' = (V, E')$ der kontrahierte Graph bezüglich einer Ordnung π und $e \in E' \setminus E$ ein Shortcut. Dann wird der Graph $(V, E \cup \{e\})$ mit der Ordnung π ebenfalls zu G' kontrahiert.*

Beweis. Durch eine neue Kante im Ausgangsgraphen wird die Nachbarschaft der Knoten höchstens vergrößert. Da für einen Knoten $v \in V$ seine Nachbarschaft mit höherer Ordnung in G' schon eine Clique ist, kann keine weitere Kante mehr hinzugefügt werden. Ist e bereits im Ausgangsgraphen enthalten, muss die Kante in der Contraction nicht mehr hinzugefügt werden, dies beeinflusst aber nicht die Cliqueneigenschaft. Da die Ordnung π nicht geändert wird, bleibt der Graph nach der Contraction insgesamt unverändert. ∎

4.2 Allgemeine Ziele

Nach Abschnitt 4.1 können wir ohne Beschränkung der Allgemeinheit davon ausgehen, dass es sich bei der neuen Kante $e = (x, y)$ nicht um einen Shortcut handelt. Durch das Einfügen der Kante ändern wir also die Topologie des Augmented Graphs. Wie oben schon erwähnt, müssten wir nach der Definition des CCH-Algorithmus die Ordnung komplett neu berechnen.

Sei G der ursprüngliche Graph und G^* der Graph, in den die neue Kante eingefügt wurde. Wir definieren, dass die Kante (x, y) einen Separator schneidet, falls x und y nicht in der gleichen Komponente sind, nachdem der Separator herausgeschnitten wurde. Dabei ist es im Allgemeinen möglich, dass entweder x oder y teil des Separators ist. Vergleichen wir die Berechnung der Ordnung auf beiden Graphen, gibt es im ersten Rekursionsschritt zwei Möglichkeiten: Entweder liegt e komplett innerhalb einer Komponente oder e schneidet den gefundenen Separator. Im ersten Fall wird der Algorithmus in G^* denselben Separator finden wie in G. Das bedeutet, dass sich die Ordnung in den Komponenten ohne e sowie im gefundenen Separator nicht verändert. Änderungen können also nur in der Komponente mit der neuen Kante auftreten. Dieses Argument setzt sich rekursiv so lange fort, bis die neue Kante erstmalig einen gefundenen Separator schneidet, dann befinden wir uns im zweiten Fall. Nur in diesem Teil des Graphen unterschieden sich die gefundenen Separatoren von G und G^* und im Allgemeinen wird sich die Ordnung für die gesamte Komponente grundlegend ändern.

Wir sehen also ein, dass es ausreicht, die Ordnung lokal an die neue Kante anzupassen. Ziel ist es daher, die Ordnung so zu verändern, dass sowohl die Vorberechnungen, als auch die Queries möglichst effizient berechnet werden können.

4.3 Alte Ordnung übernehmen

Damit die Vorberechnungen möglichst schnell sind, ist ein trivialer Ansatz, die alte Ordnung zu übernehmen. In diesem Fall müssen wir nur die Contraction und Customization erneut durchführen. Wie oben erwähnt, werden die Queries dadurch weiterhin korrekt berechnet. Nachdem die Ordnung π aber nicht auf die neue Kante optimiert wurde, erwarten wir, dass die Querylaufzeiten deutlich langsamer sind, als bei den folgenden Varianten.

4.4 Ordnung vom Teilbaum neu berechnen

Eine entgegengesetzte Strategie besteht darin, ausschließlich die Querylaufzeiten zu optimieren. Dazu berechnen wir eine neue Ordnung π_r. Wie in Abschnitt 4.2 ausgeführt, reicht es, eine neue, optimale Ordnung für die Komponente $K \subseteq V$ zu berechnen, in der die neue Kante (x, y) das erste Mal einen Separator schneidet. In K wird also im ursprünglichen Graphen das erste Mal ein Separator $S \subsetneq K$ gewählt, der x und y in unterschiedliche Komponenten teilt. Im Elimination Tree ist S der niedrigste Separator, der sowohl auf dem Pfad von x, als auch von y aus zur Wurzel liegt. Die Knoten aus K sind dann genau die Knoten, die im Elimination Tree in dem, an S gewurzelten Teilbaum liegen. In π_r berechnen wir dann für alle Knoten aus K eine neue Ordnung gemäß dem Inertial-Flow-Algorithmus aus Abschnitt 3.2.1. Alle übrigen Knoten aus dem Graphen behalten ihre ursprüngliche Ordnung.

Um alle Knoten aus K effizient zu berechnen, suchen wir zunächst den *höchsten Separatorknoten* $t \in V$ von x und y. Dabei handelt es sich um den Knoten aus S mit der höchsten Ordnung in π. Für die Berechnung von t benötigen wir zuvor den *kleinsten gemeinsamen Vorfahren* (lca) von x und y und nennen diesen ℓ. Dieser ist definiert, als der Knoten $\ell \in V$ mit kleinster Ordnung in π, der sowohl in dem Pfad von x als auch von y zur Wurzel enthalten ist. Von ℓ aus durchlaufen wir den Elimination Tree weiter nach oben und der erste Knoten mit einem Geschwisterknoten ist der höchste Separatorknoten ℓ. Der lca sowie der höchste Separatorknoten sind beide eindeutig. Das Beispiel in Abbildung 4.1 zeigt schematisch, wie die Ordnung für eine neue Kante (x, y) berechnet wird.

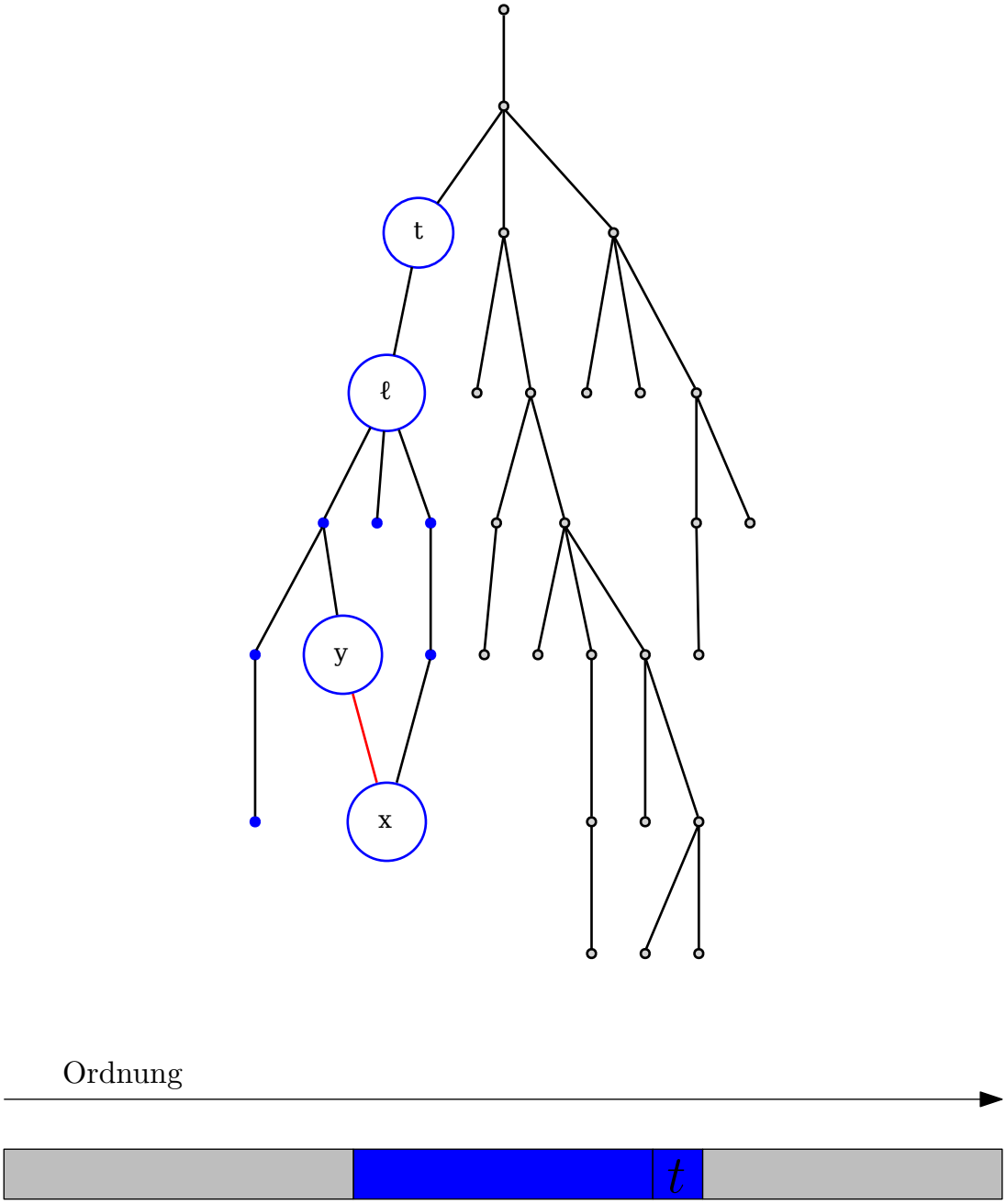

Abbildung 4.1: Schematische Übersicht, wie die Ordnung für eine neue Kante (x, y) neu berechnet wird. Oben der Elimination Tree des Graphen, in dem die Kante (x, y) rot eingezeichnet wurde. Unten die neu berechnete Ordnung π_r. Der höchste Separator, der von (x, y) geschnitten wird, besteht aus den Knoten $\{\ell, t\}$. Gleichzeitig handelt es sich ℓ um den lca von x und y, und bei t um Knoten mit der höchsten Ordnung aus dem Separator. Die Ordnung wird für alle blauen Knoten neu berechnet, alle grauen Knoten behalten ihre ursprüngliche Ordnung.

Das Neuberechnen der Ordnung für einen Teilbaum ist allerdings teuer. Es gibt verschiedene Algorithmen, um eine Nested Dissections Ordnung zu berechnen, ein Beispiel ist der Inertial-Flow-Algorithmus aus Abschnitt 3.2.1. Alle bekannten Algorithmen für dieses Problem benötigen aber auf großen Instanzen mehrere Minuten [SS15 | Blä+25]. Falls die neue Kante den höchsten Separator schneidet, muss eine neue Ordnung für den gesamten Graphen berechnet werden. Folglich ist die Laufzeit dieser Variante beschränkt durch die initiale Berechnung der Ordnung auf dem Graphen. Für kleine Teilbäume wird die Laufzeit der Vorberechnung noch durch die Contraction und Customization dominiert. Wenn der Teilbaum aber groß genug wird, dominiert das Neuberechnen die Laufzeit. Allerdings erwarten wir, dass die so berechnete Ordnung sehr gut auf die neue Kante optimiert ist. Daher gehen wir davon aus, dass diese Variante zu den schnellsten Querylaufzeiten führt.

4.5 Ordnung anpassen

Damit wir sowohl in der Vorberechnung als auch in den Queries schnelle Ergebnisse erhalten, können wir als Kompromiss die Ordnung von einem der beiden Endpunkte der neuen Kante erhöhen. Wie oben sei $K \subseteq V$ die Komponente mit zugehörigem Separator $S \subsetneq K$, in der die neue Kante (x, y) erstmals den gewählten Separator schneidet. Außerdem sei G wieder der originale Graph und G^* der Graph, in dem die Kante (x, y) eingefügt wurde. Für die neue Ordnung π_a fügen wir entweder x oder y zu dem Separator S hinzu und erhöhen dadurch die Ordnung des entsprechenden Knotens. Nachdem S ein Separator für K in G ist, sind sowohl $S \cup \{x\}$, als auch $S \cup \{y\}$ Separatoren für K in G^*. Aus dem Max-Flow-Min-Cut-Theorem folgt, dass der minimale s–t–Cut in einem Graphen durch eine weitere Kante nicht kleiner werden kann [EFS56]. Folglich kann auch der minimale Separator für K in G^* nicht kleiner werden als S. Insgesamt erhalten wir also durch die Anpassung in der Komponente einen Separator, der maximal ein Knoten größer ist, als ein optimaler Separator. Innerhalb der einzelnen Komponenten, in die K durch den neuen Separator zerfällt, bleiben die Separatoreigenschaften auf der jeweiligen Ebene erhalten. Unsere Aussagen beziehen sich allerdings auf die Optimalwerte, wie in Abbildung 4.2 kann es passieren, dass in einem optimalen Separator andere Knoten enthalten sind, als in unserem angepassten Separator.

Der verbleibende Freiheitsgrad bei dieser Variante ist, welchem der beiden Endpunkte wir eine höhere Ordnung geben möchten und wir bezeichnen den gewählten Knoten mit $s \in \{x, y\}$. Mögliche Strategien sind immer den Knoten mit aktuell höherer bzw. niedrigerer Ordnung in π zu wählen sowie eine zufällige Wahl zu treffen. Den Einfluss der Wahl evaluieren wir experimentell in Abschnitt 5.2. Wir erwarten aber, dass sich die Strategien nur geringfügig unterscheiden.

Für die konkrete Berechnung der neuen Ordnung π_a benötigen wir wieder den lca ℓ von x und y. Anschließend möchten wir dem Knoten s die nächst niedrigere Ordnung relativ zu ℓ geben. Für alle Knoten $v \in V$ mit $\pi(s) < \pi(v) < \pi(\ell)$ reduzieren wir die Ordnung von v in π_a um eins. Schließlich bekommt s die Ordnung $\pi_a(s) = \pi(l) - 1$. Alle übrigen Knoten behalten ihre ursprüngliche Ordnung bei. Abbildung 4.3 zeigt die Anpassung der Ordnung schematisch.

Diese Anpassung der Ordnung ist günstig. Die Laufzeit, um den lca zweier Knoten zu berechnen, ist nach oben beschränkt durch die Tiefe des Elimination Trees. Im schlimmsten Fall handelt es sich beim Elimination Tree um einen Pfad und wir müssen somit linear viele Knoten betrachten. Allerdings erwarten wir, dass der Elimination Tree balanciert ist, dann ist seine Tiefe logarithmisch in der Anzahl an Knoten. Im Worst Case muss danach die Ordnung

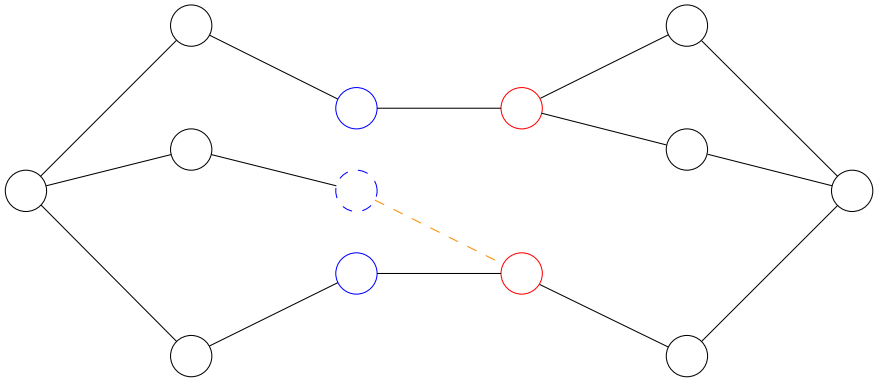

Abbildung 4.2: Beispiel, in dem der von uns angepasste Separator andere Knoten als ein optimaler Separator enthält. Die neue Kante ist orange gestrichelt dargestellt. Der vom Inertial-Flow-Algorithmus ursprünglich gewählte, optimale Separator ist in Blau und durchgehend gezeichnet, der zum Separator hinzugefügte Knoten ist blau gestrichelt. Die roten Knoten bilden einen optimalen Separator, der von unserem vollständig disjunkt ist.

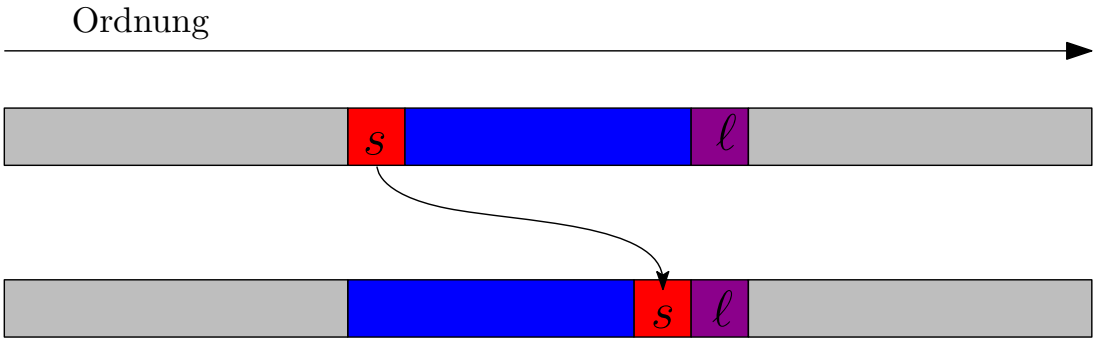

Abbildung 4.3: Schematische Übersicht, wie die Ordnung für eine neue Kante (x, y) angepasst wird. Oben die ursprüngliche Ordnung π, unten die angepasste Ordnung π_a. Wir geben dem roten Knoten $s \in \{x, y\}$ eine höhere Ordnung und fügen s dazu direkt vor dem violetten lca ℓ von x und y ein. Die Ordnung der blauen Knoten wird somit um eins reduziert. Die Ordnung der grauen Knoten bleibt unverändert.

von allen Knoten in dem Teilbaum unter dem lca angepasst werden. Das sind aber maximal linear viele, wenn der lca eine sehr hohe Ordnung hat, andernfalls deutlich weniger. Insgesamt liegt die Laufzeit dieser Variante also in $\mathcal{O}(|V|)$ und wird somit durch die folgende Contraction und Customization dominiert. Es ist also deutlich schneller, die Ordnung anzupassen, als sie für einen Teilbaum neu zu berechnen. Unsere Erwartung ist, dass die Queries durch die Anpassung schneller bearbeitet werden, im Vergleich zur alten Ordnung (Abschnitt 4.3), aber langsamer verglichen mit einer Neuberechnung der Komponente (Abschnitt 4.4).

5 Evaluation

In diesem Kapitel vergleichen wir die in Kapitel 4 vorgestellten Optimierungen. Dazu berechnen wir zunächst eine initiale Ordnung π. Anschließend fügen wir eine neue Kante der Länge d in den Graphen ein und berechnen die neuen, angepassten Ordnungen gemäß den Abschnitten 4.3, 4.4 und 4.5. Dabei untersuchen wir, wie sich die verschiedenen Varianten auf neue Straßen der Länge 1 km, 5 km 10 km, …, 40 km, 45 km und 50 km auswirken. Die Längen wurden bewusst so gewählt, dass wir sowohl unwichtige als auch sehr wichtige Kanten neu einfügen.

Um eine neue Kante der Länge d zu erzeugen, ziehen wir zufällig einen Knoten x aus allen Knoten. Danach berechnen wir, welche Knoten euklidischen Abstand $d \pm 10\ \%$ zu x haben und ziehen daraus zufällig einen Knoten y. Die neue Kante (x, y) fügen wir dann in beide Richtungen ein. Damit neue Kanten sich nicht gegenseitig beeinflussen, fügen wir in den Ausgangsgraphen immer nur eine Kante auf einmal und löschen alle Änderungen, bevor wir die nächste Kante einfügen.

Die Qualität der Ordnungen vergleichen wir, indem wir die Suchräume der Knoten analysieren. Wie in Abschnitt 3.5 erwähnt, sind die Suchräume ein gutes Maß für die Geschwindigkeit der Queries. Je kleiner die Suchräume sind, desto schneller können die Queries berechnet werden [DSW16]. Wir berechnen dann die durchschnittliche und maximale Größe der Suchräume für alle Knoten im Graphen sowie für die Knoten im gemeinsamen Teilbaum der neuen Kante. Werden bei der Berechnung alle Knoten des Graphen berücksichtigt, nennen wir die Suchräume global, andernfalls lokal. Die Größe der globalen Suchräume sind dann ein Maß für zufällige Queries [DSW16]. Die lokalen Suchräume sind dagegen ein Indikator für die Geschwindigkeit von Queries in der Nähe der neuen Kante. Dabei definieren wir, dass sich alle Knoten im gemeinsamen Teilbaum der neuen Kante in ihrer der Nähe befinden. Die neue Kante beeinflusst also potenziell alle Queries in ihrer Nähe. Auf Queries, die sich nicht in ihrer Nähe Kante befinden, hat die Kante dagegen keine Auswirkungen. Aus den maximalen Suchräumen erhält man eine gute Worst Case Abschätzung. Im Gegensatz dazu bieten die durchschnittlichen Suchräume Average Case-Bewertungen [DSW16].

5.1 Vergleich der Varianten

Zunächst vergleichen wir die Varianten, die alte Ordnung zu übernehmen aus Abschnitt 4.3, die Ordnung von dem Teilbaum neu zu berechnen aus Abschnitt 4.4 und die Ordnung anzupassen aus Abschnitt 4.5. Dazu berechnen wir für jeden Knoten seine Suchräume und analysieren, wie sich diese verändern. Wie oben beschrieben können wir beim Anpassen der Ordnung den Knoten, dem wir eine höhere Ordnung geben, wählen. In diesem Abschnitt erhöhen wir immer den Knoten mit aktuell niedrigerer Ordnung. Welchen Einfluss die Wahl des Knotens konkret hat, untersuchen wir dann in Abschnitt 5.2. Bei den Experimenten in diesem Abschnitt fügen wir für jede Kantenlänge 10 neue Kanten nacheinander ein.

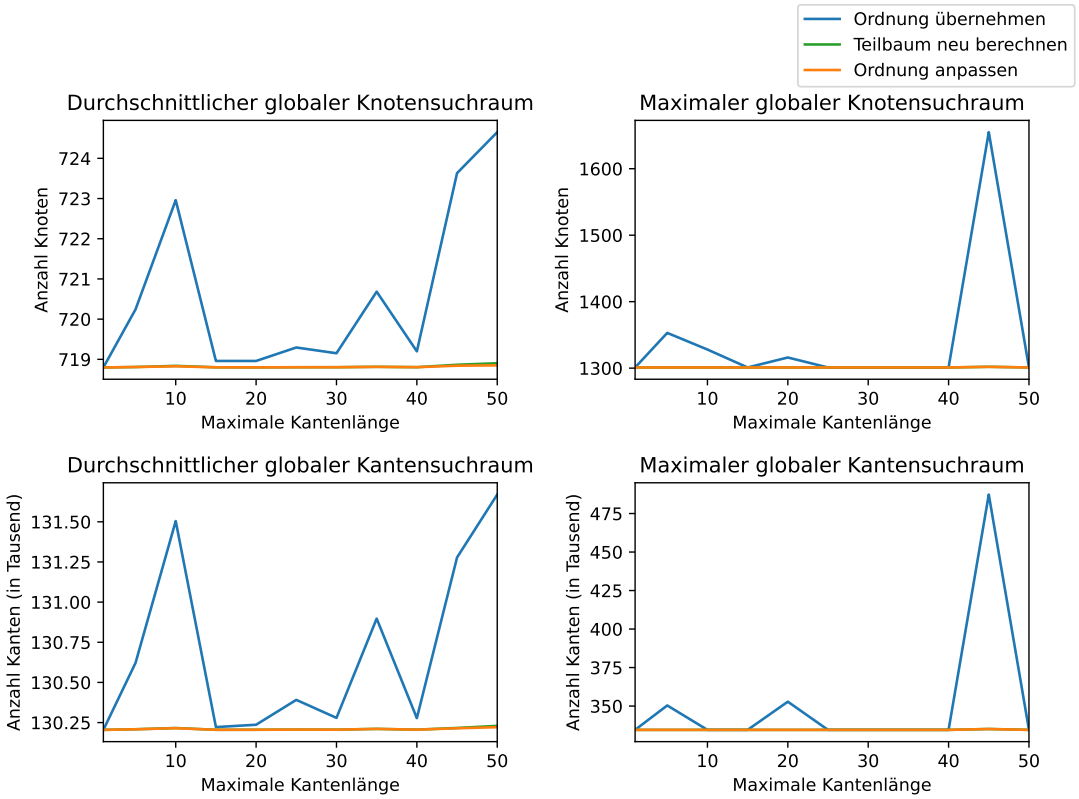

Abbildung 5.1: Plot, in dem die globalen Suchräume auf dem Europa Graphen für die Varianten „Ordnung übernehmen“, „Teilbaum neu berechnen“ und „Ordnung anpassen“ in Relation zu der eingefügten Kantenlänge gezeigt wird. Oben links sind die durchschnittlichen Knotensuchräume zu sehen. Unten links sieht man die durchschnittlichen Kantensuchräume. Rechts sind die zugehörigen maximalen Suchräume der Knoten (oben) und Kanten (unten) abgebildet. Die alte Ordnung zu übernehmen, führt in allen Fällen zu den größten Suchräumen. Die Suchräume der anderen beiden Varianten sind ähnlich groß, sodass in den meisten Fällen nur eine der beiden Linien zu sehen ist.

5.1.1 Globale Sichtweise

Betrachten wir zunächst den Einfluss von unseren Optimierungen auf zufällige Queries. Im Vordergrund steht dabei die Geschwindigkeit, mit der diese in der jeweiligen Ordnung berechnet werden können. Wie oben erwähnt, sind die globalen Suchräume dafür ein gutes Maß. In Abbildung 5.1 sieht man, dass die alte Ordnung immer deutlich größere Suchräume hat, als die anderen beiden Varianten. Die erwarteten Querylaufzeiten sind dadurch bei der Variante „alte Ordnung übernehmen“ merklich langsamer als bei den anderen. Interessiert man sich für die Worst Case Laufzeiten der Queries, sind die Unterschiede sogar noch stärker. Im Gegensatz dazu unterscheiden sich die Größen der Suchräume von den Varianten „Ordnung anpassen“ und „Teilbaum neu berechnen“ nur sehr geringfügig voneinander.

Das Neuberechnen der Ordnung für einen Teilbaum ist deutlich teurer, als das Anpassen der Ordnung. Fügen wir eine längere Kante ein, steigt die Wahrscheinlichkeit, dass diese weiter entfernte Teilbäume miteinander verbindet und somit höhere Separatoren schneidet. Wie erwartet, sieht man daher in Abbildung 5.2, dass die Größe des gemeinsamen Teilbaums

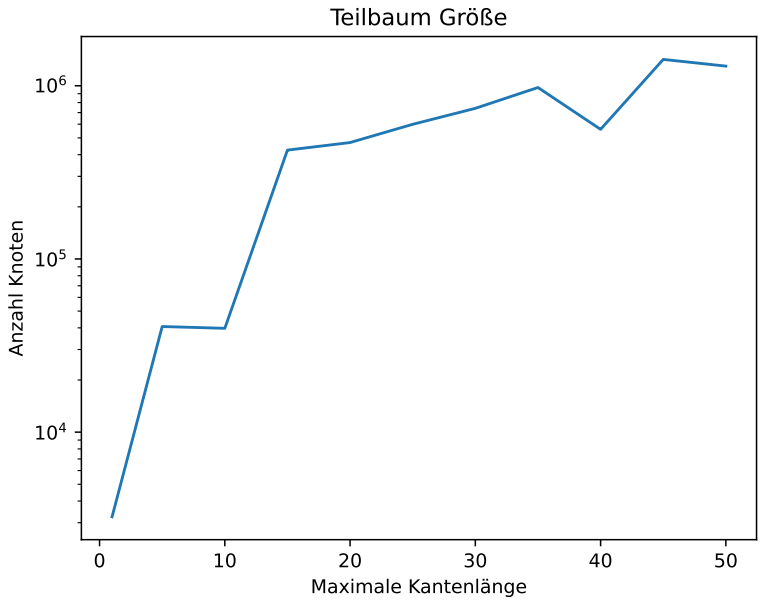

Abbildung 5.2: In dem Diagramm wird die Größe des gemeinsamen Teilbaums von der eingefügten Kante in Relation zu ihrer Länge gezeigt. Zu sehen ist, dass die Größe der Teilbäume in der Länge der eingefügten Kante wächst. Im Gegensatz zu den übrigen Experimenten in diesem Abschnitt wurde hier für jede Länge der Durchschnitt aus 100 anstatt 10 eingefügten Kanten gebildet.

mit der Länge der eingefügten Kante wächst. Das führt dazu, dass das Neuberechnen für längere Kanten auch deutlich langsamer wird. Die Laufzeit der Neuberechnung ist nach oben beschränkt, durch die Laufzeit eine initiale Ordnung für den gesamten Graphen zu berechnen, und liegt also maximal bei mehreren Minuten [SS15 | Blä+25]. Bei dem Anpassen der Ordnung muss dagegen maximal die Ordnung von jedem Knoten verschoben werden und ist damit auch für große Teilbäume deutlich schneller.

Verglichen mit der alten Ordnung liefern das Anpassen sowie das Neuberechnen für einen Teilbaum ähnlich gute Ordnungen. Daher vergleichen wir diese beiden Varianten in Abbildung 5.3 direkt miteinander. Man sieht, dass beide Varianten sich im Worst Case nahezu identisch verhalten. Im Average Case stellen wir sogar überraschenderweise fest, dass das Anpassen insgesamt zu kleineren Suchräumen führt. Dies widerspricht unserer Erwartung, dass das Neuberechnen die besten Ordnungen liefert. Nachdem diese Abweichungen nicht nur in einzelnen Datenpunkten auftreten, sollten weiterführende Arbeiten dieses Phänomen genauer untersuchen. Erwartungsgemäß wachsen aber die durchschnittlichen Suchräume in der Länge der Kanten für diese beiden Varianten.

Zusammenfassend ist das Anpassen der Ordnung für zufällige Queries eine gute Lösung. Diese Variante ist in der Vorberechnung effizienter als das Neuberechnen und die Queries können schneller beantwortet werden, als bei der alten Ordnung. Außerdem skaliert das Anpassen der Ordnung auch für lange Kanten sehr gut.

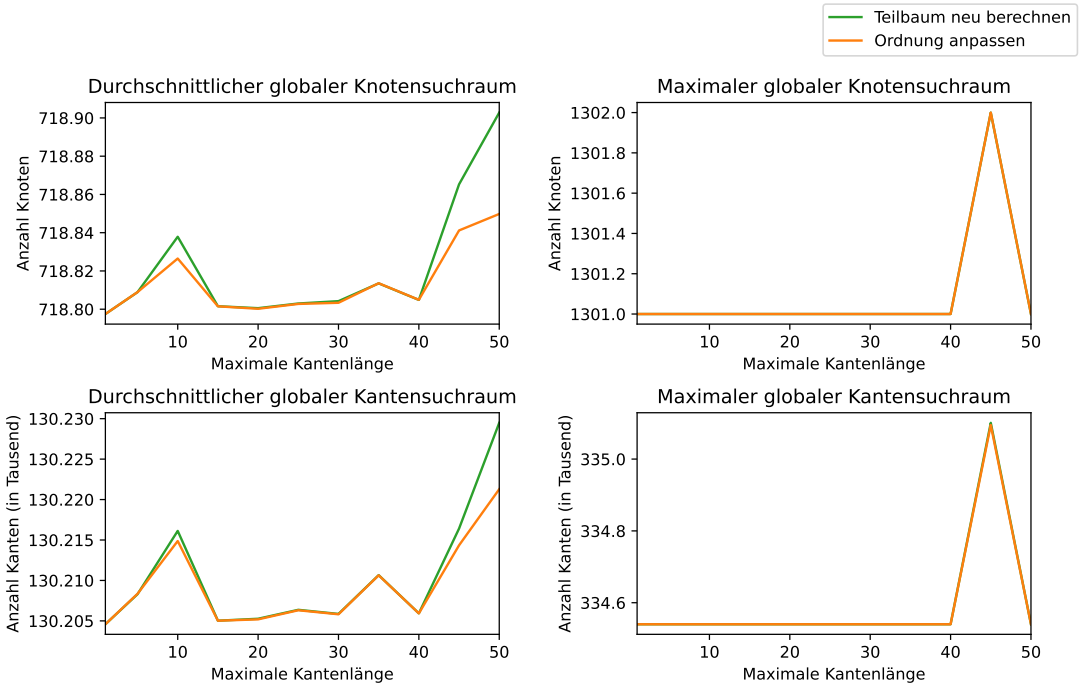

Abbildung 5.3: Die globalen Suchräume der Varianten „Teilbaum neu berechnen“ und „Ordnung anpassen“ in Abhängigkeit zu der Länge der eingefügten Kante. Oben links der durchschnittliche Knotensuchraum. Oben rechts der zugehörige maximale Knotensuchraum. Unten links der durchschnittliche und unten rechts der maximale Kantensuchraum. Die maximalen Suchräume unterscheiden sich nur sehr geringfügig. „Teilbaum neu berechnen“ hat dagegen größere durchschnittliche Suchräume als „Ordnung anpassen“.

5.1.2 Lokale Sichtweise

Nachdem wir im obigen Abschnitt zufällige Queries betrachtet haben, untersuchen wir hier lokale Queries in der Nähe der neuen Kante genauer. Wie oben schon erwähnt, kann die neue Kante nur Knoten im gemeinsamen Teilbaum, also in ihrer Nähe, beeinflussen. In den lokalen Suchräumen sind die Auswirkungen unserer Optimierungen also noch besser messbar.

In Abbildung 5.4 sieht man, dass auch lokal die Variante „Ordnung übernehmen“ deutlich schlechter performt, als die anderen beiden. Hier sind die Unterschiede sogar noch ausgeprägter als im Abschnitt 5.1.1. Die Varianten „Teilbaum neu berechnen“ und „Ordnung anpassen“ sind dagegen im Verhältnis auch wieder sehr ähnlich zueinander. Damit wird unsere Erwartung erneut bestätigt, dass die alte Ordnung zu den längsten Queryzeiten führt und dass „Ordnung anpassen“ eine gute Alternative zum Neuberechnen darstellt.

Vergleicht man die Varianten „Ordnung anpassen“ und „Teilbaum neu berechnen“ direkt, ist auch hier das Neuberechnen überraschenderweise schlechter als das Anpassen. Abbildung 5.5 zeigt die Differenz der Suchräume zwischen dem Neuberechnen und Anpassen. Unsere Erwartung ist, dass die Differenz immer positiv ist, weil wir vermuten, dass Neuberechnen bessere Ergebnisse liefert als Anpassen. Aber auch hier müssen wir feststellen, dass das Anpassen oft sogar kleinere Suchräume liefert. Allerdings sind die Unterschiede zwischen den beiden Varianten sehr gering und stark verrauscht, sodass auch dieses Phänomen in weiteren Arbeiten untersucht werden sollte.

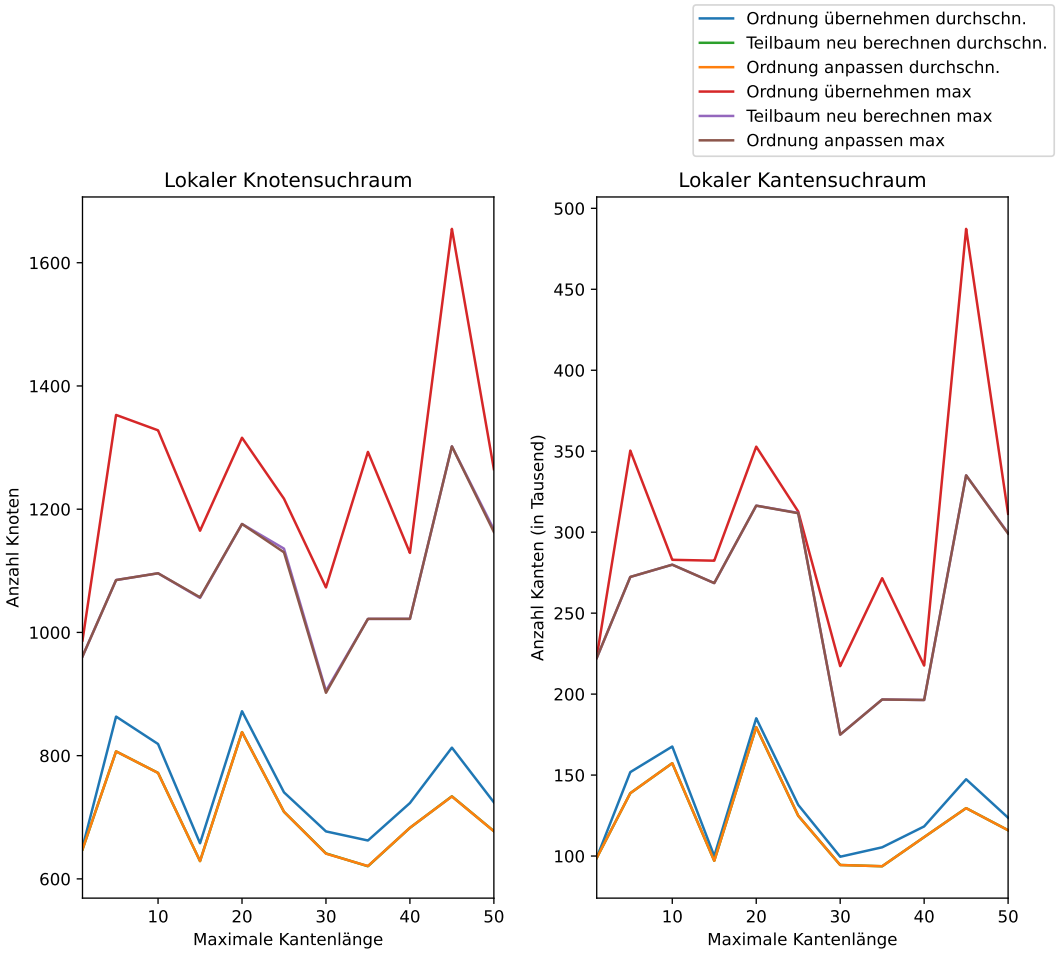

Abbildung 5.4: Lokale Suchräume in Abhängigkeit der Länge der neuen Kante für die Varianten „Ordnung übernehmen“, „Teilbaum neu berechnen“ sowie „Ordnung anpassen“. Die unteren Kurven in den Grafiken zeigen jeweils die durchschnittlichen Suchräume, die oberen die maximalen. Links sind die Knotensuchräume zu sehen und rechts die Kantensuchräume. Die Suchräume von der alten Ordnung sind immer deutlich größer als die der anderen beiden Varianten.

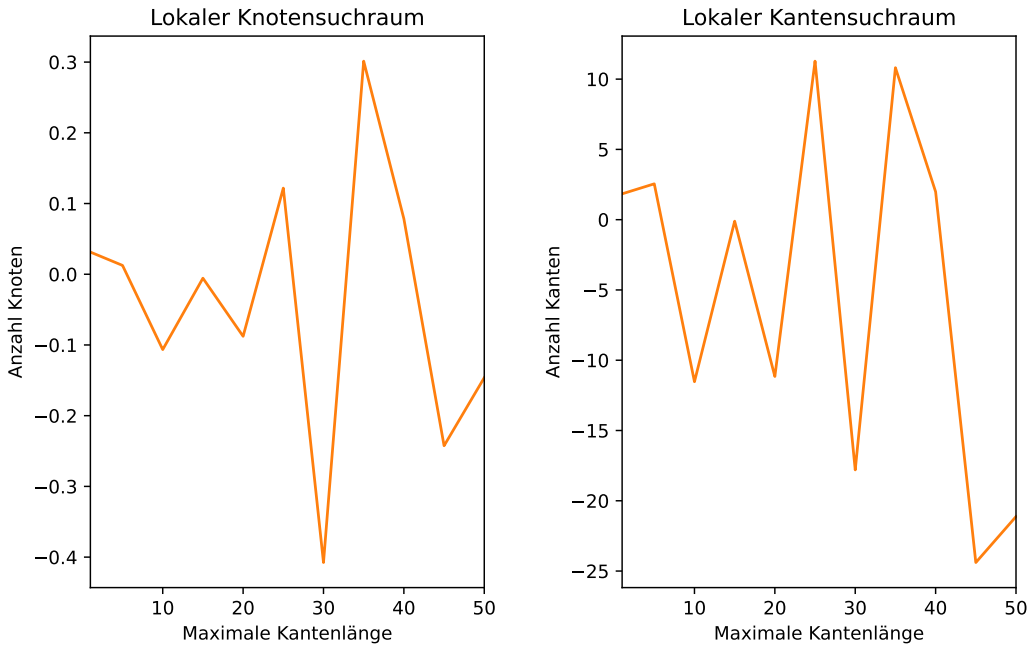

Abbildung 5.5: Lokale Suchräume in Abhängigkeit der Länge der neuen Kante für die Varianten „Teilbaum neu berechnen“ und „Ordnung anpassen“. Gezeigt wird die Differenz der durchschnittlichen Suchraumgrößen zwischen der Neuberechnung und dem Anpassen. Links handelt es sich um die Differenz in Knotensuchräume, rechts um die der Kantensuchräume. Wider erwartend ist die Differenz oftmals negativ, an den Stellen führt Neuberechnen zu schlechteren Ergebnissen als Anpassen.

Insgesamt können wir dennoch festhalten, dass das Anpassen der Ordnung ein sehr gutes Verfahren ist, um im CCH-Algorithmus mit neuen Kanten umzugehen. Dieses Verfahren liefert mit wenig Aufwand auch für lange und wichtige Kanten schnell gute Ordnungen. Dadurch kann der Algorithmus effizient auf Änderungen in der Topologie reagieren und Queries weiterhin in kurzer Zeit beantworten.

5.2 Unterschiede beim Anpassen der Ordnung

Im obigen Abschnitt haben wir festgestellt, dass wir durch das Anpassen der Ordnung sowohl die Vorberechnungen, als auch die Queries schnell bearbeiten können. Der verbleibende Freiheitsgrad beim Anpassen der Ordnung aus Abschnitt 4.5 ist, welchem der beiden Endpunkte der neuen Kante wir eine höhere Ordnung geben. Wir unterscheiden dabei zwischen den Varianten, bei denen der Knoten mit aktuell niedriger bzw. höherer Ordnung gewählt wird, sowie einer zufälligen Wahl. Wie in Abschnitt 5.1 fügen wir dazu neue Kanten in den Graphen ein und passen die Ordnung auf verschiedene Varianten an. Allerdings fügen wir bei den Experimenten in diesem Abschnitt für jede Kantenlänge 100 anstatt 10 Kanten nacheinander ein. Anschließend berechnen wir für jeden Knoten seine Suchräume. Nachdem die Unterschiede zwischen den Varianten hier sehr viel geringer sind, fokussieren wir uns dabei nur auf die lokalen Suchräume, weil dort die Unterschiede besser messbar sind.

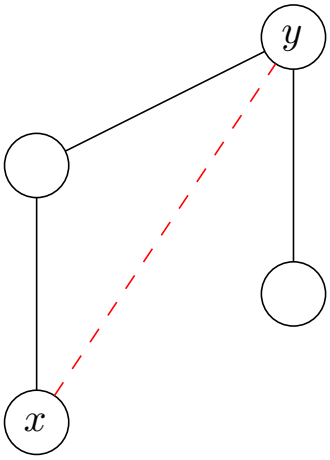

Abbildung 5.6: Beispielgraph, mit neuer Kante (x, y) in rot gestrichelt. Der Graph ist gleichzeitig auch sein Elimination Tree und y ist der lca von x und y. Wird beim Anpassen der Ordnung der Knoten mit aktuell höherer Ordnung, also y, gewählt, wird die Ordnung nicht geändert. Wählt man dagegen den anderen Knoten x, so ändert sich die Ordnung.

Eine erste Beobachtung ist, dass die Ordnung identisch bleiben kann, falls wir den Knoten mit aktuell höherer Ordnung wählen. Wählen wir dagegen den niedrigeren Knoten, ändern wir die Ordnung in jedem Fall. In Abbildung 5.6 sehen wir ein Beispiel eines Elimination Trees, in dem der höhere Endpunkt der neuen Kante (x, y) gleichzeitig der lca von x und y ist. Das bedeutet aber, dass die neue Kante im Separator endet. Folglich können wir bei der Wahl des höheren Knotens nichts ändern. Der Knoten mit aktuell niedrigerer Ordnung kann aber niemals gleichzeitig der lca sein und die Ordnung wird bei dieser Wahl definitiv geändert. Da S von der neuen Kante nicht geschnitten wird, separiert es die Komponente mit der neuen Kante ebenfalls. Eine Erweiterung des Separators ist daher nicht notwendig. Der Algorithmus, den wir für die Nested Dissections Ordnung verwenden, wählt dann also auch mit der neuen Kante denselben Separator S für die entsprechende Komponente. Außerdem hat die neue Kante keinen Einfluss auf die Komponente, in dem sich der andere Endpunkt x der Kante befindet, da sie durch das Herausschneiden des Separators ebenfalls entfernt wird. Das bedeutet, dass sich die Qualität der alten Ordnung unter diesen Umständen durch die neue Kante nicht ändert. Treffen wir in diesem Fall eine andere Wahl, können wir die Qualität der Ordnung nur verschlechtern.

Wie wir in Tabelle 5.1 sehen können, sind die durchschnittlichen, lokalen Suchräume für alle drei Varianten sehr ähnlich. Allerdings sind die Suchräume bei der Wahl des höheren Knotens in den meisten Fällen minimal kleiner. Aus diesem Grund untersuchen wir in Abbildung 5.7 die Änderungen relativ zu der Wahl des höheren Knotens. Dazu ziehen wir von den Suchräumen jeweils den Suchraum von der Wahl des höheren Knotens ab und betrachten die Differenz. Es fällt auf, dass die Schwankungen zwischen den Varianten minimale sind und im Bereich von 0,035 Knoten bzw. 2,5 Kanten liegen. Die Wahl des höheren Knotens führt aber in den meisten Fällen zu den kleinsten durchschnittlichen Suchräumen. Die maximalen Suchräume sind dagegen für alle Varianten vollständig identisch. Das bedeutet, dass die Queries bei der Wahl des höheren Knotens im Average Case minimal schneller berechnet werden können. Im Worst Case sind dagegen keine Unterschiede feststellbar.

Tabelle 5.1: Eine Auswahl der durchschnittlichen, lokalen Suchräume in Abhängigkeit der Kantenlänge für die drei Varianten, die Ordnung anzupassen. Oben Knotensuchräume, unten die Kantensuchräume. Die Größe der Suchräume unterscheiden sich nur sehr geringfügig. In jeder Spalte ist kleinste Eintrag **fett** markiert.

	10	20	30	40	50
Niedrigeren	740,870	724,389	719,761	746,348	**728,260**
Höheren	**740,859**	**724,380**	**719,744**	**746,345**	728,262
Zufälligen	740,872	724,390	**719,744**	746,348	728,261
Niedrigeren	139 467,964	133 521,370	128 942,936	144 129,162	**132 399,587**
Höheren	**139 467,773**	**133 520,815**	**128 940,750**	**144 129,033**	132 399,733
Zufälligen	139 468,246	133 521,381	128 941,326	144 129,168	132 399,613

In der Vorberechnung liegt die Laufzeit von allen Varianten in $\mathcal{O}(n)$, wobei n die Anzahl an Knoten beschreibt. In jeder Variante muss der lca von der neuen Kante (x, y) bestimmt werden. Dies benötigt im Worst Case $\mathcal{O}(n)$ Zeit. Ohne Einschränkung gehen wir davon aus, dass die Ordnung von x kleiner ist, als die von y. Im extremsten Fall hat dann x die niedrigste Ordnung und y die höchste. Wenn wir nun x erhöhen, müssen wir die Ordnung von jedem Knoten verschieben und dafür erneut $\mathcal{O}(n)$ Operationen durchführen. Wählen wir dagegen y, benötigen wir keine weiteren Operationen und sind fertig. Insgesamt müssen wir bei der Wahl des niedrigeren Knotens die Ordnung von mehr Knoten verändern, als wenn wir den höheren Knoten anpassen, da wir mindestens die Ordnung von x zusätzlich verändern. Die Laufzeiten der beiden Varianten unterscheiden sich also nur in einer Konstanten in der Landau Notation.

Betrachten wir dagegen reale, balancierte Elimination Trees, so kann der lca in $\mathcal{O}(\log n)$ bestimmt werden. Die Worst Case Abschätzung ändert sich dadurch zwar nicht, da immer noch bis zu linear viele Knoten existieren, die Ordnung zwischen y und dem lca haben. Allerdings können dann die Unterschiede deutlicher ausfallen. Wählen wir im obigen Beispiel den niedrigeren Knoten x, so müssen wir weiterhin $\mathcal{O}(n)$ Zeit investieren, bei der Wahl von y nur $\mathcal{O}(\log n)$. Wie oben schon erwähnt, wird die Laufzeit für das Anpassen der Ordnung in jedem Fall durch die folgende Contraction und Customization dominiert.

Zusammenfassend können wir also festhalten, dass die Wahl des höheren Knotens zu minimal besseren Ergebnissen führt. Dabei ist diese Variante in der Berechnung mindestens genauso schnell wie die anderen Varianten.

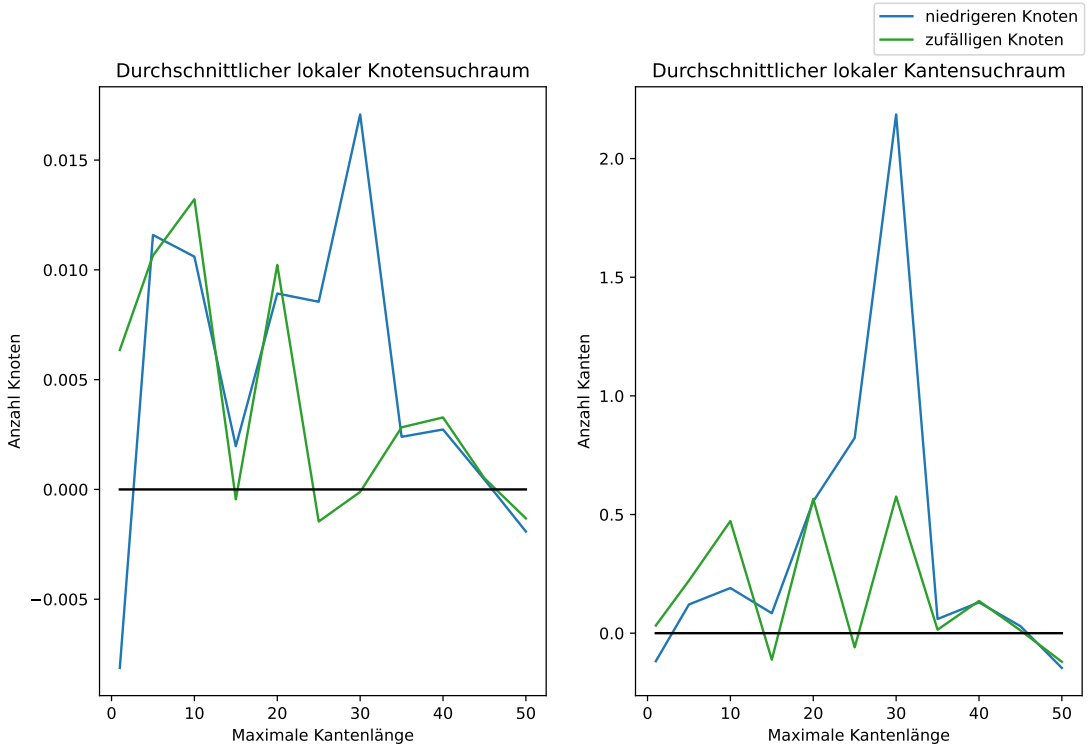

Abbildung 5.7: Lokale Suchräume in Abhängigkeit der Länge der neuen Kante. Gezeigt wird die Differenz der durchschnittlichen Suchraumgrößen zwischen der Wahl des höheren Knotens bzw. einer zufälligen Wahl und der Wahl des höheren Knotens. Links handelt es sich um die Differenz in Knotensuchräume, rechts um die der Kantensuchräume. In den meisten Fällen ist die Differenz positiv, an den Stellen ist die Wahl des höheren Knotens also die bestmögliche. Allerdings führen bei manchen Datenpunkten andere Wahlen zu besseren Ergebnissen. Die schwarze Linie zeigt jeweils den größtmöglichen Unterschied zwischen den maximalen Suchräumen. Dieser ist also offensichtlich für alle Varianten identisch.

6 Zusammenfassung

Wir sehen, dass der CCH-Algorithmus in seiner bisherigen Form nur langsam auf neue Kanten reagieren kann. Durch das Hinzufügen einer Kante ändern wir die Topologie des Ausgangsgraphen und müssen in einem teuren Schritt eine neue Ordnung berechnen. Dazu werden rekursiv kleine, balancierte Separatoren auf dem Graphen gesucht. Der CCH-Algorithmus liefert zwar bei jeder Ordnung korrekte Ergebnisse, allerdings wird der Algorithmus bei einer schlechten Ordnung sehr ineffizient. Nachdem wir die Topologie aber nur geringfügig ändern, ist unser erster Ansatz, die alte Ordnung zu übernehmen. Anschließend stellen wir fest, dass sich die Ordnung nur in dem gemeinsamen Teilbaum der neuen Kante im Elimination Tree ändert, also ab dem Punkt, an dem die neue Kante erstmalig einen Separator schneidet. Daher ist unsere zweite Überlegung, die Ordnung für den gesamten Teilbaum neu zu berechnen. Unsere letzte Idee ist, den geschnittenen Separator um einen der beiden Endpunkte der neuen Kante zu erweitern und die Ordnung somit anzupassen.

Wie erwartet, ist die alte Ordnung in der Query-Phase am langsamsten, dafür sind hier die Vorberechnungen am schnellsten. Die Neuberechnung des Teilbaums soll dagegen vor allem die Querylaufzeiten optimieren. Hier sind dafür die Vorberechnungen, insbesondere für große Teilbäume, am aufwendigsten. Allerdings stellen wir fest, dass die Ordnung anzupassen in vielen Fällen zu minimal kleineren Suchräumen führt und die Queries dadurch noch schneller berechnet werden können. Dieses Phänomen ist insbesondere in der Nähe der neuen Kante gut messbar. Das Anpassen der Ordnung ist auch in der Vorberechnung sehr effizient und die Laufzeit ist vergleichbar mit den Vorberechnung in der alten Ordnung. Durch das Anpassen der Ordnung kann der CCH-Algorithmus also schnell auf neue Kanten reagieren. Die alte Ordnung zu übernehmen, ist dagegen in jedem Fall die schlechteste Variante und sollte vermieden werden.

Beim Anpassen der Ordnung können wir den Endpunkt der Kante wählen, den wir erhöhen. Wir unterscheiden dabei zwischen der Wahl des Knotens mit aktuell niedrigerer bzw. höherer Ordnung sowie einer zufälligen Wahl. Erwartungsgemäß unterschieden sich die Laufzeiten der drei Varianten sowohl in der Vorberechnung als auch bei der Bearbeitung der Queries nur geringfügig. Bei der Wahl des höheren Knotens sind die Vorberechnungen aber minimal schneller. In der Query-Phase ist die Wahl des höheren Knotens in den meisten Fällen ebenfalls besser. Insgesamt führt diese Variante also zu den besseren Ergebnissen.

Zukünftige Arbeiten Unsere Vermutung, dass das Neuberechnen zu den kleinsten Suchräumen führt, konnte nicht bestätigt werden. Daher sollte in weiterführenden Arbeiten untersucht werden, warum das Anpassen der Ordnung zu schnelleren Antworten in der Query-Phase führt. Außerdem haben wir uns bei unseren Experimenten nur auf die Suchraumgrößen fokussiert. Man könnte daher konkrete Queries berechnen, um zu überprüfen, ob die benötigte Zeit zu unseren Ergebnissen passt. Des Weiteren haben wir in dieser Arbeit die initiale Nested Dissection Ordnung nur mittels des Inertial-Flow-Algorithmus berechnet. Man könnte die Effekte von unseren Optimierungen noch für andere Partitionierungsalgorithmen wie den Inertial-Flow-Cutter-Algorithmus untersuchen [GHUW19].

Literatur

[Blä+25] Thomas Bläsius, Valentin Buchhold, Dorothea Wagner, Tim Zeitz und Michael Zündorf. „Customizable Contraction Hierarchies–A Survey“. In: *arXiv preprint arXiv:2502.10519* (2025).

[BSW19] Valentin Buchhold, Peter Sanders und Dorothea Wagner. „Real-time traffic assignment using engineered customizable contraction hierarchies“. In: *Journal of Experimental Algorithmics (JEA)* Jg. 24 (2019), S. 1–28.

[DGJ] Camil Demetrescu, Andrew V Goldberg und David S Johnson. *Dimacs*.

[Din70] Efim A Dinic. „Algorithm for solution of a problem of maximum flow in networks with power estimation“. In: *Soviet Math. Doklady*. Bd. 11. 1970, S. 1277–1280.

[DSW16] Julian Dibbelt, Ben Strasser und Dorothea Wagner. „Customizable contraction hierarchies“. In: *Journal of Experimental Algorithmics (JEA)* Jg. 21 (2016), S. 1–49.

[EFS56] P. Elias, A. Feinstein und C. Shannon. „A note on the maximum flow through a network“. In: *IRE Transactions on Information Theory* Jg. 2 (1956), S. 117–119. DOI: *10.1109/TIT.1956.1056816*.

[GHUW19] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl und Dorothea Wagner. „Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies“. In: *ArXiv* Jg. abs/1906.11811 (2019).

[GJS76] M. R. Garey, David S. Johnson und Larry J. Stockmeyer. „Some Simplified NP-Complete Graph Problems“. In: *Theor. Comput. Sci.* Jg. 1 (1976), S. 237–267.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes und Christian Vetter. „Exact routing in large road networks using contraction hierarchies“. In: *Transportation Science* Jg. 46 (Apr. 2012), S. 388–404. DOI: *10.1287/trsc.1110.0401*.

[SS15] Aaron Schild und Christian Sommer. „On balanced separators in road networks“. In: *International Symposium on Experimental Algorithms*. Springer. 2015, S. 286–297.