Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

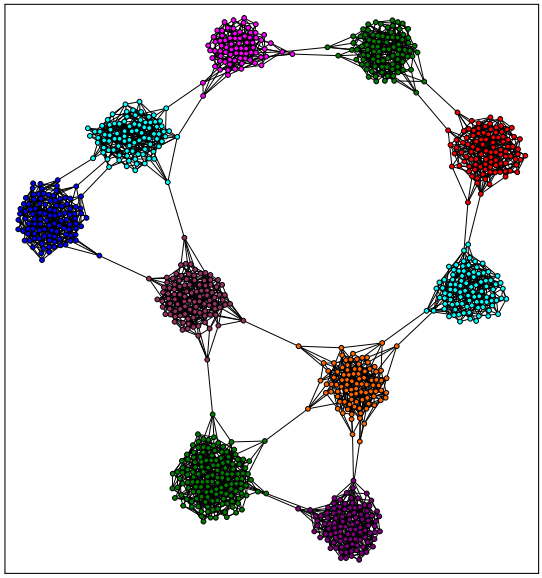
Institut für Theoretische Informatik
Algorithmik I • Prof. Dr. Dorothea Wagner

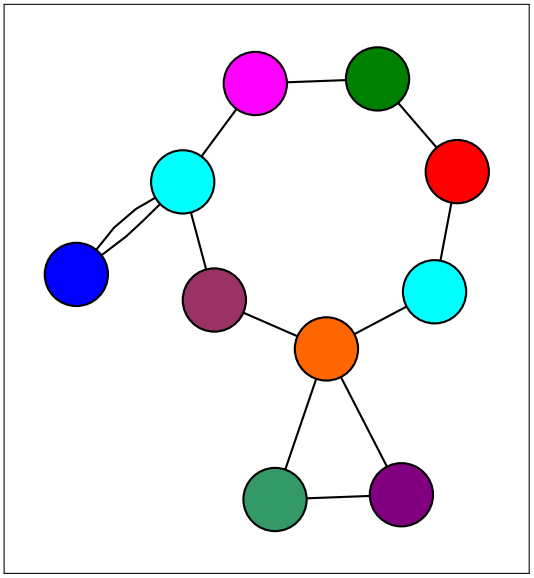
Simultane Schnitte in Graphen

Diplomarbeit

von

Manuel Krings





Betreut durch: Prof. Dr. Dorothea Wagner
Marcus Krug
Ignaz Rutter

September 2009

Die Abbildung auf dem Deckblatt zeigt links einen durch den in Kapitel 4 beschriebenen Graphgenerator erzeugten Graphen. Rechts ist die Kaktus-Repräsentation seiner minimalen Schnitte zu sehen.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, den 7. September 2009 .....

Unterschrift

Inhaltsverzeichnis

1 Einleitung

Minimale Schnitte liefern wichtige Informationen über die Struktur von Graphen. Viele Anwendungen basieren auf der Analyse dieser Schnitte [KS96, PQ82], beispielsweise um Aussagen über die Ausfallsicherheit von Netzwerken zu treffen oder Graphen zu clustern. Häufig werden dabei statische Graphen betrachtet, die Beziehungen zwischen Objekten modellieren, die sich nicht ändern oder einen bestimmten Zeitpunkt repräsentieren. In der Praxis hat man jedoch häufig mit dynamischen Graphen zu tun; die Beziehungen zwischen den durch den Graph repräsentierten Objekten verändern sich und zu verschiedenen Zeitpunkten oder in verschiedenen Zeitfenstern erhält man Graphen mit unterschiedlichen Kantenmengen; diese kann man als Familien von Graphen betrachten. Auch bei Graphfamilien interessiert man sich für minimale Schnitte, um wertvolle Informationen über die Struktur der Graphen zu erhalten. Deshalb ist es wichtig das Problem der minimalen Schnitte auf dynamische Graphen zu übertragen. Im Wesentlichen kann man hier zwei verschiedene Ansätze verfolgen, um das Problem für Graphfamilien zu definieren.

Die erste Möglichkeit ist einen minimalen Schnitt zu jedem Zeitpunkt, also zu jedem Graphen der Familie, zu berechnen. Solche Schnitte weisen vermutlich noch eine große Ähnlichkeit zu den direkt folgenden und vorhergehenden Graphen auf; im Allgemeinen können minimale Schnitte von zwei Graphen zu unterschiedlichen Zeitpunkten aber sehr unterschiedlich aussehen. Minimale Schnitte sind in diesem Fall also auf einen bestimmten Zeitpunkt beschränkt.

Das führt zu einer zweiten Fragestellung, nämlich der Frage nach simultanen minimalen Schnitten, die stabil über die Zeit sind. Da man im allgemeinen Fall davon ausgehen muss, dass es keinen Schnitt gibt der zu allen Zeitpunkten minimal ist, muss das Problem relaxiert werden; man hat die Wahl entweder an Stelle der Gleichheit nur eine Ähnlichkeit

der Schnitte zu fordern oder statt minimalen Schnitten nur kleine Schnitte zu verlangen. Mit diesen beiden Möglichkeiten befasst sich diese Arbeit. Als Anwendungsbeispiel dieses Problems wird die Berechnung einer zeitstabilen Clusterung einer Graphfamilie auf Grundlage simultaner kleiner Schnitte betrachtet.

Im folgenden Abschnitt werden wichtige Definitionen und Notationen eingeführt. In Abschnitt 2 werden die zwei Problemvarianten formal definiert. Mit verwandten Arbeiten auf dem Gebiet der minimalen Schnitte befasst sich Abschnitt 3. Im vierten Abschnitt wird schließlich ein Überblick über den Aufbau der Kapitel in dieser Arbeit gegeben.

1.1 Definitionen und Notationen

Ein *Graph* ist ein Tupel $G = (V, E)$ mit Knotenmenge V und Kantenmenge $E \subseteq \binom{V}{2}$, das heißt der Graph ist ungerichtet und enthält weder Schlingen noch Mehrfachkanten. Es gelten die üblichen Notationen $n = |V|$, $m = |E|$. Im Kontext mehrerer Graphen werden die Knoten- und Kantemenge von G mit $V(G)$ bzw. $E(G)$ bezeichnet. Soll der Graph gewichtet sein, so wird den Kanten durch eine Gewichtsfunktion $c : E \to \mathbb{R}_+$ eine positive reelle Zahl zugeordnet. Wird keine Gewichtsfunktion explizit angegeben, so wird $c(e) = 1$ für alle $e \in E$ angenommen.

Als eine *Familie von Graphen* $(G_i)_{1 \leq i \leq k}$ wird eine Folge von k Graphen mit gleicher Knotenmenge V bezeichnet. Es gilt $G_i = (V, E_i)$ und $c_i : E_i \to \mathbb{R}_+$.

Ein *Schnitt* in einem Graphen $G = (V, E)$ ist eine ungeordnete Partition $(S, V \setminus S)$ der Knotenmenge V, wobei S eine echte nichtleere Teilmenge von V ist. Ein Schnitt $(S, V \setminus S)$ wird kurz mit S bezeichnet und mit seinem Komplement $V \setminus S$ identifiziert. Man sagt, ein Schnitt S *trennt* zwei Knoten $u, v \in V$, falls $|\{u, v\} \cap S| = 1$ gilt. Schnitte, die zwei vorgegebene Knoten $s, t \in V$ trennen heißen s-t-Schnitte. Die Menge der Schnittkanten in einem Graphen G bezüglich eines Schnitts S wird mit $E(S, G) = \{e \in E(G) \mid |e \cap S| = 1\}$ bezeichnet, kurz $E(S)$, wenn der Kontext klar ist. Für Teilmengen $V_1, V_2 \subseteq V$ bezeichne $E(V_1, V_2) = \{e \in E \mid e \cap V_1 \neq \emptyset, e \cap V_2 \neq \emptyset\}$ die Menge aller Kanten die einen Endpunkt in V_1 und einen Endpunkt in V_2 haben. Die *Größe eines Schnitts* $c_G(S)$ (oder kurz $c(S)$) ist die Summe über die Kantengewichte der Schnittkanten, das heißt $c(S) = \sum_{e \in E(S)} c(e)$. Analog ist $c(V_1, V_2)$ definiert als $c(V_1, V_2) = c(E(V_1, V_2))$.

Der *Kantenzusammenhang* $\lambda(G)$ (kurz λ) eines Graphen G ist definiert als die Größe des kleinstmöglichen Schnitts in G, also $\lambda(G) = \min_{\emptyset \neq S \subsetneq V(G)} c(S)$. Mit $\lambda(s, t, G)$ wird die Größe eines minimalen s-t-Schnitts in einem Graphen G bezeichnet; das heißt es gilt $\lambda(s, t, G) = \min_{\{s\} \subset S \subset V(G) \setminus \{t\}} c(S)$.

Ein *minimaler Schnitt* ist ein Schnitt S mit $c(S) = \lambda$. Die Menge der minimalen Schnitte, die s und t trennen wird mit $\mathcal{C}(s, t, G)$ bezeichnet; die Menge ist leer, wenn alle s-t-Schnitte nicht minimal sind. Die Menge aller minimalen Schnitte in einem Graphen G wird als $\mathcal{C}(G) = \{S \subset V \mid \emptyset \neq S \neq V, c(S) = \lambda\}$ notiert (kurz $\mathcal{C}(s, t)$ bzw. $\mathcal{C}$).

1.2 Schnitte in Familien von Graphen

Bei der Formalisierung der Problemstellung der simultanen Berechnung minimaler Schnitte haben sich zwei wichtige Varianten herauskristallisiert. Die erste sucht in jedem Graphen einen minimalen Schnitt, so dass sich die minimalen Schnitte möglichst ähnlich sind. Die zweite Variante fordert dagegen einen Schnitt, der in allen Graphen gleich ist; an Stelle der Minimalitätsforderung tritt nun die Bedingung, dass dieser Schnitt möglichst klein sein soll. Diese beiden Problemstellungen sollen nun definiert werden.

Vergleich minimaler Schnitte. Das simultane Schnittproblem für minimale Schnitte ist wie folgt definiert:

Problem 1. Gegeben ist eine Familie von k ungerichteten und gewichteten Graphen $(G_i)_{1\leq i\leq k}$ mit $G_i = (V, E_i)$ und $c_i : E_i \to \mathbb{R}_+$.

Gesucht sind minimale Schnitte $(S_1, \ldots, S_k) \in \mathcal{C}(G_1) \times \cdots \times \mathcal{C}(G_k)$ die sich möglichst ähnlich sind, das heißt eine Bewertungfunktion $f : \mathcal{C}(G_1) \times \cdots \times \mathcal{C}(G_k) \to \mathbb{R}$ für die Ähnlichkeit von Schnitten soll minimiert werden.

Verschiedene Ähnlichkeitsmaße werden in Kapitel 2 untersucht und damit die Problemstellung vervollständigt.

Vergleich kleiner Schnitte. Zur Definition der zweiten Problemvariante muss zunächst definiert werden, was ein *kleiner* Schnitt ist. Als Bezugsgröße hierfür bietet sich der Kantenzusammenhang $\lambda(G_i)$ der einzelnen Graphen an. Nun gibt es die Möglichkeit die absolute oder relative Differenz zwischen $c(S, G_i)$ und $\lambda(G_i)$ zu minimieren. Letzteres ist wohl die bessere Wahl, da hier dem jeweiligen Kantenzusammenhang besser Rechnung getragen wird. Damit erhalten wir folgende Problemstellung

Problem 2. Gegeben ist eine Familie von k zusammenhängenden ungerichteten und gewichteten Graphen $(G_i)_{1\leq i\leq k}$, mit $G_i = (V, E_i)$ und $c_i : E_i \to \mathbb{R}_+$.

Gesucht ist *ein* Schnitt S, der folgenden Ausdruck minimiert:

$$\min_{\emptyset \neq S \subsetneq V} \max_{i=1,\ldots,k} \frac{c(S, G_i)}{\lambda(G_i)}$$

1.3 Verwandte Arbeiten

In diesem Abschnitt werden zunächst Arbeiten zur Berechnung von minimalen s-t-Schnitten vorgestellt. Aufbauend darauf folgen Verfahren zur Bestimmung globaler minimaler Schnitte und ihrer Repräsentation. Schließlich werden Algorithmen umrissen, die kleine Schnitte berechnen.

1.3.1 Minimale s-t-Schnitte

In den vergangenen Jahrzehnten wurden sehr viele Algorithmen zur Berechnung von minimalen s-t-Schnitten entwickelt. Viele davon bauen aufeinander auf, einige sind nur für bestimmte Graphklassen geeignet, wie planare Graphen, ungewichtete, ganzzahlig gewichtete oder ungerichtete Graphen. An dieser Stelle sollen wichtige Algorithmen und die wesentliche Ideen, die zur Weiterentwicklung beigetragen haben, skizziert werden, mit besonderem Augenmerk auf die Algorithmen, die auf ungerichtete reell-gewichtete Graphen anwendbar sind.

Grundlegende Sätze. Menger formuliert 1927 den nach ihm benannten Satz, der besagt, dass die Zahl der kantendisjunkten Pfade zwischen zwei Knoten s und t der Größe eines minimalen s-t-Schnitts entspricht [Men27]. Ford und Fulkerson sowie unabhängig Elias et al. verallgemeinern den Satz zum Max-Flow-Min-Cut-Theorem; sie zeigen, dass die maximale Größe eines s-t-Flusses gleich der Größe eines minimalen s-t-Schnitts ist [FF56, EFS56].

Methode der erhöhenden Pfade. Ford und Fulkerson veröffentlichen zusammen mit dem Max-Flow-Min-Cut-Theorem einen Algorithmus, der in $O(m\lambda)$ Schritten maximale Flüsse berechnet, indem er erhöhende Pfade findet [FF56]. Edmonds und Karp verbessern den Algorithmus auf eine Laufzeit in $O(nm^2)$, indem sie kürzeste erhöhende Pfade suchen [EK72]. Dinic bringt die Idee der blockierenden Flüsse ein und kann damit die Laufzeit auf $O(n^2m)$ senken [Din70]. Auf seinem Algorithmus basieren einige weitere Algorithmen, unter anderem der von Karzanov, der als erster mit Präflüssen arbeitet und die Laufzeit auf $O(n^3)$ verbessert [Kar74].

Push-Relabel-Methode. Goldberg greift die Idee der Präflüsse auf, verfolgt aber den neuen Push-Relabel-Ansatz; statt erhöhenden Pfaden werden den Knoten Distanz-Labels zugeordnet, mit deren Hilfe die Überschüsse eines Präfluss in Richtung Senke gepushed werden können [Gol85]. Zusammen mit Tarjan und seiner Technik der dynamischen Bäume [ST85] kann der Algorithmus auf $O(nm \log(n^2/m))$ verbessert werden [GT88].

Aufzählung von s-t-Schnitten. Picard und Queyranne konstruieren aus dem Residualgraphen, einem Graphen dessen Kantengewichte den noch freien Kapazitäten entsprechen, einen gerichteten azyklischen Graphen, indem sie starke Zusammenhangskomponenten kontrahieren. Mit Hilfe dieses Graphen lassen sich alle s-t-Schnitte aufzählen. [PQ80].

1.3.2 Globale minimale Schnitte

In einem Graph gibt es maximal $\binom{n}{2}$ minimale Schnitte [Bix75, DKL76]. Um diese zu bestimmen gibt es neben der Verwendung von Algorithmen für s-t-Schnitte seit den 90er-Jahren auch Algorithmen, die nicht auf Flussberechnungen basieren.

Iterative Berechnung von s-t-Schnitten. Der Standardalgorithmus zur Berechnung globaler minimaler Schnitte ist lange Zeit die Berechnung von $n-1$ verschiedenen s-t-Schnitten; man hält dazu einen beliebigen Knoten $s \in V$ fest und iteriert über die übrigen Knoten $t \in V \setminus \{s\}$ [GH61]. Die Zeitkomplexität dieses Algorithmus liegt damit in $O(n \cdot \text{maxflow}(m,n))$ und verbessert sich zunächst nur analog zur Entwicklung der s-t-Schnitt-Algorithmen.

Padberg und Rinaldi haben Bedingungen formuliert, unter denen Kanten kontrahiert werden können, ohne die Größe des minimalen Schnitts zu verändern, was in vielen Fällen zu einer deutlichen Verbesserung der Laufzeit führt, asymptotisch aber keine Verbesserung darstellt [PR90].

Hao und Orlin senken die Zeitkomplexität zur Berechnung von $n-1$ s-t-Schnitten auf die Zeit zur Berechnung eines einzigen s-t-Schnitts, indem sie die Information der berechneten Distanz-Labels aus einer Berechnung in den weiteren Iterationen wiederverwenden [HO92].

Ein neuer Ansatz Nagamochi und Ibaraki entwickeln den ersten Algorithmus zur Berechnung der Größe minimaler Schnitte, der nicht auf Flussberechnungen basiert und in $O(n(m + n \log n))$ läuft [NI92a, NI92b]; er wird von Stoer und Wagner sowie unabhängig davon von Frank stark vereinfacht [SW97, Fra94]. Der Algorithmus berechnet in $n-1$ Phasen jeweils die Größe eines minimalen s-t-Schnitts und kontrahiert s und t. Nagamochi und Ibaraki verbessern den Algorithmus, indem in jeder Phase auch mehrere Kanten kontrahiert werden können [NOI94]. Arikati und Mehlhorn erweitern den Algorithmus so, dass nicht nur die Größe der jeweiligen s-t-Schnitte, sondern auch ein maximaler s-t-Fluss berechnet wird [AM99].

Auch randomisierte Algorithmen sind bekannt, die in $O(n^2 \log n)$ Schritten mit hoher Wahrscheinlichkeit einen minimalen Schnitt finden [KS96, Kar00].

1.3.3 Repräsentationen von Schnitten

Gomory und Hu haben einen Algorithmus zur Konstruktion eines Schnitt-Baums angegeben, der für jedes s-t-Paar *einen* minimalen s-t-Schnitt repräsentiert [GH61]. Der Schnittbaum eines Graphen hat einen Speicherbedarf von $O(n)$ und kann mit einer Zeitkomplexität in $O(n \cdot \text{maxflow}(m,n))$ berechnet werden.

Kaktus-Repräsentation. Dinic, Karzanov und Lomonosov haben eine Datenstruktur entdeckt, die alle minimalen Schnitte eines Graphen repräsentieren kann [DKL76]. Dabei handelt es sich um einen Kaktus, das heißt einen zusammenhängenden ungewichteten Graph, in dem jede Kante zu genau einem Kreis gehört; zusammen mit einer Abbildung der Graphknoten auf die Kaktusknoten repräsentieren alle minimalen Schnitte im Kaktus genau alle minimalen Schnitten im Graphen.

Karzanov und Timofeev [KT86] haben den ersten Algorithmus zur Konstruktion eines solchen Kaktus formuliert, der aus den berechneten minimalen Schnitten in $O(n^2)$ einen

Kaktus konstruiert. Fleischer konnte die Laufzeit für die Konstruktion eines Kaktus durch die Kombination des Karzanov-Timofeev-Frameworks mit dem Hao-Orlin-Algorithmus auf $O(mn \log(n^2/m))$ verbessern [Fle99]. Nagamochi et al. entwickelten einen Algorithmus mit Laufzeit $O(mn + n^2 \log n)$ der auf dem Stoer-Wagner-Algorithmus basiert [NNI03].

1.3.4 Kleine Schnitte

Es gibt verschiedene Schranken für die Anzahl kleiner Schnitte, das heißt Schnitte S mit $c(S) < (1+\varepsilon)\lambda$. Für $\varepsilon < 1/2$ gilt die obere scharfe Schranke von $O(n^2)$ [NNI97, HW96]; für den allgemeinen Fall haben Karger und Stein gezeigt, dass es höchstens $n^{2(1+\varepsilon)}$ kleine Schnitte gibt [KS96]. Um kleine Schnitte zu finden gibt es zwei verschiedene Ansätze; die erste Möglichkeit besteht darin sukzessive solange weitere Schnitte aufzuzählen bis die Berechnung abgebrochen werden kann während die zweite Methode alle Schnitte, die eine bestimmte Größe nicht überschreiten berechnet.

Algorithmus mit polynomieller Verzögerung. Hamacher veröffentlichte einen Algorithmus der s-t-Schnitte eines Graphen in aufsteigender Größe aufzählt [Ham82, HPQ84]. Der Algorithmus gibt in jedem Schritt den jeweils kleinsten bisher gefundenen Schnitt aus und berechnet bis zu $n-1$ weitere s-t-Schnitte. Vazirani und Yannakakis beschreiben einen Algorithmus, der globale minimale Schnitte in aufsteigender Größe aufzählt [VY92]. Ähnlich wie bei Hamachers Algorithmus werden bis zu $n-1$ weitere s-t-Schnitte zwischen zwei Ausgaben berechnet. Dabei kann garantiert werden, dass jeweils der kleinste noch nicht ausgegebene Schnitt bereits berechnet wurde. Die Laufzeit liegt demnach in $O(rn \cdot \text{maxflow}(m,n))$ für die r kleinsten Schnitte eines Graphen. Yeh et al. konnten auf Grundlage des Algorithmus von Vazirani und Yannakakis in Verbindung mit dem Hao-Orlin-Algorithmus die Zeit auf $O(r \cdot \text{maxflow}(m,n))$ reduzieren [YW08].

Algorithmus für alle Schnitte bis zu einer vorbestimmten Größe. Einen weiteren Algorithmus haben Nagamochi, Nishimura und Ibaraki entwickelt [NNI97]. Mit Hilfe von Splitting-Operation können Kantengewichte so verändert werden, dass die Größe fast aller Schnitte erhalten bleibt, während alle Kanten die zu einem bestimmten Knoten s inzident sind, sowie der Knoten s selbst, entfernt werden können. Wiederholt man diesen Vorgang bis nur noch ein Knoten übrig bleibt, so kann man aus den Split-Operationen alle Schnitte S mit $c(S) < (1+\varepsilon)\lambda$ berechnen. Die Laufzeit für $\varepsilon > 0$ beträgt $O(m^2 n + n^{2(1+\varepsilon)} m)$.

1.4 Überblick

Im Folgenden wird ein Überblick über die übrigen Kapitel dieser Arbeit gegeben.

Kapitel 2: Minimale Schnitte. Das zweite Kapitel beschäftigt sich mit der Problemstellung der minimalen Schnitte. Es werden Ähnlichkeitsmaße diskutiert und die Kaktus-Repräsentation für minimale Schnitte vorgestellt. Als Neuerung wird die dynamische Kaktus-Repräsentation eingeführt; dazu werden Operationen definiert, die die Repräsentation der minimalen Schnitte eines Graphen G aktualisieren, wenn in G eine neue Kante eingefügt oder eine bestehende Kante entfernt wird. Auf Grundlage dessen wird ein Algorithmus zur effizienten Berechnung von Kaktus-Repräsentationen für Familien von Graphen vorgestellt. Schließlich wird gezeigt, wie die minimalen Schnitte miteinander verglichen werden können, um das Problem optimal oder heuristisch lösen zu können.

Kapitel 3: Kleine Schnitte. Im drittel Kapitel wird die Frage nach kleinen Schnitten behandelt. Bekannte Algorithmen für die Berechnung kleiner Schnitte werden erläutert; darauf aufbauend werden Algorithmen entwickelt, die kleine Schnitte in Familien von Graphen aufzählen. Auch eine Heuristik zur Lösung des Problems wird präsentiert. Schließlich wird gezeigt, wie sich aus kleinen Schnitten in Graphfamilien eine gemeinsame Clusterung berechnen lässt.

Kapitel 4: Erzeugung von Graphen mit vorgegebener Schnittstruktur. Das vierte Kapitel beschäftigt sich mit der Frage, wie ein Graph mit einer vorgegebenen minimalen Schnittstruktur erzeugt werden kann, um geeignete Graphen insbesondere zur Lösung des Problems der minimalen Schnitte zu finden. Es wird ein vollständiger Graphgenerator vorgestellt, der Graphen mit beliebiger minimaler Schnittstruktur erzeugen kann.

Kapitel 5: Experimentelle Auswertung. Kapitel 5 stellt die verwendeten Testgraphen vor, die als Eingabe für die implementierten Algorithmen dienten. Die Laufzeitmessungen und Ergebnisse der Berechnungen für beide Problemvarianten und das Clusterungsverfahren werden analysiert und mit den Resultaten aus dem theoretischen Teil verglichen.

Kapitel 6: Zusammenfassung. Schließlich werden im sechsten Kapitel die wichtigsten Ergebnisse noch einmal zusammengefasst und ein Ausblick auf weiterführende Fragestellungen gegeben..

2

Minimale Schnitte

Dieses Kapitel befasst sich mit der ersten Problemvariante. In einer Familie von Graphen werden also minimale Schnitte gesucht, die sich möglichst ähnlich sind. Das Problem wurde im ersten Kapitel im Abschnitt 1.2 bis auf die Ähnlichkeitsmaße definiert. Diese sind Gegenstand des ersten Abschnitts. Der zweite Abschnitt stellt die verwendeten Algorithmen zur Berechnung von minimalen Schnitten vor und untersucht die Veränderungen von Schnitten, wenn Kanten hinzugefügt oder entfernt werden. Im dritten Abschnitt wird der Kaktus als Repräsentation minimaler Schnitt vorgestellt und herausgearbeitet, wie der Kaktus nach dem Einfügen oder Löschen von Kanten aktualisiert werden kann. Schließlich werden im vierten Abschnitt Methoden zum Vergleich der gefundenen Schnitte untersucht.

2.1 Ähnlichkeitsmaße

Es soll nun die Frage diskutiert werden, wann sich zwei oder mehr Schnitte in Graphen einer Familie ähnlich sind. Die Ähnlichkeit soll durch eine Bewertungsfunktion $f : \mathcal{C}(G_1) \times \cdots \times \mathcal{C}(G_k) \to \mathbb{R}$ ausgedrückt werden. Dabei können zum einen die unterschiedlichen Knoten eines Schnitts S und zum anderen damit verbundene Kanten, zum Beispiel die Schnittkanten $E(S)$ betrachtet werden. Was sinnvoller ist, hängt vom jeweiligen Anwendungsfall ab. Ein Vergleich der Kanten in mehreren Graphen bietet sich nur an, wenn sich die Kanten in den Graphen auch ähnlich sind und wenn es für das vorliegende Problem wichtig ist, dass möglichst viele Kanten übereinstimmen. Geht es nur darum die Knotenmenge zu teilen, so dass wenige Kanten zwischen ihnen liegen, so wird man eher die Knotenmengen betrachten.

Intuitiv kann man sagen, dass es möglichst wenige Knoten geben soll, die in manchen Schnitten, aber nicht in allen Schnitten vorkommen. Das Problem hierbei ist, dass ein Schnitt S mit seinem Komplement $V \setminus S$ identifiziert wird und es daher sinnvoll ist von der Bewertungfunktion zu verlangen, dass sie symmetrisch ist, das heißt es soll

$$\forall i \in \{1, \dots k\} : f(S_1, \dots, S_i, \dots S_k) = f(S_1, \dots, V \setminus S_i, \dots S_k) \tag{2.1}$$

gelten. Die naive Formulierung des ersten Ansatzes, $f_1'(S_1, \dots, S_k) = |\bigcup S_i \setminus \bigcap S_i|$ zu minimieren, erfüllt die Forderung allerdings nicht. Man kann diese zwar symmetrisch machen, indem man f_1 als

$$f_1(S_1, \dots, S_k) = \min_{(S_1', \dots, S_k') \in \{S_1, V \setminus S_1\} \times \dots \times \{S_k, V \setminus S_k\}} \left| \bigcup_{i=1}^{k} S_i' \setminus \bigcap_{i=1}^{k} S_i' \right| \tag{2.2}$$

definiert, allerdings gibt es so um den Faktor 2^k mehr Möglichkeiten Schnitte aus k Graphen zu kombinieren; selbst wenn es nur einen minimalen Schnitt in jedem Graphen gibt sind 2^k Berechnungen von f_1' zur Bestimmung von f_1 erforderlich.

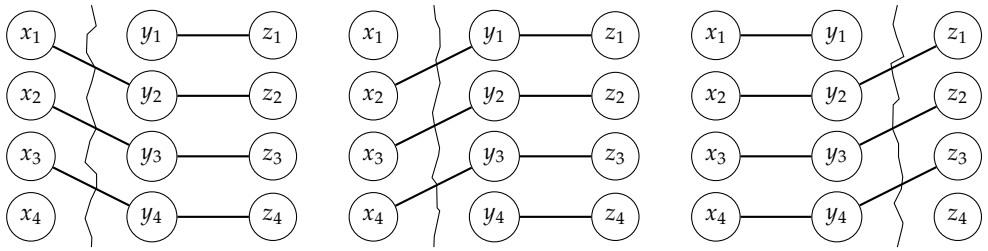
Die Forderung der Symmetrieeigenschaft (2.1) führt zur Überlegung statt der Knotenmenge S eines Schnitts die Schnittkanten $E(S)$ zu betrachten, da $E(S) = E(V \setminus S)$ gilt. Dabei bietet es sich an nicht nur die Anzahl der Kanten zu betrachten, sondern auch die Gewichte der Kanten zu nutzen. Analog zum vorherigen Vorschlag ergibt sich als Minimierungsfunktion daraus

$$f_2(S_1, \dots, S_k) = c(\bigcup_{i=1}^{k} E(S_i) \setminus \bigcap_{i=1}^{k} E(S_i)), \tag{2.3}$$

Problematisch ist hier allerdings gegenüber dem Vergleich von Knotenmengen, dass die Kantenmengen in den verschiedenen Graphen im Allgemeinen nicht gleich sind; es kann also Kanten geben, die nicht in allen Graphen vorkommen. Die Bewertungfunktion f_2 würde das Nicht-Enthaltensein solcher Kanten bestrafen; im Beispiel des Vergleichs der Schnitte von G_1 und G_2 in Abbildung 2.1 wäre das eine ungünstige Eigenschaft, die der Intuition widerspricht, da f_2 die Ähnlichkeit der Schnitte mit $f_2(S_1, S_2) = 6$ bewerten würde, obwohl die Knotenmengen gleich sind.

Löst man das Problem, indem man die Kanten, die nicht in allen Graphen vorkommen immer zu $\bigcap E(S_i)$ rechnet, so erhöhen diese Kanten den Funktionswert nicht mehr. Das wäre aber im Fall der Graphen G_1 und G_3 in Abbildung 2.1 ungünstig, denn hier unterscheiden sich die Schnitte sowohl in der Kanten-, als auch in der Knotenmenge, f_2 würde den Vergleich aber mit 0 bewerten. Somit stellt auch die modifizierte Version von f_2 kein gutes Ähnlichkeitsmaß dar.

Da die Gleichheit von Knoten in Schnitten wichtig erscheint, führt die folgende Überlegung wieder auf den Vergleich von Knotenmengen: Man betrachtet zu einem Schnitt S die zu den Schnittkanten inzidente Knotenmenge $\bigcup_{\{u,v\} \in E(S)} \{u, v\}$. Wiederum vergleicht



(a) Graph G_1 (b) Graph G_2 (c) Graph G_3

Abbildung 2.1: Drei Teilgraphen und zugehörigen Schnitten: Die Schnitte von G_1 und G_2 trennen die gleichen Knoten, schneiden aber unterschiedliche Kanten.

	S_1	S_2	S_3	S_1, S_2	S_1, S_3	S_2, S_3	S_1, S_2, S_3	Zeitkomplexität
f_1	0	0	0	0	4	4	4	$O(2^k nk)$
f_2	0	0	0	6	6	6	9	$O(km)$
f_3	0	0	0	4	6	8	9	$O(kn)$
f_4	0	0	0	0	32	32	32	$O(kn^2)$

Tabelle 2.1: Die Bewertungfunktionen f_1, f_2, f_3 und f_4 angewendet auf das Beispiel in Abbildung 2.1 im Vergleich.

man diese Mengen eines jeden Graphen miteinander, indem man die Differenz zwischen Vereinigung und Schnittmenge betrachtet:

$$\begin{aligned} f_3(S_1, \ldots, S_k) &= \left| \left(\bigcup_{i=1}^{k} f_3'(S_i; G_i) \right) \setminus \left(\bigcap_{i=1}^{k} f_3'(S_i; G_i) \right) \right| \\ f_3'(S; G) &= \bigcup_{e \in E(S;G)} e \end{aligned} \tag{2.4}$$

Die Funktion f_3 berücksichtigt sowohl Knoten als auch Kanten und ist somit ein Kompromiss zwischen f_1 und f_2. Da Knoten über unterschiedliche Schnittkanten in die Vereinigung und den Schnitt von f_3 eingehen können, kann das Fehlen einer Kante in einem Graphen durch andere Kanten kompensiert werden.

Sowohl f_2 als auch f_3 haben den Nachteil, dass zwei völlig unterschiedliche Schnitte, die keine Kanten oder zu Schnittkanten inzidente Knoten gemeinsam haben immer gleich schlecht bewertet werden. Abbildung 2.2 illustriert diese Eigenschaft: Es gilt jeweils $f_i(S_1, S_2) = f_i(S_1, S_3)$ für $i \in \{2, 3\}$. Von einer guten Bewertungfunktion würde man aber erwarten, dass die näher beieinanderliegenden Schnitte (im Beispiel S_1 und S_2) besser bewertet werden.

Das führt zu einer letzten Bewertungfunktion, die nicht nur Teile der Knoten bzw. Kantenmenge betrachtet, sondern die Beziehungen zwischen allen Knoten berücksichtigt. Dabei wird für jedes Knotenpaar gezählt, von wie vielen Schnitten es getrennt wird.

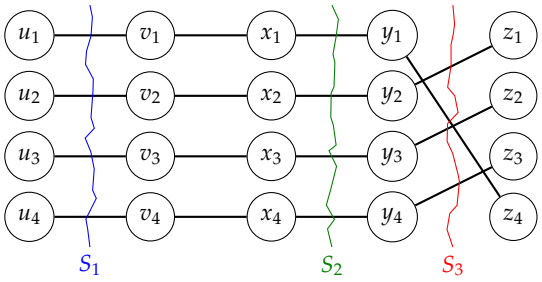


Abbildung 2.2: Weit auseinanderligende Schnitte in einem Graphen.

Schnitte sind sich dann ähnlich, wenn die Knotenpaare jeweils entweder von (fast) allen Schnitten getrennt wurden oder von (fast) keinem Schnitt. Daher wird für k Schnitte angenommen, dass sie sich ähnlicher sind, je näher für die Knotenpaare die Anzahl der Trennungen entweder an 0 oder an k ist. Besonders schlecht ist es, wenn die Anzahl der Trennungen eines Knotenpaares $k/2$ entspricht, da hier die Entfernung zu 0 und k maximiert wird. Damit kann die Funktion formal definiert werden als

$$f_4(S_1, \ldots, S_k) = \sum_{\{u,v\} \subset V} \min\{\delta(S_1, \ldots, S_k, u, v), k - \delta(S_1, \ldots, S_k, u, v)\}, \tag{2.5}$$
$$\text{wobei } \delta(S_1, \ldots, S_k, u, v) = |\{S_i \mid S_i \text{ trennt } u \text{ und } v\}|$$

Ein Schnitt S_i, der genau die Knotenpaare trennt, die von den meisten anderen Schnitten $S_1, \ldots, S_{i-1}$ auch getrennt werden, erhöht f_4 nicht, das heißt es gilt dann $f_4(S_1, \ldots, S_i) = f_4(S_1, \ldots, S_{i-1})$, denn $\min\{\delta(S_1, \ldots, S_k, u, v), k - \delta(S_1, \ldots, S_k, u, v)\}$ erhöht sich dadurch nicht. So einen Schnitt muss es allerdings nicht geben, da die Knotentrennungen nicht transitiv sein müssen. Außerdem hat f_4 die Eigenschaft, dass sie bei Hinzunahme eines weiteren Schnitts nur Größer werden kann, da die Differenz zu 0 bzw zu k nicht verringert werden kann, sondern entweder gleich bleibt oder größer wird. Für die Teilgraphen in Abbildung 2.1 gilt zum Beispiel $f_4(S_1, S_2) = 0$ und $f_4(S_1, S_3) = f_5(S_2, S_3) = f_5(S_1, S_2, S_3) = 32$ (siehe auch Tabelle 2.1). Die Bewertungsfunktion f_4 bestimmt die Ähnlichkeit von Schnitten nur auf Grundlage der Schnittknoten; Schnittkanten und damit auch die Kantengewichte werden hier nicht berücksichtigt. Die Zeitkomplexität der Berechnung für jeden Schnitt liegt in $O(n^2)$ da $\binom{n}{2}$ Knotenpaare geprüft werden müssen. Das Berechnen von f_4 mit k Schnitten liegt daher in $O(k \cdot n^2)$.

Betrachtet man die Ähnlichkeitsmaße für die Graphen aus Abbildung 2.1 in Tabelle 2.1 so kommen f_1 und f_4 der intuitiven Vorstellung am nächsten. Aufgrund der einfacheren Berechnung von f_4 wird diese im Weiteren als Ähnlichkeitsmaß für den Vergleich der minimalen Schnitte verwendet.

2.2 Minimale Schnitte und ihre Berechnung

In diesem Abschnitt werden zunächst verschiedene Algorithmen und Beschleunigungstechniken zur Berechnung von s-t-Schnitten und globalen minimalen Schnitten vorge-

stellt. Es folgt ein kurzer Überblick über wichtige Eigenschaften minimaler Schnitte. Schließlich wird der Fall simultaner minimaler Schnitte besprochen, das heißt es wird untersucht, wie sich die Menge minimaler Schnitt nach Kantenoperationen verändert.

2.2.1 Minimale *s*-*t*-Schnitte

In der Einleitung wurde bereits skizziert, dass die bekannten Algorithmen zur Berechnung von minimalen *s*-*t*-Schnitten entweder die Technik der erhöhenden Pfade oder die Push-Relabel-Methode verwenden. Für den praktischen Teil der Arbeit wurde die Push-Relabel-Methode verwendet. Die Erläuterung der Funktionsweise erfordert noch die Definition einiger Begriffe.

Flüsse in Netzwerken. Ein *Netzwerk* $N = (V, E, c, s, t)$ ist ein gerichteter Graph mit Knotenmenge V, Kantenmenge $E \subseteq V \times V$, Kapazitätsfunktion c und zwei ausgezeichneten Knoten s und t, die als *Quelle* bzw. *Senke* bezeichnet werden. Ein *Fluss* in einem Netzwerk N ist eine Funktion $f : E \to \mathbb{R}_+$, die jeder Kante einen Fluss zuweist, der über sie führt. Dabei müssen folgende Bedingungen eingehalten werden.

a) Die *Kapazitätsbedingung* fordert, dass über eine Kante kein Fluss transportiert werden kann, der ihre Kapazität übersteigt: $\forall e \in E : f(e) \leq c(e)$.

b) Die *Flusserhaltung* verlangt, dass für alle Knoten außer s und t genauso viel ankommt, wie abfließt: $\forall v \in V \setminus \{s, t\} : \sum_{e=(u,v)\in E} f(e) = \sum_{e=(v,u)\in E} f(e)$. Für s und t gilt $\sum_{e=(u,s)\in E} f(e) \leq \sum_{e=(s,u)\in E} f(e)$ bzw. $\sum_{e=(u,t)\in E} f(e) \geq \sum_{e=(t,u)\in E} f(e)$.

Die *Größe des Flusses* $val(f)$ entspricht dem Fluss, der von der Quelle s ausgeht und an der Senke t ankommt $val(f) = \sum_{e=(s,u)\in E} f(e) = \sum_{e=(u,t)\in E} f(e)$. Die *Residualkapazität* der Kanten eines Netzwerks bezüglich eines Flusses f ist definiert als $c_f(e) = c(e) - f(e)$. Ein *Präfluss* erfüllt die Kapazitätsbedingung, verletzt möglicherweise aber die Flusserhaltung in dem Sinn, dass mehr Fluss an einem Knoten ankommt, als abfließt, das heißt für einen Knoten v ist ein *Überschuss* $p(v) = \sum_{e=(u,v)\in E} f(e) - \sum_{e=(v,u)\in E} f(e) \geq 0$ erlaubt. Die *Größe eines Präflusses* ist der Fluss, der bei der Senke t ankommt: $val(f) = \sum_{e=(u,t)\in E} f(e)$. Aus jedem Präfluss lässt sich immer ein Fluss konstruieren, der die gleiche Größe hat.

Push-Relabel-Technik. Die Idee des Algorithmus besteht darin einen Präfluss zu erzeugen, indem zunächst von der Quelle aus so viel Fluss, wie über die inzidenten Kanten möglich ist, zu den Nachbarknoten gepushed wird. Der so entstandene Überschuss wird dann versucht in Richtung Senke t weiterzuschieben. Falls es nicht möglich ist allen Überschuss zur Senke zu transportieren, wird er zurück zur Quelle geschoben. Die Richtung, in der der Überschuss gepushed werden soll, ergibt sich aus Distanzlabeln $dist : V \to \mathbb{N}$. Es werden *aktive* Knoten, die einen Überschuss haben von nicht-aktiven unterschieden. Solange es aktive Knoten u gibt, wird, wenn möglich, eine *Push*-Operation, sonst eine *Relabel*-Operation auf u ausgeführt. Die Push-Operation kann nur angewendet werden, wenn u einen Nachbarknoten v hat mit $c_f(u, v) > 0$ und $dist(u) = dist(v) + 1$. Sie leitet so viel Überschuss wie möglich an v weiter. Wird die Relabel-Operation angewandt, so

wird $dist(v) = min_{(u,v)\in E, c_f(u,v)>0} dist(v) + 1$ gesetzt, womit wieder eine Push-Operation möglich ist. Die Laufzeit dieses Algorithmus liegt in $O(m^2 n)$ [Gol85].

Beschleunigungstechniken. Um die Laufzeit zu verbessern wurden einige Beschleunigungstechniken entwickelt. Es gibt im Wesentlichen zwei Stellen an denen diese Techniken ansetzen. Zum einen ist die Wahl des aktiven Knotens, auf den eine Operation ausgeführt wird, im Originalalgorithmus beliebig. Das kann ausgenutzt werden, indem der Algorithmus durch geschickte Auswahlkriterien beschleunigt wird. Zum anderen können die Distanzen durchdachter aktualisiert werden. Nicht alle Beschleunigungstechniken verbessern die asymptotische Laufzeit, führen aber dennoch zu deutlichen Laufzeitverbesserungen [CGK+97]. Die wichtigsten Beschleunigungstechniken sind:

a) Aufteilung des Algorithmus in zwei *Phasen*: In der ersten Phase wird zunächst ein maximaler Präfluss erzeugt. Knoten, die ihren Überfluss nicht mehr loswerden können, werden nicht mehr als aktiv bezeichnet. In einer zweiten Phase wird aus dem Präfluss dann ein Fluss konstruiert, indem zuerst zyklische Flüsse mittels Tiefensuche entfernt werden und die Überschüsse dann auf eindeutigen Wegen zurückgeschoben werden.

b) *Global Relabel* bezeichnet ein nach einer bestimmten Anzahl von Push- oder Relabel-Operationen wiederholtes aktualiseren aller Distanzlabel. Dabei wird von der Senke t aus eine Breitensuche gestartet, die allen Knoten die Entfernung von t zuweist [Che79].

c) Durch die *Gap-Heuristik* wird erkannt, wenn Lücken zwischen den Distanzlabeln entstehen, dass heißt es gibt Knoten mit Distanzen a und b, $a + 1 < b$ und es gibt keine Distanz c mit $a < c < b$. In diesem Fall kann ein Überschuss nicht mehr zur Senke transportiert werden; das Distanzlabel kann dann auf n gesetzt werden und der Überschuss geht zurück zur Quelle [DM89].

d) *Dynamic Trees* bezeichnet eine Datenstruktur, die aus einer Menge von Bäumen besteht, deren Kanten alle freie Kapazitäten haben, wobei nicht alle derartigen Kanten zu einem Baum gehören. Überschüsse können nun entlang von Baumkanten Richtung Senke weitergeschoben werden. Bäume können dabei dynamisch verschmolzen werden, andererseits fallen Kanten weg, wenn sie gesättigt werden. Dadurch verbessert sich die Laufzeit auf $O(mn \log(n^2/m))$ [GT88, ST85].

e) Die *FIFO-Strategie* hält die aktiven Knoten in einer Warteschlange; die Laufzeit reduziert sich damit auf $O(n^3)$. Das *Highest-Label-First*-Auswahlkriterium nutzt dagegen eine Vorrangwarteschlange, die jeweils den Knoten mit den höchsten Distanzlabel auswählt, womit eine Zeit von $O(n^2\sqrt{m})$ erreicht wird [GT88].

Mit diesen Techniken liegt die Laufzeit also in $O(mn \log(n^2/m))$. King, Rao und Tarjan haben einen Algorithmus veröffentlicht, der in $O(mn \log_{m/(n \log n)} n)$ läuft, damit ist er schneller, falls $m/n \in \omega(\log n)$ [KRT94].

Ein minimaler s-t-Schnitt in einem Graphen $G = (V, E)$ mit Kantengewichtsfunktion c lässt sich nun finden, indem aus G ein Netzwerk $N = (V, E', c', s, t)$ konstruiert wird. Für jede Kante in G enthält N zwei gerichtete Kanten, das heißt für die Kantemenge

in N gilt $E' = \{(u,v) \mid \{u,v\} \in E\}$ und $c'(u,v) = c(\{u,v\})$. Aus dem Max-Flow-Min-Cut-Theorem folgt dann, dass die Größe eines maximalen Flusses in N der Größe eines minimalen s-t-Schnitts in G entspricht [FF56, EFS56].

2.2.2 Globale minimale Schnitte

Die wohl einfachsten und schnellen bekannten Algorithmen zur Berechnung minimaler Schnitte beruhen auf einer bestimmten Anordnung der Knoten. Die Knoten werden dabei so geordnet, dass gilt

$$\forall i,j \in \{2,\dots n\}, i < j : c(\{v_1,\dots v_{i-1}\},\{v_i\}) \geq c(\{v_1,\dots v_{i-1}\},\{v_j\});$$

die Anordnung heißt *Maximum Adjacency Ordering* (MAO) [TY84]. Eine solche Ordnung kann in $O(m + n \log n)$ gefunden werden, indem beginnend mit einem beliebigen Knoten v_1 jeweils der Knoten v_i als nächstes ausgewählt wird, der zu den bisherigen Knoten am stärksten verbunden ist, der also $c(\{v_1,\dots v_{i-1}\},\{v_i\})$ maximiert. Diese Anordnung hat eine wichtige Eigenschaft, es gilt: $\lambda(v_{n-1},v_n) = c(\{v_n\})$; die beiden Knoten v_{n-1}, v_n nennt man auch *Pendent Pair*. Damit hat man mit der Berechnung einer MAO bereits einen s-t-Schnitt gefunden, allerdings ohne s und t festlegen zu können. Dies erkannten Nagamochi und Ibaraki bzw. in einer vereinfachten Form Stoer und Wagner und entwarfen einen Algorithmus zu Berechnung minimaler Schnitte [NI92a, NI92b, SW97]. Für den Kantenzusammenhang $\lambda(G)$ gilt dann $\lambda(G) = \min(c(\{v_n\}), \lambda(G'))$, wobei $G' = G/\{v_{n-1}, v_n\}$ den Graphen bezeichnet, den man durch die Kontraktion der Knoten v_{n-1}, v_n erhält. Der Kantenzusammenhang $\lambda(G)$ kann also durch $n-1$ MAO-Berechnungen und Kontraktionen bestimmt werden. Damit ergibt sich eine Laufzeit $O(mn + n^2 \log n)$. Mit dem Algorithmus von Arikati und Mehlhorn lässt sich aus einer MAO ein s-t-Fluss bestimmen [AM99].

2.2.3 Eigenschaften

Ein Paar minimaler Schnitte $S_1, S_2 \in \mathcal{C}(G)$ eines Graphen G heißt *kreuzend*, falls weder $T_1 = S_1 \cap S_2, T_2 = S_1 \setminus S_2, T_3 = S_2 \setminus S_1$ noch $T_4 = \overline{S_1} \cap \overline{S_2}$ leer sind. Für kreuzende Schnitte ist bekannt:

Lemma 2.1 (Kreuzende Schnitte (siehe bspw. [NI08]))**.** Seien S_1 und S_2 minimale kreuzende Schnitte in G und T_1, T_2, T_3 und T_4 wie oben definiert, dann sind auch T_1, T_2, T_3 und T_4 minimale Schnitte und es gilt $c(T_1,T_2) = c(T_1,T_3) = c(T_2,T_4) = c(T_3,T_4) = \lambda(G)/2$ und $c(T_1,T_4) = c(T_2,T_3) = 0$.

Zwei minimale Schnitte S_1, S_2, die sich nicht kreuzen sind zueinander *parallel*. Für sie gilt $\exists A \in \{S_1, \overline{S_1}\}, B \in \{S_2, \overline{S_2}\} : A \subset B$, das heißt S_1 und S_2 (oder ihr Komplement) enthalten sich gegenseitig. Eine Menge minimaler Schnitte $\{S_1,\dots S_k\} \subset \mathcal{C}(G)$ eines Graphen G heißt parallel, wenn ihre Schnitte paarweise parallel sind. Eine solche Menge paralleler Schnitte lässt sich durch eine *parallele Partition (minimaler Schnitte)* $(V_1,\dots,V_{k+1})$

der Knotenmenge V darstellen, für die gilt $S_i = \bigcup_{j=1}^{i} V_j$. Für eine parallele Partition $(V_1, \ldots, V_{k+1})$ wird gefordert, dass sie sich nicht erweitern lässt, das heißt, dass keine Partition $(V_1, \ldots, V_i', V_i'', \ldots V_{k+1})$ minimaler Schnitte existiert mit $V_i = V_i' \cup V_i''$ und $V_i' \cap V_i'' = \emptyset$. Den einzelnen V_i können Typen zugeordnet werden: Gilt $i = 1$, $i = k+1$ oder $c(V_i) > \lambda(G)$, so ist V_i vom Typ *Kette*. Anderenfalls ist V_i vom Typ *alter Kreis*, falls $c(V_{i-1} \cup V_i) = \lambda(G)$, sonst vom Typ *neuer Kreis*. Für Kreis-Typen gilt also, dass sie selbst auch minimale Schnitte sind, wobei nur alte Kreise mit ihrem Vorgänger zusammen einen Schnitt bilden. Karzanov und Timofeev haben folgende wichtige Eigenschaften bewiesen:

Lemma 2.2 ([KT86])**.** Sei $G = (V, E)$ und $\{s, t\} \in E$, dann ist $\mathcal{C}(s, t, G)$ parallel.

Lemma 2.3 ([KT86, Bix75, Fle99])**.** Seien S_1, S_2 zwei kreuzende Schnitte in G. Dann enthält G eine *zirkuläre Partition* $(V_1, \ldots V_k)$, so dass gilt: $\forall 1 \leq i \leq \ell \leq k : \bigcup_{j=i}^{\ell} V_j \in \mathcal{C}(G)$. Für zwei verschiedene zirkuläre Partitionen $(V_1, \ldots, V_k)$ und $(W_1, \ldots, W_p)$ gilt, dass sie zueinander *kompatibel* sind, das heißt es gibt eindeutige $i, j, i \neq j$ mit $\forall q \neq j : W_q \subset V_i$ und $\forall r \neq i : V_r \subset W_j$.

Für zirkuläre Partitionen wird analog zu den parallelen Partitionen gefordert, dass sich sich nicht erweitern lassen.

2.2.4 Simultane Schnitte

Da minimale Schnitte nicht nur in einem Graphen, sondern in einer Familie von Graphen gesucht sind, stellt sich die Frage, ob die Berechnung minimaler Schnitte vereinfacht werden kann. Es wird davon ausgegangen, dass sich die Graphen ähnlich sind, das heißt, dass sie „viele“ Kanten gemeinsam haben. Die Annahme ist beispielsweise für Graphen, die Beziehungen von Objekten untereinander zu verschiedenen Zeitpunkten widerspiegeln eine realistische Annahme. In diesem Abschnitt wird untersucht, wie man von den minimalen Schnitten eines Graphen auf die minimalen Schnitte eines ähnlichen Graphen schließen kann. Dazu wird zunächst betrachtet, wie sich die Menge minimaler Schnitte durch eine *Kantenoperation*, das heißt das Einfügen oder Entfernen einer Kante, verändert.

Hinzufügen von Kanten. Es sei daran erinnert, dass $\mathcal{C}(G)$ die Menge minimaler Schnitte in G bezeichnet, und das $\mathcal{C}(u, v, G) \subseteq \mathcal{C}(G)$ die Menge minimaler Schnitte ist, die u und v trennen. Falls es keine minimalen u-v-Schnitte gibt, so ist $\mathcal{C}(u, v, G)$ die leere Menge. Für das Einfügen von Kanten in einem Graphen gilt das folgende Lemma:

Lemma 2.4. Sei $G = (V, E)$ ein Graph, für den $\mathcal{C}(G) \neq \mathcal{C}(u, v, G)$ gilt und bezeichne $G' = (V, E')$ mit $E' = E \cup \{\{u, v\}\}$ den Graphen, den man erhält, wenn man die Kante $\{u, v\}$ zu G hinzunimmt. Dann gilt für die Menge $\mathcal{C}(G')$ der minimalen Schnitte in G':

$$\mathcal{C}(G') = \mathcal{C}(G) \setminus \mathcal{C}(u, v, G)$$

Beweis. Konstruiert man aus einem Graphen G durch Einfügen einer Kante $e = \{u, v\}$ den Graphen G', dann erhöht sich die Größe genau derjenigen Schnitte, die die Knoten u und v trennen, da die neue Kante Schnittkante aller dieser Schnitte ist. Schnitte, die u und v nicht trennen und damit e nicht schneiden, bleiben in ihrer Größe unverändert. Sofern es also in G außer den Schnitten, die u und v trennen noch weitere minimale Schnitte gibt ($\mathcal{C}(G) \neq \mathcal{C}(u, v, G)$), bilden diese die Menge aller minimalen Schnitte in G'. □

Im Fall $\mathcal{C}(G) \neq \mathcal{C}(u, v, G)$ lässt sich damit die Menge minimaler Schnitte zu einen Graphen schnell aktualisieren. Allerdings macht das Lemma keine Aussage für den Fall $\mathcal{C}(G) = \mathcal{C}(u, v, G)$, denn kein minimaler Schnitt hätte in G und G' die gleiche Größe und damit folgt $\lambda(G) < \lambda(G')$. Im ungewichteten Fall ist zwar klar, dass $\mathcal{C}(G) \subset \mathcal{C}(G')$ gilt, aber ob es weitere minimale Schnitte in G' gibt lässt sich aus $\mathcal{C}(G)$ nicht herleiten. Im gewichteten Fall lässt sich überhaupt keine Aussage treffen. Damit bleibt nichts anderes übrig, als die minimalen Schnitte komplett neu zu berechnen.

Entfernen von Kanten. Auch für Graphen, die sich durch Entfernen von Kanten ergeben, lässt sich eine Aussage treffen:

Lemma 2.5. Bezeichne $G' = (V, E')$ mit $E' = E \setminus \{\{u, v\}\}$ den Graphen, den man erhält, wenn man die Kante $\{u, v\} \in E$ aus G entfernt. Dann gilt für die Menge $\mathcal{C}(G')$ der minimalen Schnitte in G':

$$\mathcal{C}(G') = \begin{cases} \mathcal{C}(u, v, G) & \text{falls } \mathcal{C}(u, v, G) \neq \emptyset \\ \mathcal{C}(G) & \text{falls } \mathcal{C}(u, v, G) = \emptyset, \lambda(u, v, G') > \lambda(G) \\ \mathcal{C}(G) \cup \mathcal{C}(u, v, G') & \text{falls } \mathcal{C}(u, v, G) = \emptyset, \lambda(u, v, G') = \lambda(G) \\ \mathcal{C}(u, v, G') & \text{falls } \mathcal{C}(u, v, G) = \emptyset, \lambda(u, v, G') < \lambda(G) \end{cases}$$

Beweis. Für den Beweis werden zunächst die Fälle $\mathcal{C}(u, v, G) \neq \emptyset$ und $\mathcal{C}(u, v, G) = \emptyset$ unterschieden:

a) Falls $\mathcal{C}(u, v, G) \neq \emptyset$, das heißt, falls es in G bereits minimale Schnitte gab, die u und v trennten, dann werden diese Schnitte durch das Entfernen von $\{u, v\}$ kleiner, während alle anderen Schnitte in ihrer Größe unverändert bleiben. Damit bilden die u-v-Schnitte in G die Menge aller Schnitte in G'.

b) Falls es in G keine minimalen u-v-Schnitte gibt und der kleinste Schnitt, der u und v in G' trennt

   i) größer ist als $\lambda(G)$, dann bleibt die Menge der minimalen Schnitte unverändert, da alle Schnitte, die durch das Entfernen von $\{u, v\}$ kleiner werden, größer sind, als die minimalen Schnitte.

   ii) gleich $\lambda(G)$ ist, dann bleiben alle minimalen Schnitte aus G auch in G' minimal und es kommen nur Schnitte hinzu, die u und v trennen, da alle anderen Schnitt in ihrer Größe unverändert bleiben.

   iii) kleiner ist als $\lambda(G)$, dann sind die minimalen Schnitte in G' genau die u-v-Schnitte in G. □

Auch beim Entfernen von Kanten treten also verschiedene Fälle auf. Zur Bestimmung von $\mathcal{C}(G')$ muss demnach zuerst festgestellt werden, welcher Fall vorliegt. Ob Fall 1 aus Lemma 2.5, das heißt $\mathcal{C}(u,v,G) \neq \emptyset$, zutrifft, ist analog wie beim Einfügen in $O(|\mathcal{C}(G)|)$ festzustellen; $\mathcal{C}(G')$ ist dann auch in dieser Zeit berechenbar. Anderenfalls ist es zur Unterscheidung der übrigen Fälle erforderlich $\lambda(u,v,G')$ zu bestimmen. Im zweiten Fall ändert sich die Menge der minimalen Schnitte nicht. In den Fällen 3 und 4 muss auch $\mathcal{C}(u,v,G')$ berechnet werden, um $\mathcal{C}(G')$ zu bestimmen. Die Bestimmung von $\mathcal{C}(u,v,G')$ und die Feststellung, ob $\lambda(u,v,G') > \lambda(G)$ gilt ist zwar asymptotisch in gleicher Zeit möglich, jedoch ist eine einzelne Flussberechnung durch kleinere konstante Faktoren schneller durchzuführen als eine komplette Neuberechnung aller minimaler Schnitte. Für die Feststellung, ob $\lambda(u,v,G') > \lambda(G)$, kann die Berechnung sogar schon abgebrochen werden, sobald ein Präfluss gefunden ist, der größer ist als $\lambda(G)$. Denn aus jedem Präfluss lässt sich ein Fluss gleicher Größe konstruieren, indem Überschüsse zurück zur Quelle transportiert werden; dadurch kann sich der Fluss, der in t ankommt nicht verringern.

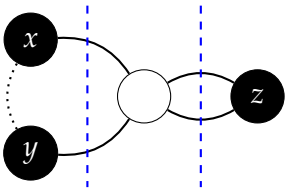
2.3 Kaktus-Repräsentation minimaler Schnitte

Speichert man Schnitte trivial durch Markierung oder Aufzählung der Schnittknoten oder Schnittkanten so benötigt man $O(n)$ bzw. $O(m)$ Speicherplatz pro Schnitt; bei bis zu $\binom{n}{2}$ minimalen Schnitten in einem Graphen [DKL76] bedeutet das einen Speicherbedarf von $O(n^3)$. Dinic, Karzanov und Lomonosov haben den Kaktus als Repräsentation für alle minimalen Schnitte entdeckt, der alle minimalen Schnitte mit Speicherbedarf in $O(n)$ abbildet [DKL76]. Für die Beschreibung der Kaktus-Repräsentation müssen zunächst noch einige Begriffe definiert werden.

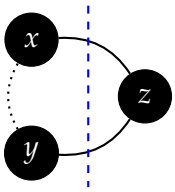
Ein *einfacher Pfad* in einem Graphen $G = (V, E)$ bezeichne eine Knotenfolge $(v_1, \ldots, v_\ell)$, so dass gilt: $\forall i \in \{1, \ldots, \ell - 1\} : \{v_i, v_{i+1}\} \in E$ und $\forall i, j \in \{1, \ldots, \ell\} : i = j \vee v_i \neq v_j$. Ein *einfacher Kreis* bezeichne die Menge der Knoten eines Pfads $(v_1, \ldots, v_n)$ mit $\{v_1, v_n\} \in E$. Die Menge aller Kreise eines Graphen G wird mit $\mathcal{K}(G)$ bezeichnet. Im Folgenden ist mit einem Kreis bzw. einem Pfad immer ein einfacher Kreis oder Pfad gemeint. Der Einfachheit halber wird je nach Kontext mit einem Kreis oder Pfad auch die Folge der Kanten bezeichnet, die zwischen zwei Knoten v_i, v_{i+1} liegen.

Ein *Kaktus* $\mathcal{R} = (W, F)$ ist ein zusammenhängender, ungewichteter und ungerichteter Graph, in dem jede Kante zu genau einem Kreis gehört. Abweichend von der Definition eines Graphen, kann ein Kaktus Mehrfachkanten enthalten; die folglich jeweils zu einem Kreis der Größe 2 gehören.[1] Da jede Kante zu genau einem Kreis gehört, lässt sich ein Kaktus eindeutig in eine Menge von Kreisen $\mathcal{K}$ zerlegen. Ein Kaktusknoten heißt k-Verbindungsknoten, wenn er zu $k \geq 2$ Kreisen gehört.

[1] In der Literatur ist auch eine alternative Definition gebräuchlich, in der keine Mehrfachkanten erlaubt sind, dafür aber Kanten zugelassen werden, die die zu keinem Kreis gehören und *Baumkanten* genannt werden. Die Definition erfordert allerdings, dass Baumkanten Gewicht 2 und alle anderen Kanten Gewicht 1 haben.



(a) Der Kaktus repräsentiert zwei mal den gleichen Schnitt.



(b) Nach Kontraktion des Kreises (und damit der Entfernung des leeren Knotens) wird der gleiche Schnitt nur noch einmal repräsentiert.

Abbildung 2.3: Leere Knoten (weiß), die nur zu zwei Kreisen gehören, von denen einer die Größe zwei hat, lassen sich durch Kontraktion eliminieren ohne die repräsentierten Schnitte zu verändern.

Für einen Graphen $G = (V, E)$ sei $(\mathcal{R}, \varphi)$ ein Tupel aus einem Kaktus $\mathcal{R} = (W, F)$ und einer Abbildung $\varphi : V \to W$, die Graphknoten auf Kaktusknoten abbildet. Für $x \in W$ und $S \subset V$ gelte $\varphi(S) = \{y \in W \mid \exists s \in S : \varphi(s) = y\}$, $\varphi^{-1}(x) = \{v \in V \mid \varphi(v) = x\}$ und für $X \subset W$ sei $\varphi^{-1}(X) = \{v \in V \mid \varphi(v) \in X\}$. Dann ist $(\mathcal{R}, \varphi)$ genau dann *Kaktus-Repräsentation* minimaler Schnitte in G, wenn gilt:

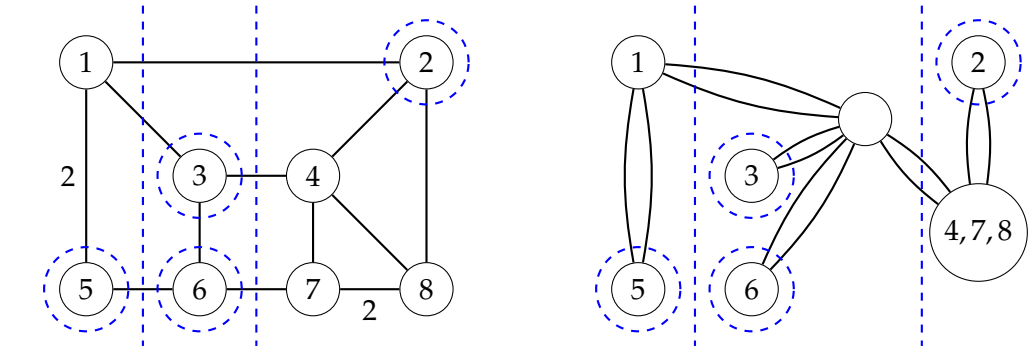
a) Für jeden minimalen Schnitt $S \in \mathcal{C}(\mathcal{R})$ gilt $\varphi^{-1}(S)$ ist ein minimaler Schnitt in G.

b) Umgekehrt gilt für jeden minimalen Schnitt $S \in \mathcal{C}(G)$, dass auch $\varphi(S)$ ein minimaler Schnitt in $\mathcal{R}$ ist.

Kaktusknoten x, für die $\varphi^{-1}(x) = \emptyset$ gilt, heißen *leer*. Abbildung 2.4 zeigt einen Graphen und eine zugehörige Kaktus-Repräsentation seiner minimalen Schnitte. Zur Vereinfachung wird gefordert, dass es keine leeren Knoten gibt, die zu zwei Kreisen gehören, von denen einer ein Kreis der Größe zwei ist, denn dann ließen sich der Kreis kontrahieren, ohne dass sich die Menge der Repräsentierten Schnitte ändern würde und der leere Knoten würde verschwinden (siehe Abbildung 2.3). Wenn klar ist, dass die Kaktusrepräsentation gemeint ist, so wird diese auch kurz mit *Kaktus* bezeichnet.

Zur besseren Unterscheidung werden die Knoten in einem Graphen als *Graphknoten* oder einfach nur als Knoten bezeichnet und die Knoten eines Kaktus in jedem Fall als *Kaktusknoten*; W bezeichnet immer die Kaktusknotenmenge, und F die Kaktuskantenmenge. Graphknoten werden mit s, t, u, v, w bezeichnet, währen Kaktusknoten x, y oder z heißen.

Dinic et al. haben gezeigt, dass zu jedem Graphen eine Kaktus-Repräsentation existiert [DKL76]. Mit ihr lassen sich minimale Schnitte eines Graphen kompakt, das heißt mit einem Platzbedarf in $O(n)$, repräsentieren. Aufgrund der einfachen Struktur eines Kaktus sind dessen minimalen Schnitte leicht zu bestimmen; sie schneiden jeweils zwei Kanten des gleichen Kreises. Umgekehrt bilden zwei Kanten eines Kreises immer eine Schnittkantenmenge.



(a) Ein Graph G mit allen seinen minimalen Schnitten der Größe 3.

(b) Ein Kaktus zum Graph G; die Abbildung φ ergibt sich aus den Knotenbeschriftungen.

Abbildung 2.4: Beispiel eines Graphen mit seiner Kaktusrepräsentation.

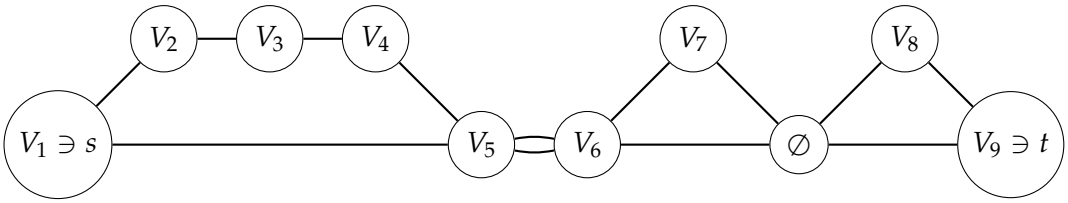


Abbildung 2.5: Beispiel eines s-t-Kaktus. V_1, V_5, V_6 und V_9 sind vom Typ Kette. V_2, V_7 und V_8 sind vom Typ neuer Kreis und V_3 sowie V_4 vom Typ alter Kreis.

2.3.1 Konstruktion

Der Kaktus zu einem Graphen ist nicht eindeutig. Da jedoch eine eindeutige Kaktus-Repräsentation für die korrekte Berechnung wichtig ist, ist es erforderlich weitere Eigenschaften festzulegen. Die ersten Algorithmen von Karzanov und Timofeev oder Naor und Vazirani zur Konstruktion der Kaktus-Repräsentation enthielten Fehler (siehe [Fle99]), die erst mit der Festlegung einer korrekten kanonischen Darstellung einer Kaktus-Repräsentation durch Nagamochi und Kameda [NK94] beseitigt werden konnten. Fleischer konnte die Laufzeit für die Konstruktion eines Kaktus durch die Kombination des Karzanov-Timofeev-Frameworks mit dem Hao-Orlin-Algorithmus auf $O(mn \log(n^2/m))$ verbessern [Fle99]. Nagamochi et al. entwickelten einen Algorithmus mit Laufzeit $O(mn + n^2 \log n)$ der auf MAO basiert [NNI03].

Die Methode von Nagamochi ist in Algorithmus 1 zu sehen. In Zeile 8 wird ein Pendent Pair einer MAO bestimmt, so dass eine Kante s und t verbindet. Da für eine MAO $(v_1, \ldots, v_n)$ mit $\{v_{n-1}, v_n\} \notin E$ gilt, dass auch $\{v_1, \ldots, v_{n-2}, v_n\}$ eine MAO in $G \setminus \{v_{n-1}\}$ ist, kann hier die Kante $\{v_p, v_n\}$ mit maximalem p gewählt werden. In Zeile 15 werden alle minimalen s-t-Schnitte mit Hilfe des Algorithmus von [PQ80] berechnet, indem starke Zusammenhangskomponenten im Residualgraph bezüglich f gesucht werden. Daraus wird eine parallele Partition $(V_1, \ldots, V_k)$ konstruiert. In Zeile 17 wird schließlich

ein s-t-Kaktus zusammengesetzt, also ein Kaktus, der alle s-t-Schnitte aus $P(s,t)$ enthält, wobei je nach Typ der Partitionsmenge ein (neuer) Kreis gebildet wird oder die Knoten eine Kette bilden (Abbildung 2.5). Falls ein neuer Kreis auf einen Kreis folgt, wird ein leerer Knoten eingefügt. In Zeile 20 wird der Kaktus-Algorithmus schließlich rekursiv aufgerufen und die einzelnen Kakteen zu einem Kaktus zusammengeführt; Details hierzu siehe [NI08].

Algorithmus 1 : Kaktus

Input : Graph G
Output : Kaktus-Repräsentation $(\mathcal{R}, \varphi)$

```
1  begin
2      if |V| = 1 then
3          R ← ({x}, ∅);
4          φ maps the graph node to x ;
5          return (R, φ)
6      σ ← computeMaximumAdjacencyOrdering(G) ;
8      (s, t) ← pendentPair(σ) ;
9      if λ(s, t) > λ then
10         contract(s, t);
11         return cactus(G, λ);
12     else
13         f ← pendentFlow(G, s, t, σ);
15         P(s, t) ← s-t-Partition(f);
17         (R, φ) ← s-t-Kaktus(P(s, t)) ;
18         foreach V_i ∈ P(s, t) do
20             (R, φ) = (R, φ)⊕Kaktus(G/(V \ V_i));
21         return (R, φ);
22 end
```

2.3.2 Dynamische Kaktus-Repräsentation

In diesem Abschnitt werden die Erkenntnisse über der Veränderung minimaler Schnitte beim Einfügen oder Entfernen von Kanten auf die Kaktus-Repräsentation übertragen, um eine dynamische Kaktus-Repräsentation zu entwickeln. Das heißt, statt einer vollständigen Neuberechnung des Kaktus zu einem Graphen G' der durch eine Kantenoperation α auf G entsteht, soll der Kaktus durch eine Operation β aktualisiert werden (siehe Abbildung 2.6). Dazu wird für den Fall aus Lemma 2.4 und für jeden Fall aus Lemma 2.5 jeweils eine Kaktus-Operation vorgestellt, die die Kaktus-Repräsentation nach dem Einfügen oder Entfernen einer Kante aktualisiert, so dass sie eine Repräsentation aller minimalen Schnitte bleibt. Es wird gezeigt, dass der Kaktus in einigen Fällen in $O(n)$ aktualisiert werden kann, während es Fälle gibt, in denen asymptotisch keine Zeitersparnis möglich ist.

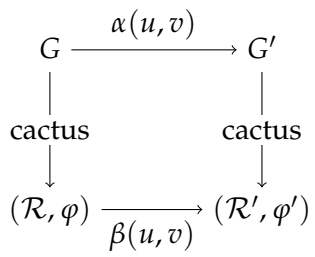


Abbildung 2.6: Die dynamische Kaktus-Repräsentation stellt für Kanten-Einfüge- und Kanten-Lösch-Operationen α Operationen β zur Verfügung, die den Kaktus aktualiseren.

Veränderungen der Kaktus-Repräsentation durch Hinzufügen von Kanten

Sei nun wieder $G = (V, E)$ ein Graph und bezeichne $G' = (V, E')$ mit $E' = E \cup \{\{u, v\}\}$ den Graphen, den man erhält, wenn man die Kante $\{u, v\}$ zu G hinzunimmt. Mit $(\mathcal{R}, \varphi)$ wird die Kaktus-Repräsentation zu G und mit $(\mathcal{R}', \varphi')$ die zu G' bezeichnet. Das Einfügen einer Kante führt genau dann dazu, dass alle bisherigen minimalen Schnitte größer werden, wenn alle minimalen Schnitte u und v trennen. Lemma 2.4 liefert für diesen Fall keine Möglichkeit aus der Menge der minimalen Schnitte $\mathcal{C}(G)$ in G auf minimale Schnitte in $\mathcal{C}(G')$ zu schließen. Damit ist eine vollständige Neuberechnung von $\mathcal{R}'$ erforderlich.

Existieren aber auch Schnitte, die u und v nicht trennen, so lässt sich der Kaktus $\mathcal{R}'$ aus $\mathcal{R}$ konstruieren, indem alle Schnitte aus $\mathcal{R}$, die u-v-Schnitte Repräsentieren, entfernt werden. Dazu wird noch eine Definition benötigt:

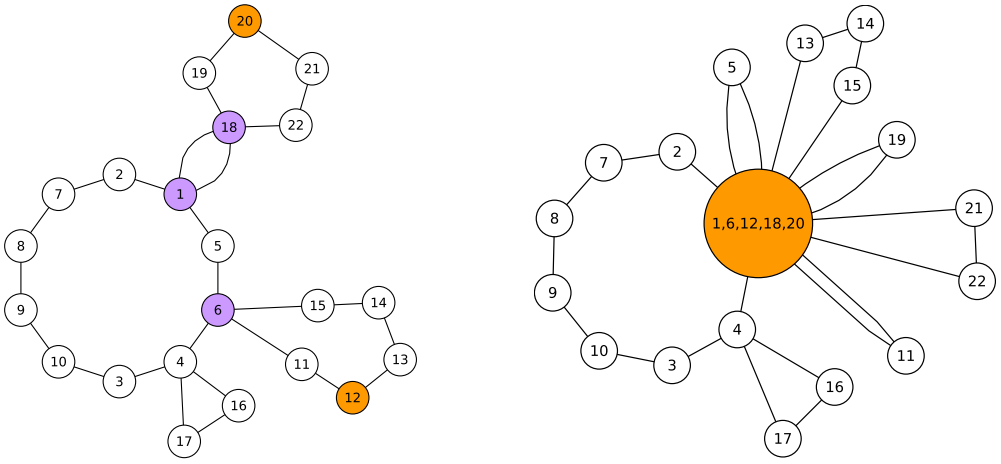
Die *Kontraktion* der Knoten u_1 mit u_2 ist definiert als das Ersetzen der Knoten u_1 und u_2 durch einen neuen Knoten v, sowie aller Kanten $\{u_i, w\}$ durch $\{v, w\}$, falls $v \neq w$. Angewandt auf eine Kaktus-Repräsentation $(\mathcal{R}, \varphi)$ bedeutet das zusätzlich, dass φ durch

$$\varphi'(u) = \begin{cases} v & \text{falls } \varphi(u) \in \{u_1, u_2\} \\ \varphi(u) & \text{sonst.} \end{cases}$$

ersetzt wird.

Da sich jede Kante genau einem Kreis zuordnen lässt, implizieren die Kanten eines kürzesten Pfads zwischen je zwei Knoten x, y in einem Kaktus eine kürzeste Kreisefolge, die aus Kreisen $K_i \in \mathcal{K}(\mathcal{R})$ besteht und keinen Kreis zweimal enthält. Diese kürzeste Kreisfolge ist in jedem Kaktus eindeutig. Bezeichne $P_{x,y} = (K_1, \ldots, K_k)$ diese Kreisfolge, wobei K_1 (bzw. K_k) der Kreis ist, der x (bzw. y) enthält und es gilt $|K_i \cap K_{i+1}| = 1$ für $i \in \{1, \ldots, k-1\}$. Seien weiter $z_i \in K_i \cap K_{i+1}$ die eindeutigen Knoten, die zu den Kreisen K_i und K_{i+1} gehören. Diese Knoten z_i werden als *Pfad-Verbindungsknoten* in $\mathcal{R}$ zwischen x und y bezeichnet.

Die Operation `kontrahiere`$((\mathcal{R}, \varphi), u, v)$ kontrahiert die Knoten $\varphi(u)$ und $\varphi(v)$ und alle Pfad-Verbindungsknoten in $(\mathcal{R}, \varphi)$ (siehe Abbildung 2.7).



(a) Der Kaktus vor der Kontraktion der Knoten 20 und 12 mit den Verbindungsknoten 1, 6 und 18.

(b) Der Kaktus nach der Kontraktion.

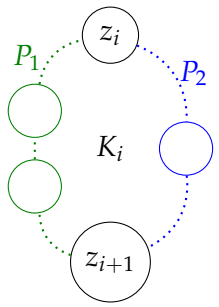
Abbildung 2.7: Beispiel: Die Operation `kontrahiere` angewendet auf einen Kaktus.

Lemma 2.6. Sei $G = (V, E)$ ein Graph, $u, v \in V$ und sei $(\mathcal{R}, \varphi)$ die Kaktus-Repräsentation seiner minimalen Schnitte. Dann repräsentiert die Kaktus-Repräsentation $(\mathcal{R}', \varphi')$, die durch die Anwendung der Operation `kontrahiere`$((\mathcal{R}, \varphi), u, v)$, entsteht, alle minimalen Schnitte, die u und v nicht trennen, das heißt er repräsentiert genau die Schnittmenge $\mathcal{C}(G) \setminus \mathcal{C}(u, v, G)$. Die Konstruktion von $(\mathcal{R}', \varphi')$ benötigt $O(n)$ Schritte.

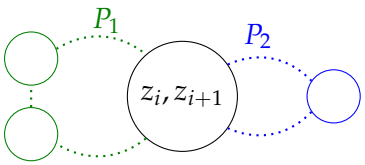
Beweis. Seien $x = \varphi(u)$ und $y = \varphi(v)$. Die Schnitte in $\mathcal{R}$, die x und y trennen, repräsentieren genau die Schnitte in G, die u und v trennen. Da u und v nach der Kontraktion auf den gleichen Knoten abgebildet werden, dass heißt es gilt $\varphi'(u) = \varphi'(v)$, kann es keine Schnitte mehr zwischen u und v geben, die $\mathcal{R}'$ repräsentiert.

Da jeder Schnitt in einem Kaktus immer genau aus zwei Schnittkanten eines Kreises besteht, muss für die Korrektheit von $\mathcal{R}'$ noch gezeigt werden, dass für jeden Kreis gilt, dass durch die Kontraktion alle Schnitte erhalten bleiben, die x und y nicht trennen. Sei $P_{x,y} = (K_1, \ldots, K_k)$ die kürzeste Kreisfolge der Kreise zwischen x und y und seien $z_1, \ldots, z_{k-1}$ die Pfad-Verbindungsknoten, so dass z_i der Knoten ist, den die Kreise K_i und K_{i+1} gemeinsam haben. Jeder Schnitt in einem Kreis K_i, der die zwei Knoten z_{i-1} und z_i trennt, trennt auch x und y. Seien P_1 und P_2 die beiden Pfade der Kanten in $\mathcal{R}$, die zwischen den Knoten z_{i-1} und z_i liegen. Schnitte, die nur zwei Kanten aus P_1 (bzw. P_2) enthalten, trennen die Knoten z_{i-1} und z_i nicht, während Schnitte die jeweils eine Kante aus P_1 und P_2 enthalten z_{i-1} und z_i trennen (siehe Abbildung 2.8). Kontrahiert man nun z_{i-1} und z_i, so erhält man zwei Kreise, die genau die Kanten aus P_1 bzw. P_2 enthalten und somit bleiben durch die Kontraktion alle Schnitte, die x und y nicht trennen, erhalten.

Im Kaktus $\mathcal{R}$ kann in $O(n)$ mit einer Tiefensuche der Pfad von x nach y, sowie alle Verbindungsknoten z_i in $\mathcal{R}$ bestimmt werden. Die Anzahl der Knoten die entfernt werden sowie die Anzahl der Kanten die ersetzt werden liegen ebenfalls in $O(n)$. □



(a) Ein Kreis K_i mit Pfaden P_1 und P_2 vor der Kontraktion. Genau die Schnitte, die z_i und z_{i+1} trennen, Schneiden Kanten aus P_1 und P_2.



(b) K_i bildet nach der Kontraktion zwei Kreise, die aus den Kanten aus P_1 und P_2 bestehen.

Abbildung 2.8: Kontraktion

Die Anwendung von Lemma 2.6 liefert für Kakteen, die nur u-v-Schnitte repräsentieren einen trivialen Kaktus mit nur einem Knoten. Somit kann also auch in $O(n)$ festgestellt werden, ob eine Neuberechnung des Kaktus erforderlich ist.

Veränderungen der Kaktus-Repräsentation durch Entfernen von Kanten

Es werden nun die Veränderungen einer Kaktus-Repräsentation $(\mathcal{R}, \varphi)$ zu einem Graphen G durch Entfernen von Kanten in G betrachtet. In Lemma 2.5 wurden vier Fälle identifiziert, wie sich die Menge der minimalen Schnitte in G durch diese Operation verändert. Diese sollen nun auf Kakteen übertragen werden. Zur Vorbereitung wird noch folgendes Lemma benötigt:

Lemma 2.7. In einem Graphen $G = (V, E)$ mit einer Kante $\{u, v\} \in E$ gibt es in der zugehörigen Kaktus-Repräsentation $(\mathcal{R}, \varphi)$ einen Pfad zwischen $\varphi(u)$ und $\varphi(v)$ dessen Kanten alle zu unterschiedlichen Kreisen gehören.

Beweis. Aus Lemma 2.2 folgt, dass alle u-v-Schnitte parallel sind. Angenommen jeder Pfad zwischen $\varphi(u)$ und $\varphi(v)$ enthält mindestens zwei Kanten die zum gleiches Kreis gehören.Dann liegt auf dem Pfad ein Kreis K mit zwei Pfad-Verbindungsknoten z_1 und z_2 und die beiden Kreiskanten in K, die zu z_1 inzident sind, haben als jeweils zweiten Endknoten die Knoten x_1, x_2 mit $x_i \neq z_2$. Dann gibt es in G' zwei Schnitte S_1, S_2 mit $\varphi^{-1}(\{z_1, x_1\}) \subset S_1$ und $\varphi^{-1}(\{z_2, x_2\}) \subset \overline{S_1}$ sowie $\varphi^{-1}(\{w_1, x_2\}) \subset S_2$ und $\varphi^{-1}(\{w_2, x_1\}) \subset \overline{S_2}$, da diese Schnitte u und v trennen. Nach dem Lemma 2.6 gilt dann, dass auch $S_1 \setminus S_2 = \varphi^{-1}(x_1)$ ein Schnitt sein müsste. Das ist aber ein Widerspruch zu Lemma 2.2 (siehe Abbildung 2.9). □

Nun kann die erste Operation `Kette`$((\mathcal{R}, \varphi), u, v)$ definiert werden. Sei $\mathcal{R} = (W, F)$, $x = \varphi(u)$ und $y = \varphi(v)$. In jedem Kaktus gibt es genau zwei kantendisjunkte einfache Pfade zwischen je zwei Knoten, da Kakteen aus Kreisen bestehen und es zwischen zwei

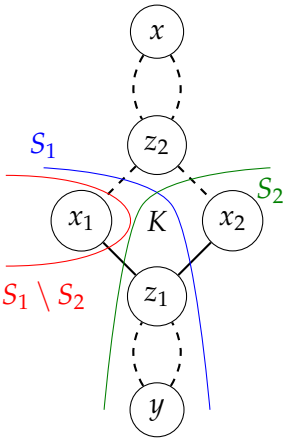
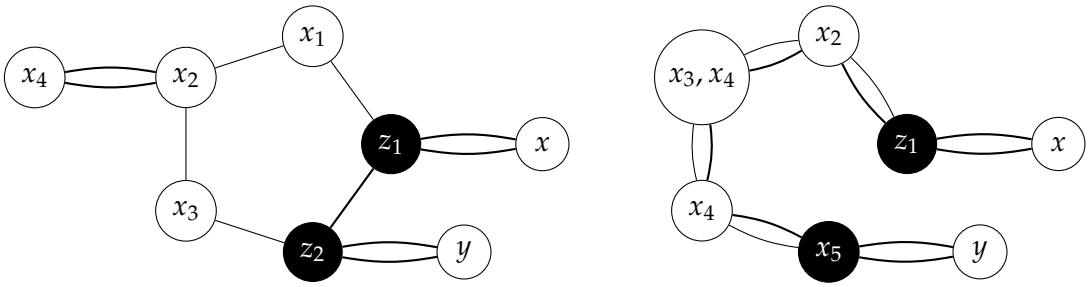


Abbildung 2.9: Widerspruch: Gäbe es einen solchen Kreis K auf den Pfad zwischen $x = \varphi(u)$ und $y = \varphi(v)$, so könnte man einen Schnitt $S_1 \setminus S_2$ konstruieren.



(a) Der „alte“ Kaktus $\mathcal{R}$: Die Verbindungsknoten (schwarz) im Kreis sind durch eine Kante verbunden.

(b) Der neue Kaktus $\mathcal{R}'$: Die Kante zwischen den Verbindungsknoten wird entfernt, alle übrigen Kreiskanten werden verdoppelt.

Abbildung 2.10: Beispiel: Konstruktion des Kaktus mit der Operation `Kette`.

Kreisknoten jeweils zwei Pfade in jedem Kreis gibt. Bezeichne nun $P_{x,y}$ den kürzesten Pfad von x nach y und sei $P'_{x,y}$ der zu $P_{x,y}$ kantendisjunkte Pfad zwischen x und y. Die Operation ist anwendbar, wenn $P_{x,y}$ nur aus Kanten besteht, die zu paarweise verschiedenen Kreisen gehören.

Die Operation `Kette` konstruiert nun die Kaktusrepräsentation $(\mathcal{R}', \varphi')$ mit $\mathcal{R} = (W, F')$, indem alle Kaktuskanten auf dem Pfad $P_{x,y}$ entfernt und alle Kanten aus $P'_{x,y}$ verdoppelt werden, das heißt $F' = F \setminus P_{x,y} \uplus P'_{x,y}$, wobei $\uplus$ die Vereinigung für Mehrfachmengen bezeichne. Zudem werden für jeden Kreis K, der keine Kanten aus $P_{x,y}$ oder $P'_{x,y}$ enthält, alle Knoten von K kontrahiert (Siehe Abbildung 2.10).

Die Operation wird durch Algorithmus 2 umgesetzt.

Lemma 2.8. Sei $G = (V, E)$ ein Graph und $(\mathcal{R}, \varphi)$ die Kaktus-Repräsentation seiner minimalen Schnitte sowie $\{u, v\} \in E$ eine Kante mit $\varphi(u) \neq \varphi(v)$. Bezeichne $G' = (V, E')$ den Graphen, der durch Entfernen der Kante $\{u, v\}$ aus G entsteht, also $E' = E \setminus \{\{u, v\}\}$.

Algorithmus 2 : Kette

Input : Kaktus-Repräsentation $(\mathcal{R} = (W, F), \varphi)$, Kaktusknoten $x, y \in W$
Output : Kaktus-Repräsentation $(\mathcal{R} = (W, F), \varphi)$, die nur (x,y)-Schnitte enthält.

```
1  begin
2      P ← Pfad(x, y) ⊂ F ;                          // kürzester Pfad von x nach y
3      forall e = {s, t} ∈ P do
4          entferneKante (e);
5          P_e ← Pfad(s, t);
6          forall {a, b} ∈ P_e do
7              neueKante (a, b);
8              kontrahiere (a) ;          // Kontrahiere alle Knoten, die von a aus
                                          // ohne Knoten aus P oder P_e erreichbar sind
9      kontrahiere (b);
10 end
```

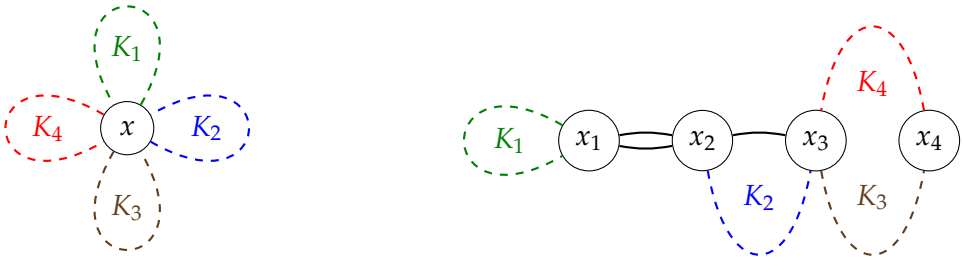
Dann ist die Kaktus-Repräsentation $(\mathcal{R}', \varphi')$, die durch Anwendung der Operation `Kette`$((\mathcal{R}, \varphi), u, v)$ entsteht, die Repräsentation aller minimaler Schnitte in G'.

Beweis. In der Kaktus-Repräsentation entsprechen die Schnitte, die u und v trennen, genau den Schnitten, die $x = \varphi(u)$ und $y = \varphi(v)$ trennen. Nach Lemma 2.5 gilt, dass $\mathcal{C}(G') = \mathcal{C}(u, v, G)$ ist; damit ist zu zeigen, dass $\mathcal{C}(\mathcal{R}') = \mathcal{C}(x, y, \mathcal{R})$ gilt. Seien $P_{x,y}$ und $P'_{x,y}$ definiert wie in der Definition der Operation `Kette`. Aus Lemma 2.7 folgt, dass die Operation `Kette` anwendbar ist und das $P_{x,y}$ nur aus Pfad-Verbindungsknoten sowie x und y besteht. Genau die Schnitte $\mathcal{C}(G') \subset \mathcal{C}(\mathcal{R})$, die im Kaktus aus einer Kante aus $P_{x,y}$ und einer weiteren Kante des gleichen Kreises bestehen, sind u-v-Schnitte. Durch das Entfernen der Kanten aus $P_{x,y}$ und dem Verdoppeln der Kanten aus $P'_{x,y}$ werden die Kreise, auf denen Kanten aus $P_{x,y}$ liegen, aufgespalten. Die Kontraktionen betreffen nur Kreise, die nicht zwischen x und y liegen. Damit bilden diese doppelten Kanten genau die Schnitte aus $\mathcal{C}(G')$, es wird also kein x-y-Schnitt entfernt.

Umgekehrt sind die Schnitte der so erhaltenen Kette alle x-y-Schnitte. Da alle Kreise, die nicht in eine Kette umgewandelt wurden, kontrahiert wurden, können diese keine Schnitte mehr repräsentieren.

Für die Laufzeit gilt: Das Suchen der Pfade im Kaktus benötigt $O(n)$ Schritte. Alle Kaktuskanten müssen genau einmal betrachtet und entweder entfernt, verdoppelt oder kontrahiert werden. Die Zeitkomplexität von `Kette` liegt damit in $O(n)$. □

Als nächstes soll die Operation `extrahiere`$((\mathcal{R}, \varphi), G, u, v)$ vorgestellt werden; sie ist das Gegenstück der Operation `kontrahiere` (siehe Abschnitt 2.3.2). Durch die Operation `extrahiere` wird die Kaktus-Repräsentation $(\mathcal{R}, \varphi)$ um die minimalen Schnitte in G erweitert, die u und v trennen. Sie ist anwendbar, wenn $\varphi(u) = \varphi(v)$ gilt. Anschaulich bedeutet das, dass der Kaktusknoten $x = \varphi(u)$ in mehrere Kaktusknoten $x_1, \ldots, x_r$



(a) Ein Kaktusknoten x, der zu Kanten aus 4 Kreisen K_1, K_2, K_3, K_4 inzident ist.

(b) Der Knoten x wurde aufgespalten in x_1, x_2, x_3, x_4. Es gibt verschiedene Möglichkeiten, wie die Kreise K_1, K_2, K_3, K_4 auf die Knoten x_i verteilt werden.

Abbildung 2.11: Beispiel: Konstruktion des Kaktus mit der Operation `Kette`.

aufgespalten wird und die zu x inzidenten Kreise zwischen den x_i „aufgehängt" werden (siehe Abbildung 2.11). Die Operation ist wie folgt definiert:

Sei $x = \varphi(u) = \varphi(v)$ der Kaktusknoten, der extrahiert werden soll und sei $X = \varphi^{-1}(x)$. Da die Knoten aus X nach entfernen von $\{u, v\}$ durch mindestens einen Schnitt getrennt werden, müssen diese als erstes auf neu zu erzeugende Kaktusknoten $x_1, \dots x_r$ abgebildet werden. Sei nun $(X_1, \dots, X_r)$ eine durch u-v-Schnitte induzierte Partition von X, so dass gilt

$$\forall s \in X_i, t \in X_j, i \neq j \exists S \in \mathcal{C}(u, v, G') : S \text{ trennt } u \text{ und } v, \tag{2.6}$$

$$\forall S \in \mathcal{C}(u, v, G') \exists j \in \{1, \dots, r\} : \bigcup_{i=1}^{j} X_i \subset S \wedge \bigcup_{i=j+1}^{r} X_i \subset \overline{S}, \tag{2.7}$$

wobei ohne Beschränkung der Allgemeinheit $u \in S$ angenommen wird. Dann wird x ersetzt durch $x_1, \dots, x_r$, das heißt $W' = W \setminus \{x\} \cup \{x_1, \dots, x_r\}$ und φ' definiert durch

$$\varphi'(w) = \begin{cases} x_i & \text{, falls } w \in X_i \\ \varphi(w) & \text{, sonst} \end{cases}$$

In einem zweiten Schritt werden die noch zu x inzidenten Kanten mit den neuen Knoten x_i verbunden. Zwei Kanten gehören dabei immer zu einem Kreis. Diese Kreise werden dann entweder zwischen x_i und x_{i+1} für ein $i < r$ „aufgehängt" oder beide Kanten desselben Kreises werden mit dem gleichen x_i verbunden.

Sei $E' = \{\{s, t\} \in E \mid s \neq x, t \neq\}$ also die Menge der Kanten aus E die nicht zu x inzident sind. Für Kreise K sei $\psi(K)$ der durch die Kanten $\{x, y\}, \{x, z\} \in K$ induzierte Schnitt T mit $y, z \in T$. Für jeden Kreis K gilt:

a) Falls es keinen u-v-Schnitt S in G' gibt, der $\varphi^{-1}(y)$ und $\varphi^{-1}(z)$ trennt, dann werden die Kanten $\{x_i, y\}, \{x_i, z\}$ in E' eingefügt für das kleinste i mit

$$\exists S \in \mathcal{C}(u, v, G') : X_i \cup \psi(K) \subset S, X_{i+1} \subset \overline{S}$$

Algorithmus 3 : Extrahiere

Input : Kaktus-Repräsentation $(\mathcal{R} = (W, F), \varphi)$, Graph $G = (V, E)$, Graphknoten $u, v \in V$

Output : Kaktus-Repräsentation $(\mathcal{R} = (W, F), \varphi)$, die um u-v-Schnitte erweitert wurde

```
1  begin
2      M ← Partition(u,v);
3      forall M ∈ M do
4          X ← φ(M) ;                    /* Kaktusknoten, die M repräsentieren */
5          if φ(u) ∈ X then
6              y ← neuerKaktusKnoten;
7              schliesseKreise;
8              kompletteKreiseAn (y, X) ;  /* Alle Kreise, die vollständig in X  */
                                           /* enthalten sind, werden an y angehängt. */
9          else
10             if NeuerKreis(M) ;          /* Der Kreis zu M ist noch nicht offen */
11             then
12                 oeffneKreis;
13 end
```

b) Falls es $S \in \mathcal{C}(u, v, G')$ gibt mit S trennt X_i und X_{i+1} und $\varphi(S)$ trennt y und z, wobei ohne Beschränkung der Allgemeinheit $X_i \subset S, y \in \varphi(S)$ sei, dann werden die Kanten $\{x_i, y\}, \{x_{i+1}, z\}$ in E' eingefügt.

Schließlich werden zwischen allen Paaren (x_i, x_{i+1}) bis zu zwei Kanten eingefügt bis es jeweils zwei kantendisjunkte Pfade zwischen x_i und x_{i+1} gibt. Das heißt x_i und x_{i+1} sind entweder durch zwei Kreise (sie bilden dann zusammen einen neuen Kreis), zwei Kanten oder durch einen Kreis und eine Kante verbunden.

Der Pseudocode der Operation ist in Algorithmus 3 zu sehen. Das folgende Lemma soll nun zeigen, dass die Operation das Geforderte leistet:

Lemma 2.9. Sei $G = (V, E)$ ein Graph und $(\mathcal{R}, \varphi)$ die Kaktus-Repräsentation seiner minimalen Schnitte. Bezeichne $G' = (V, E')$ mit $E' = E \setminus \{\{u, v\}\}$ den Graphen, der durch Entfernen einer Kante $\{u, v\} \in E$ mit $\varphi(u) = \varphi(v)$ und entsteht. Falls $\lambda(u, v, G') = \lambda(G)$ gilt, dann ist die Kaktus-Repräsentation $(\mathcal{R}', \varphi')$, die durch Anwendung der Operation `extrahiere`$((\mathcal{R}, \varphi), u, v)$ entsteht, eine Repräsentation aller minimaler Schnitt in G'. Die Operation `extrahiere` benötigt $O(\text{maxflow}(m, n))$ Schritte.

Beweis. Nach Lemma 2.5 gilt $\mathcal{C}(G') = \mathcal{C}(G) \cup \mathcal{C}(u, v, G')$. Zu zeigen ist also, dass $(\mathcal{R}', \varphi')$ zum einen alle minimalen Schnitte repräsentiert, die $(\mathcal{R}, \varphi)$ repräsentiert und zum anderen alle neuen minimalen u-v-Schnitte in G'. Umgekehrt müssen alle neuen Schnitte minimale u-v-Schnitte sein.

Die Kaktusrepräsentation $(\mathcal{R}', \varphi')$ repräsentiert alle Schnitte, die auch $(\mathcal{R}, \varphi)$ repräsentiert, denn Schnittkanten eines Schnitts im Kaktus sind immer zwei Kanten eines Kreises. Kreise, die $x = \varphi(u)$ in $\mathcal{R}$ nicht enthalten, werden nicht verändert und repräsentieren daher in $\mathcal{R}'$ die gleichen Schnitte. Für Kreise, die x in $\mathcal{R}$ enthalten, gilt, dass nur die zwei Kanten $\{x, y\}, \{x, z\} \in F$ verändert werden zu $\{x_i, y\}, \{x_j, z\} \in F'$ (für $i, j \leq r$) und die übrigen Kanten in $\mathcal{R}'$ unverändert beibehalten werden (siehe Abbildung 2.11). Damit kann in $\mathcal{R}'$ durch die gleichen Schnittkanten wie in $\mathcal{R}$ ein minimaler Schnitt erzeugt werden. Diese repräsentieren auch die gleichen Knotenmengen; die Seite eines Schnitts, die x in $\mathcal{R}$ nicht enthält, enthält auch $x_1, \ldots, x_r$ nicht, da der Teil des Kreises auf dieser Seite zwischen den Knoten y und z nicht verändert wurde.

Es wird nun gezeigt, dass jeder u-v-Schnitt durch $(\mathcal{R}', \varphi')$ repräsentiert wird. Dazu wird benötigt, dass die Bedingung aus Gleichung (2.7) der Definition von `kontrahiere` tatsächlich für alle Schnitte $S \in \mathcal{C}(u, v, G')$ erfüllt werden kann. Das ist der Fall, denn gäbe es Schnitte S_1, S_2 mit $\bigcup_{i=1}^{j} X_i \subseteq S_1, \bigcup_{i=j+1}^{r} X_i \subseteq \overline{S_1}$ und $X_1 \cup X_k \subseteq S_2, X_\ell \subseteq \overline{S_2}$ mit $1 < \ell < k \leq j$, dann gäbe es nach Lemma 2.1 einen Schnitt T mit $X_\ell \subseteq T$; da T kein u-v-Schnitt ist, wäre T auch ein Schnitt in G und das ist ein Widerspruch zur Kaktus-Repräsentation $(\mathcal{R}, \varphi)$.

Sei also S ein beliebiger u-v-Schnitt. Dann gibt es ein $j < k$ mit $\bigcup_{i=1}^{j} X_i \subseteq S, \bigcup_{i=j+1}^{r} X_i \subseteq \overline{S}$. Für jeden Kreis $K = (y_1, \ldots, y_p)$ in $\mathcal{R}'$ gilt, dass er entweder ganz oder teilweise in S enthalten ist. Falls K ganz in S enthalten ist, so ist dieser Kreis wegen Buchstabe a der Definition mit einem x_i für ein $i \leq j$ verbunden. Ist K nur teilweise in S enthalten, so ist dieser Kreis wegen Buchstabe b der Definition mit x_j und x_{j+1} verbunden. Weiter gilt, falls die Komponente $\varphi^{-1}(y_\ell)$ in S enthalten ist, so müssen auch alle Komponenten $\varphi^{-1}(y_i)$ für $i < \ell$ in S enthalten sein, denn anderenfalls würden die Komponenten von K, die in S enthalten sind, zusammen einen Schnitt bilden, der nicht u-v-Schnitt ist, aber nicht von $(\mathcal{R}, \varphi)$ repräsentiert wird. Damit wird S also durch $(\mathcal{R}', \varphi')$ repräsentiert. Umgekehrt folgt aus Lemma 2.1, dass, falls S ein minimaler Schnitt mit $\varphi^{-1}(y_i) \in S$ ist, auch $S \cup \varphi^{-1}(y_{i+1})$ ein minimaler Schnitt sein muss. Alle neuen Schnitte in $(\mathcal{R}', \varphi')$ müssen daher minimale Schnitte sein. □

Die letzte Operation ist die Operation u-v-`Kaktus`. Es wird eine Kaktus-Repräsentation $(\mathcal{R}, \varphi)$ für einen Graphen G berechnet, der nur minimale Schnitte enthält, die u und v trennen. Diese Schnitte müssen parallel sein, da G sonst auch Schnitte enthalten würde, die u und v nicht trennen. Die Operation ist daher die gleiche, wie die zur Berechnung eines gesamten Kaktus ohne den Abstieg in die Rekursion. Die Zeitkomplexität entspricht damit der Zeit für die Flussberechnung $O(\text{maxflow}(m, n))$.

Damit wurden alle Fälle der dynamischen Kaktus-Repräsentation behandelt. Die Lemmata, Eigenschaften und Definitionen aus den vorangegangenen Abschnitten lassen sich nun zusammenfassen. Insgesamt ergibt sich als Ergebnis für die dynamische Kaktusrepräsentation folgender Satz:

Satz 1. Aus der Kaktus-Repräsentation $(\mathcal{R}, \varphi)$ für $G = (V, E)$, kann in $O(\text{maxflow}(m, n))$ Schritten eine Kaktus-Repräsentation $(\mathcal{R}', \varphi')$ für $G' = (V, E')$ mit

- $E' = E \setminus \{\{u, v\}\}$ konstruiert werden durch

$$(\mathcal{R}', \varphi') = \begin{cases} \texttt{kette}((\mathcal{R}, \varphi), u, v) & \text{falls } \varphi(u) \neq \varphi(v) \\ (\mathcal{R}, \varphi) & \text{falls } \varphi(u) = \varphi(v) \wedge \lambda(u, v, G') > \lambda(G) \\ \texttt{extrahiere}((\mathcal{R}, \varphi), G', u, v) & \text{falls } \varphi(u) = \varphi(v) \wedge \lambda(u, v, G') = \lambda(G) \\ u\text{-}v\text{-}\texttt{kaktus}(G', u, v) & \text{falls } \varphi(u) = \varphi(v) \wedge \lambda(u, v, G') < \lambda(G) \end{cases}$$

- $E' = E \cup \{\{u, v\}\}$ konstruiert werden durch

$$(\mathcal{R}', \varphi') = \begin{cases} \texttt{kontrahiere}((\mathcal{R}, \varphi), G', u, v) & \text{falls } \mathcal{C}(G) \neq \mathcal{C}(u, v, G) \\ \texttt{kaktus}(G') & \text{sonst} \end{cases}$$

2.3.3 Simultanes Berechnen minimaler Schnitte

Die Einfüge- und Entfernen-Operationen mit den dazugehörigen Kakteen legen Algorithmen nahe, die die Kaktus-Repräsentationen $\mathcal{R}_i$ einer Familie von Graphen $G_i = (V, E_i)$ berechnet.

Der erste Ansatz geht von dem Graphen $G = (V, E)$ aus, der alle gemeinsamen Kanten enthält, das heißt $E = \bigcap E_i$. Für jeden Graphen G_i werden dann die fehlenden Kanten sukzessive einfügt. Das Problem hierbei ist, dass der Kaktus dabei zu einem Knoten zusammenschrumpfen kann und ein völliges Neuberechnen der Kaktus-Repräsentation erforderlich ist.

Der zweite Ansatz geht von dem Graphen $G = (V, E)$ aus, der alle Kanten enthält, die in einem der Graphen G_i vorkommen, das heißt $E = \bigcup E_i$. Hier werden dann für jedes G_i die Kanten, die nicht zum Graphen gehören, sukzessive entfernt. Je nach Lage der zu entfernenden Kante müssen im schlechtesten Fall minimale u-v-Schnitte zwischen zwei gegebenen Knoten neu berechnet werden. Auch wenn der asymptotische Aufwand zur Berechnung eines u-v-Flusses der gleiche ist, wie zur Neuberechnung einer Kaktus-Repräsentation, so ist dennoch zu erwarten, dass durch das Wegfallen hoher konstanter Faktoren für die Kaktusberechnung eine effizientere Berechnung möglich ist. Der Vergleich der tatsächlichen Laufzeiten wird in Abschnitt 5.2 behandelt.

2.4 Vergleich minimaler Schnitte

In diesem Abschnitt wird nun untersucht, wie die mit Algorithmus 4 berechneten minimalen Schnitte einer Familie von Graphen $(G_i)_{1 \leq i \leq k}$ bezüglich des Ähnlichkeitsmaßes f aus Gleichung (2.5) verglichen werden können.

Algorithmus 4 : berechneKakteen

Eingabe : Familie von Graphen (G_i)
Ausgabe : Kaktus-Repräsentationen $(\mathcal{R}_i, \varphi_i)$ für (G_i)

```
1  begin
2      G = (V, E) ← (V, ∪E_i) ;          /* Graph der alle Kanten aus (G_i) enthält */
3      (R, φ) ← cactus(G) ;
4      forall G_i do
5          E' ← E ;
6          (R_i, φ_i) ← (R, φ) ;
7          forall e = {u, v} ∈ E \ E_i do
8              E'' ← E' \ {e};
9              if φ(u) = φ(v) then                  /* Flussberechnung erforderlich */
10                 if λ(u, v, G'') = λ(G') then
11                     (R_i, φ_i) ← extrahiere((R_i, φ_i), G'', u, v) ; /* um u-v-Schnitte  */
                                                                        /* erweitert */
12                 if λ(u, v, G'') < λ(G') then
13                     (R_i, φ_i) ← u-v-kaktus(G'', u, v) ;    /* nur mit u-v-Schnitten */
14                     λ(G') ← λ(G') − c(u, v);
15             else
16                 (R_i, φ_i) ← kette((R_i, φ_i), u, v) ;        /* Kaktus, der nur noch */
                                                                 /* u-v-Schnitte enthält */
17             E' ← E'' ;
18 end
```

2.4.1 Exakte Lösungverfahren

Es gibt $\prod_{i=1}^{k} |\mathcal{C}(G_i)|$ Möglichkeiten die minimalen Schnitte zu kombinieren. Alle diese Schnittkombinationen durchzuprobieren führt daher zu exponentiellem Aufwand in der Anzahl der Graphen k. Um den Vergleichsaufwand zu verringern, soll deshalb ein Branch-and-Bound-Ansatz verfolgt werden.

Branch and Bound. In dem zu Algorithmus 5 gehörenden Berechnungsbaum werden in Höhe i als Kindknoten die Schnitte des Graphen $G_{\sigma(i)}$ nacheinander gewählt und der zu überprüfenden Schnitt-Kombination hinzugefügt. Dabei ist $\sigma : \{1, \dots k\} \to \{1, \dots k\}$ die Permutation die die Graphen aufsteigend nach der Anzahl ihrer Schnitte sortiert, es gilt also für $i \in \{1, \dots, k-1\}$: $\left|\mathcal{C}(G_{\sigma(i)})\right| \leq \left|\mathcal{C}(G_{\sigma(i+1)})\right|$. Damit erhöht sich der Nutzen, falls die Berechnung in einem Zweig nicht fortgesetzt werden muss. Da sich $f(S_1, \dots, S_{i+1})$ aus $f(S_1, \dots, S_i)$ berechnen lässt und f dadurch nur größer werden kann, kann als Schranke für das absteigen in den Berechnungsbaum $f(S_1, \dots, S_i) + \Delta_i \geq f_{\min}$ gewählt werden, wobei $f_{\min}$ das beste bisher gefundene Ergebnis ist und idealerweise mit dem Ergebnis einer schnellen Heuristik (siehe Abschnitt 2.4.2) initialisiert wurde.

Δ_i ist eine obere Schranke für den Wert, der durch Hinzunahme von $S_{i+1}, \ldots, S_k$ zu f mindestens noch aufaddiert werden muss, das heißt für

$$\min_{(S_{i+1},\ldots S_k)\in\mathcal{C}(G_{\sigma(i)})\times\cdots\times\mathcal{C}(G_{\sigma(k)})} f(S_1,\ldots,S_k) - f(S_1,\ldots,S_i)$$

und kann beispielsweise definiert werden durch die Summe der minimalen Zuwächse, die sich durch das Hinzufügen jeweils zwei weiterer Schnitte der übrigen Graphen ergibt:

$$\Delta_i = \sum_{j=\lceil i/2\rceil}^{\lfloor (k-1)/2\rfloor} \min_{(S,T)\in\mathcal{C}(G_{\sigma(2j)})\times\mathcal{C}(G_{\sigma(2j+1)})} f(S,T)$$

Das erfordert einmalig Rechenzeit von $O(kn^2)$. Der Abstieg erfolgt in aufsteigender Reihenfolge bezüglich des Ähnlichkeitsmaßes f für die Schnitte $S_i \in \mathcal{C}(G_{\sigma(i)})$.

Algorithmus 5 : minCutCompare

Input : $(\mathcal{C}(G_i))_{1\le i\le k}$, sortiert nach der Anzahl der Schnitte,
$(S_1,\ldots,S_{j-1})$ gewählte Schnitte
Output : $(f(bestCuts), bestCuts)$

```
1  begin
2      sort(C(G_j)) ;                    // Sortiere Schnitte nach Ähnlichkeit bzgl. f
                                         // mit den bisher gewählten Schnitten
3      foreach S_j ∈ C(G_j) do
4          if f(S_1,...,S_{j-1},S) + Δ < f_min then
5              (m, cuts) ← mincutCompare((C(G_i))_{1≤i≤k}, (S_1,...,S_{j-1},S_j)) ;
6              if m < f_min then
7                  bestCuts ← cuts;
8                  f_min ← m;
9      return (f_min, bestCuts) ;
10 end
```

2.4.2 Heuristik

Da trotz Branch-And-Bound-Optimierung die Laufzeit im schlechtesten Fall exponentiell bleibt, ist eine Heuristik sinnvoll, die möglichst schnell eine gute Lösung liefert. Diese soll außerdem als Eingabe für die exakte Lösung eine Verwendung finden. Eine solche Heuristik soll nun vorgestellt werden.

Die Heuristik beginnt mit einem beliebigen Schnitt $S_1 \in \mathcal{C}(G_1)$ im ersten Graphen und wählt dann sukzessive für $i \in \{2,\ldots,k\}$ den Schnitt $S_i \in \mathcal{C}(G_i)$, der bei bereits gewählten $S_1,\ldots,S_{i-1}$ die Funktion $f(S_1,\ldots,S_{i-1},S_i)$ minimiert. Dieses Vorgehen wird für jedes $S_j \in \mathcal{C}(G_1)$ wiederholt. Auch hier ist es sinnvoll die Graphen vorher in der Reihenfolge

Algorithmus 6 : minCutCompareHeuristic

Input : $(\mathcal{C}(G_i))_{1 \leq i \leq k}$, sortiert nach der Anzahl der Schnitte
Output : $(f(bestCuts), bestCuts)$

```
1  begin
2      f_min ← ∞;
3      foreach S_1 ∈ C(G_1) do
4          currentCuts[1] = S_1;
5          for i = 2 to k do
6              S_i = argmin_{S∈C(G_i)} f(currentCuts, S_i);
7              currentCuts[i] ← S_i;
8              if i = k ∧ f(currentCuts) < f_min then
9                  bestCuts ← currentCuts;
10                 f_min ← f(bestCuts);
11     return (f_min, bestCuts) ;
12 end
```

der Anzahl ihrer minimalen Schnitte zu sortieren. Der Pseudocode ist in Algorithmus 6 dargestellt.

Da die Wahl des Schnitts eines Graphen, der früh an die Reihe kam, das heißt ein Graph G_i mit kleinem i, nur von den vorher gewählten Schnitten abhing, ist die Wahrscheinlichkeit bei diesen Graphen hoch, dass es Schnitte gibt, die ein kleineres f liefern. Dem kann man Entgegensteuern, indem man mehrmals über die Graphen iteriert und jeweils den besten minimalen Schnitt eines Graphen G_i nimmt bezüglich der Funktion f, wenn man alle anderen $S_j, j \neq i$ festhält. Die Anzahl der Iterationen kann vorgegeben werden oder es wird so lange Iteriert, bis sich nichts mehr ändert.

Die Laufzeit von Algorithmus 6 beträgt $O(|\mathcal{C}(G_1)|\,(\sum_{i=2}^{k} |\mathcal{C}(G_i|)n^2)$, denn für jeden Schnitt in G_1 muss für jeden anderen Schnitt der übrigen Graphen einmal f berechnet werden. Da $\mathcal{C}(G_i) \in O(n^2)$ kommt man theoretisch auf $O(kn^6)$, da quadratisch viele minimale Schnitte in Graphen aber sehr ungewöhnlich sind, ist eine deutlich geringere Laufzeit zu erwarten.

3

Kleine Schnitte

In diesem Kapitel wird die relaxierte Variante des simultanen Schnitt-Problems betrachtet, das heißt, es wird ein Schnitt gesucht, der in allen Graphen möglichst klein ist. In einem Graphen G ist für die Anzahl kleiner Schnitte $S \in \mathcal{C}(G)$ mit $c(S) < (1+\varepsilon)\lambda$ die obere Schranke $O(n^{2(1+\varepsilon)})$ bekannt [Kar93]. Für die Anzahl der k-größten Schnitte kennt man die Schranke $O(n^{3k-1})$ [VY92]. Für den Spezialfall $\varepsilon < 1/2$ gilt die Schranke $O(n^2)$ für die Anzahl der Schnitte [HW96].

Im ersten Abschnitt werden exakte Lösungsverfahren vorgestellt. Diese basieren auf bekannten Algorithmen von Nagamochi et al. [NNI97] sowie Vazirani und Yannakakis [VY92]. Eine Heuristik zur schnellen Lösung des Problems wird in Abschnitt 2 präsentiert. Im dritten Abschnitt wird die Frage diskutiert, wie Familien von Graphen, die nicht zusammenhängend sind, behandelt werden können. Abschnitt 4 behandelt ein rekursives Verfahren zum simultanen Clustern von Graphen basierend auf simultanen kleinen Schnitten.

3.1 Exakte Lösungsverfahren

Um den Vergleich von Graphen einer Familie (G_i) mit verschiedenen $\lambda(G_i)$ zu erleichtern, soll zunächst der Begriff der Normalisierung eines Graphen eingeführt werden. Ein Graph $G = (V, E)$ mit Kantengewichtsfunktion $c : E \to \mathbb{R}$ heißt *normal*, wenn $\lambda(G) = 1$ gilt. Graphen können *normalisiert* werden, indem die Kantengewichtsfunktion c ersetzt wird durch c' mit

$$c'(e) := \frac{c(e)}{\lambda_G}$$

Das Problem der kleinen Schnitte für Familien von normalen Graphen lässt sich dann formulieren als die Frage nach einem Schnitt S mit

$$c(S) = \min_{\emptyset \neq S \subsetneq V} \max_{i=1,\ldots k} c(S, G_i),$$

denn es gilt für alle $S \subset V : c(S, G) = c(S, G_{\text{normal}}) \cdot \lambda(G)$.

3.1.1 Lösung auf Basis des Algorithmus von Nagamochi, Nishimura und Ibaraki

Nagamochi et al. [NNI97] veröffentlichten 1997 einen Algorithmus, der für ein $\varepsilon > 0$ alle Schnitte in einem Graphen, die höchstens $(1 + \varepsilon)$ mal so groß sind wie der minimale Schnitt, in $O(m^2 n + n^{2(1+\varepsilon)} m)$ Schritten berechnet. Er ist also exponentiell in ε, was sich im allgemeinen Fall auch nicht verbessern lässt, da es exponentiell viele Schnitte gibt [NNI97].

Der Algorithmus von Nagamochi et al. arbeitet in zwei Phasen. In der ersten Phase wird der Graph bis auf einen Knoten verkleinert. Dabei werden Informationen gesammelt, mit denen in der zweiten Phase die geforderten Schnitte berechnet werden.

In der ersten Phase werden Split-Operationen verwendet, die für Knoten $s, u, v \in V$ die Kantengewichte zwischen ihnen ändert. Genauer heißt das, die Kantengewichtsfunktion c wird durch c' ersetzt mit

$$c'(s,u) = c(s,u) - \delta, \quad c'(s,v) = c(s,v) - \delta, \quad c'(u,v) = c(u,v) + \delta$$

und $c'(x,y) = c(x,y)$ für $\{x,y\}$ mit $|\{x,y\} \cap \{s,u,v\}| \leq 1$ (siehe Abbildung 3.1). Dabei werden Kanten, die Kantengewicht 0 erhalten, automatisch entfernt bzw. neue Kanten eingefügt, falls ihr Kantengewicht erhöht wird und sie noch nicht existieren. Die Größe aller Schnitte, die s und $\{u,v\}$ trennen, reduziert sich dabei um 2δ, während die Größe aller weiteren Schnitte durch die Anwendung der Split-Operation erhalten bleibt. Nagamochi et al. zeigen, dass man den Parameter δ geschickt wählen kann, so dass die Größe aller minimalen Schnitte höchstens dann kleiner wird, wenn $\{s\}$ der einzige kleinste Schnitt ist. Durch mehrfache Anwendung der Split-Operation lässt sich ein Knoten s isolieren, so dass s zu keiner Kante inzident ist; dieser kann dann entfernt werden. Der Algorithmus isoliert und entfernt sukzessive alle Knoten, bis nur noch ein Knoten übrig bleibt und dokumentiert dabei die Split-Operationen.

In einem zweiten Schritt wird eine Menge von kleinen Schnitten konstruiert. Angefangen mit einer leeren Menge T, wird für jeden Knoten v_i in umgekehrter Reihenfolge der Isolierung die Menge erweitert durch $T = T \cup \{S \cup \{v_i\} : S \in T\} \cup \{\{v_i\}\}$. Mit Hilfe der dokumentierten Split-Operationen können die Schnittgrößen berechnet werden, indem für jeden Schnittkandidaten S und jede Split-Operation $\texttt{split}(s,u,v,\delta)$ geprüft wird, ob S die Mengen $\{s\}$ und $\{u,v\}$ trennt und gegebenenfalls δ addiert wird. Schnitte, die größer sind als eine vorgegebene Schranke werden aussortiert. Wichtig ist, dass die

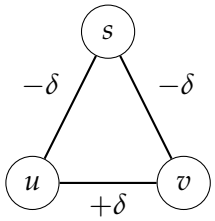


Abbildung 3.1: Die Operation split(s, u, v, δ).

weitgehende Erhaltung des Kantenzusammenhangs während der ersten Phase dafür sorgt, dass Schnitte, die zu groß würden, schon frühzeitig aussortiert werden können.

Der Algorithmus lässt sich zur simultanen Berechnung kleiner Schnitte in einer Familie von Graphen anpassen. Dafür werden die Graphen zunächst normalisiert. In Phase 1 müssen die Knoten aller Graphen in der gleichen Reihenfolge betrachtet werden. Der Vorteil der simultanen Berechnung ergibt sich aus dem zweiten Teil. Hier werden nur Schnitte in der Menge T mitgeführt, die in *allen* Graphen klein genug sind; das Überschreiten der gegebenen Grenze von nur einem Graphen führt zur Aussortierung. Dadurch reduziert sich zum einen die Menge der infrage kommenden kleinen Schnitte und zum anderen steht jeder dieser Schnitte nur einmal im Speicher. Am Ende entfällt auch ein Vergleich der gefundenen minimalen Schnitte von verschiedenen Graphen; es muss lediglich der kleinste der gefundenen Schnitte ausgegeben werden (siehe Algorithmus 7).

Lemma 3.1. Algorithmus 7 berechnet für ein $\varepsilon > 0$ die Menge aller Schnitte S mit $c(S, G_i) < (1 + \varepsilon)\lambda(G_i)$.

Beweis. In Phase 1 werden die Split-Operationen für alle G_i berechnet. Dabei beeinflussen sich die Berechnungen für verschiedene Graphen lediglich in der Wahl der Reihenfolge der zu isolierenden Knoten, da diese in allen Graphen gleich sein muss. Die Reihenfolge der Knoten ist die einzige Abweichung vom Original-Algorithmus von Nagamochi et al. in Phase 1. Da die Reihenfolge für die Korrektheit des Algorithmus irrelevant ist [NNI97] folgt die Korrektheit von Phase 1 in Algorithmus 7 aus der Korrektheit von Phase 1 im Originalalgorithmus.

Die Phase 2 wird für alle Graphen simultan durchgeführt; für $k = 1$ entspricht sie dem Originalalgorithmus. Die Zeile 15 wird in allen Graphen unabhängig von den übrigen Graphen ausgeführt. Nur in Zeile 17 beeinflussen sich die Graphen, da hier jeder Graph die Schnitte aussortiert, die Größer sind als $1 + \varepsilon$, das heißt ein Schnitt S wird dann aus T entfernt, wenn es einen Graphen G gibt, in dem $c(s, G) > 1 + \varepsilon$ gilt. Nagamochi et al. haben gezeigt [NNI97], dass alle Schnitte, die durch Ausführung von Zeile 11 entstehen, mindestens die Größe $c(s, G)$ haben. Damit ist es irrelevant, welche Größe ein Schnitt S oder ein aus S durch Ausführung von Zeile 11 konstruierbarer Schnitt in den anderen Graphen hat. Das Entfernen dieses Schnitts entspricht also der erforderlichen Schnittmengenbildung und damit gilt also wie vom Algorithmus verlangt $\bigcap_{i=1}^{k} \{S \subset V \mid c(S, G_i) < (1 + \varepsilon)\}$. □

Algorithmus 7 : SimultaneousSmallCutsA

Input : Graphfamilie (G_i)
Output : Ein kleinster Schnitt S

```
1  begin
2      Für i = 1,...,k: normalize((G_i));
3      while |V| > 1 do                                          // Phase 1
4          s ← argmin_{s∈V} Σ_{i=1}^{k} deg(s, G_i) ;
5          for i = 1 to k do
6              Q_{s,i} ← isolate(s,G_i) ;      // Q_{s,i} enthält die Parameter δ,u,v aller
                                  // Split-Operationen, die isolate(s,G_i) ausgeführt hat.
7          V ← V \ {s} ;
8      T ← ∅ ;                                  // T enthält Schnittkandidaten,
                                          // sowie deren Größe für jeden Graphen.
9      foreach isolated node s in reverse order do                // Phase 2
11         T ← {S ∪ {s} : S ∈ T} ∪ T ∪ {s} ;
12         for i = 1 to k do
13             foreach (δ, u, v) ∈ Q_{s,i} do
15                 updateWeights(T,(δ,u,v),i);
17             T ← {S ∈ T | c(s, G_i) < (1 + ε)};
18     return argmin_{S∈T} max_{i≤k} c(S, G_i);
19 end
```

Lemma 3.2. Die Laufzeit von Algorithmus 7 liegt in $O(m^2n + n^{2k}m)$.

Beweis. Für die Laufzeitanalyse von Phase 1 wird die Anzahl der Split-Operationen gezählt; Nagamochi et al. verwenden, dass die Knoten in einer bestimmten Reihenfolge isoliert werden. Sie haben gezeigt, dass jede Knotenisolierung höchstens $\deg(s) \leq 2m/n$ Split-Operationen erfordert und die Kantenzahl nicht erhöht. Im simultanen Fall kann man den Knoten s der nächsten Isolierung durch $s = \operatorname{argmin}_{v \in V} \sum_{i=1}^{k} \deg(v, G_i)$ wählen. Es gilt dann analog zum Beweis von Nagamochi et al. $\sum_{i=1}^{k} \deg(v, G_i) \leq 2 \sum_{i=1}^{k} m_i/n$, wenn m_i die Anzahl der Kanten in G_i ist. Für ein m mit $m_i \in O(m)$ gilt dann $\sum_{i=1}^{k} \deg(v, G_i) \leq 2km/n$. Damit erhöht sich die Anzahl der Splitoperationen für k Graphen gegenüber dem Originalalgorithmus um den Faktor k auf $O(mk)$. Für die erste Phase bedeutet das eine Laufzeit von $O(kmn(m + n \log n))$, da eine Split-Operation in $O(n(m + n \log n))$ möglich ist..

Das Aktualisieren der Schnittgrößen in Zeile 15 ist in $O(|T|)$ möglich; die Laufzeit der beiden inneren Schleifen beträgt damit $O(km\,|T|)$. Da die Anzahl von Schnitten kleiner $1 + \varepsilon$ in $O(n^{2(1+\varepsilon)})$ liegt [Kar93], ergibt sich damit eine Laufzeit von $O(kmn^{2(1+\varepsilon)})$ für Phase 2.

Insgesamt beträgt die Laufzeit beider Phasen damit $O(k(m^2n + n^{2(1+\varepsilon)}m))$. □

Da die Anzahl der Split-Operationen, die dokumentiert werden müssen, in $O(km)$ liegt und es $O(n^{2(1+\varepsilon)})$ Schnitte mit Platzbedarf in $O(n+k)$ gibt liegt der Speicherbedarf insgesamt in $O(km + (n+k)n^{2(1+\varepsilon)})$. Der Algorithmus von Nagamochi et al. lässt sich verbessern, indem er mit einem Branch-And-Bound-Ansatz verbunden wird, das heißt, die Schnitte können mittels Tiefensuche aufgezählt werden. Der Berechnungsbaum ist dabei ein vollständiger binärer Baum der Höhe n. Die Knoten auf Ebene i im Baum repräsentieren einen teilweise definierten Schnitt; für die Knoten $\{v_1, \ldots, v_i\}$ einer Graphfamilie ist also festgelegt, ob sie Teil der Schnittmenge sind. Das linke (bzw. rechte) Kind u_{i+1} eines Berechnungsbaumknotens u_i repräsentiert den Schnitt seines Vaterknotens erweitert um die Information, ob der Graphknoten v_i Teil des Schnitts ist. Damit enthalten alle Blätter vollständig definierten Schnitte. Die Suche wird analog zum Originalalgorithmus abgebrochen, falls die Schnittgröße die gegebene Schranke übersteigt. Wird ein Blattknoten im Berechnungsbaum erreicht, so merkt sich der Algorithmus den besten Schnitt oder gibt alle Schnitte aus, falls alle Schnitte unterhalb einer Grenze aufgezählt werden sollen. Insgesamt werden die gleichen Berechnungen ausgeführt wie im Originalalgorithmus und damit die gleiche Zeitkomplexität erreicht. Der Speicherbedarf reduziert sich aber auf $O(km)$.

Da die Schranke $(1+\varepsilon)$ entscheidend für die Laufzeit ist, und für die Lösung nur der kleinste gültige Schnitt interessant ist, sollte diese so klein wie möglich gewählt werden. Wie eine solche Schranke mit einer Heuristik gefunden werden kann wird in Abschnitt 3.2 besprochen. Der modifiziere Algorithmus mit Branch-And-Bound-Ansatz bietet jedoch eine weitere Möglichkeit, die eine Schranke überflüssig macht. Man setzt dazu die Schranke auf unendlich und passt diese dann dynamisch dem besten bisher gefundenen Schnitt an.

3.1.2 Lösung auf Basis des Algorithmus von Vazirani und Yannakakis

Vazirani und Yannakakis haben einen Algorithmus entwickelt, der die kleinsten Schnitte in einem Graphen in aufsteigender Reihenfolge aufzählt [VY92]. Der Algorithmus arbeitet mit einem Heap, in dem Schnittkandidaten gespeichert werden. Initialisiert wird der Heap mit einem minimalen Schnitt, die Priorität eines Schnitts S entspricht der Größe des Schnitts $c(S)$. Es wird jeweils der kleinste Schnitt S aus dem Heap entfernt und ausgegeben. Aus S werden dann bis zu n weitere Schnitte mit jeweils einer Maximalflussberechnung bestimmt und in den Heap eingefügt. Zu jedem Zeitpunkt gilt, dass sich der jeweils kleinste noch nicht ausgegebene Schnitt im Heap befindet.

Nun kann man die Schnitte in verschiedenen Graphen vergleichen, bis man in allen Graphen identische Schnitte gefunden hat. Geht man davon aus, dass sich die Graphen ähnlich sind, für die man zwei gleiche kleine Schnitte sucht, so wird man erwarten, dass das ε relativ klein ist und man die optimale Lösung damit in kurzer Zeit bestimmen kann. Die Laufzeit des Algorithmus ergibt sich aus den Flussberechnungen. Da für jeden Schnitt der ausgegeben wird bis zu n Flussberechnungen nötig sind, liegt die Laufzeit des Algorithmus zur Berechnung der ℓ kleinsten Schnitte in $O(\ell n \cdot \text{maxflow}(n,m))$.

Algorithmus 8 : SimultaneousSmallCutsB

Input : Graphfamilie (G_i)
Output : Ein kleinster Schnitt S

```
1  begin
2      normalize((G_i));
3      H ← ∅ ;
4      for i = 1 to k do
6          H ← H ∪ {S, i} für einen minimalen Schnitt S in G_i;
7      M ← ∅ ;
8      repeat
10         (S, i) ← H.getMin() ;
11         M ← M ∪ {S};
12         H ← H∪ vaziraniCuts(S, i) ;
13     until ∃S ⊂ V : S ist k mal in M enthalten ;
14     return S;
15 end
```

Der Algorithmus lässt sich zur Berechnung eines kleinsten Schnitts in einer Familie von Graphen verwenden. Dazu werden, wie im Fall des Algorithmus von Nagamochi, die Graphen normalisiert. Die Schnitte aller Graphen können in einem gemeinsamen Heap gespeichert werden. Der Heap wird mit jeweils einem minimalen Schnitt aus jedem Graphen initialisiert und jeder Schnitt wird um die Information erweitert zu welchem Graphen er gehört. Für jeden Schnitt wird festgehalten, wie oft er aus dem Heap entnommen wurde. Da jeder Schnitt für jeden Graphen nur einmal in den Heap eingefügt wurde, ist eine Lösung gefunden, sobald es einen Schnitt gibt, der k mal aus dem Heap entfernt wurde (siehe Algorithmus 8).

Lemma 3.3. Algorithmus 8 berechnet einen kleinsten Schnitt S in einer Familie von Graphen mit $c(S) = \min_{\emptyset \neq S \subsetneq V} \max_{i=1,\dots k} \frac{c(S,G_i)}{\lambda}$.

Beweis. Vazirani und Yannakakis haben gezeigt, dass in ihrem Algorithmus für einen Graphen folgende Eigenschaft gilt: Zu jedem Zeitpunkt befindet sich ein Schnitt S im Heap, der minimal unter allen noch nicht aus dem Heap entnommenen Schnitten ist [VY92]. Verwendet man einen Heap H für eine ganze Familie von Graphen, so dass jeder Schnitt um die Information erweitert wurde, für welchen Graphen er im Heap steht, so muss sich zu jedem Zeitpunkt für jeden Graphen G ein Schnitt in H befinden, der minimal unter allen noch nicht entnommenen Schnitten von G ist. Denn H wird in Zeile 6 mit je einem minimalen Schnitt aus jeden Graphen initialisiert. Wird ein Schnitt in Zeile 10 entnommen, so wird durch den Algorithmus von Vazirani und Yannakakis sichergestellt, dass die Bedingung anschließend wieder erfüllt ist.

Würde man die Schleife nicht abbrechen, so würde der Original-Algorithmus und damit auch der modifizierte Algorithmus außerdem jeden Schnitt für jeden Graphen genau einmal einfügen. Daraus folgt, dass irgendwann ein erster Schnitt S zum k-ten mal aus H ent-

nommen wird. Gäbe es nun einen Schnitt T mit $\max_{i=1,\dots,k} c(T, G_i) < \max_{i=1,\dots,k} c(S, G_i)$, dann gäbe es ein $j \leq k$ mit $\max_{i=1,\dots,k} c(T, G_i) < c(S, G_j)$ und damit hätte T für alle Graphen aus H entnommen werden müssen, bevor S aus H entnommen würde. Das wäre ein Widerspruch zur Heapeigenschaft. □

Der Algorithmus von Vazirani und Yannakakis lässt sich beschleunigen, indem die bis zu n Flussberechnungen für jeden Graphen so spät wie möglich vorgenommen werden. Dazu werden zunächst alle kleinsten Schnitte ausgegeben und in einer Warteschlange W gesammelt, bevor sie für weitere Flussberechnungen verwendet werden. Aus der Warteschlange werden immer nur so viele Schnitte entnommen und Flussberechnungen durchgeführt, bis wieder ein Schnitt der gleichen Größe gefunden wurde oder die Warteschlange leer ist. Die Zeitersparnis durch die Verwendung der Warteschlange entspricht $|W| \cdot \text{maxflow}(m, n)$ Schritten, wenn W die Warteschlange bei Terminierung des Algorithmus ist.

Die Laufzeit des angepassten Algorithmus hängt stark davon ab, wie ähnlich sich die Graphen und damit die Schnitte sind. Je mehr sich die Graphen unterscheiden, umso mehr Schnitte müssen berechnet werden. Im schlimmsten Fall bedeutet das bei exponentiell vielen Schnitten eine exponentielle Laufzeit.

3.2 Heuristik

Da die Berechnung einer optimalen Lösung im schlechtesten Fall exponentiellen Zeitaufwand hat, ist eine schnelle Heuristik zur Berechnung eines kleinen Schnitts sinnvoll. Auch kann die Größe eines mit einer Heuristik gefundenen Schnitts als Schranke $(1 + \varepsilon)$ für den Algorithmus von Nagamochi et al. verwendet werden, denn diese Schranke ist ein entscheidender Parameter für die Laufzeit. Da die optimalen kleinen Schnitte häufig trivial sind, also aus nur einem Knoten bestehen, lohnt es sich vor komplexeren Berechnungen die Größe aller trivialen Schnitte in allen Graphen zu berechnen. Da für jeden solchen Schnitt nur die jeweils inzidenten Kanten eines Knotens betrachtet werden müssen liegt die Zeitkomplexität dafür in $O(k(n + m))$.

Maximumsgraph. Ein Vorschlag zur Bestimmung eines kleinen Schnitts besteht darin einen Graphen H zu einer Graphfamilie (G_i) zu konstruieren, dessen minimale Schnitte auch in (G_i) klein sind. Für einen kleinsten Schnitt S in einer normalisierten Familie von Graphen (G_i) gilt für alle $i \in \{1, \dots, k\}$ die Ungleichung $c(S, G_i) < (1 + \varepsilon)$. Der *Maximumsgraph* einer Familie von Graphen $(G_i)_{1,\dots k}$ mit $G_i = (V, E_i)$ und c_i sei der Graph $H = (V, E_H)$ mit mit

$$E_H := \bigcup_{i \in \{1,\dots k\}} E_i \text{ und } c_H(e) := \max_{i \in \{1,\dots k\}} c_i(e)$$

das heißt, der Graph, der alle Kanten einer Familie enthält mit dem maximalen Gewicht jeder Kante. Für Maximumsgraphen gilt das folgende Lemma:

Lemma 3.4. Sei $H = (V, E_H)$ der Maximumsgraph einer normalisierten Familie von Graphen (G_i). Dann ist die Größe jedes Schnitts in H eine obere Schranke für die Größe des Schnitts in den Graphen G_i.

Beweis. Sei S ein minimaler Schnitt in H. Dann gilt für jedem Graphen der Familie G_i, dass die Schnittkanten von S in G_i auch in H enthalten sind und in H kein kleineres Gewicht haben, mit anderen Worten: es gilt $E(S, G_i) \subset E(S, H)$ und

$$\forall i \leq k \forall \{u, v\} \in E(S, G_i) : c(\{u, v\}, G_i) \leq c(\{u, v\}, H).$$

Damit gilt auch $c(S, G_i) \leq c(S, H)$. □

Ist man nicht nur an einem kleinen Schnitt S, sondern auch an einer oberen Schranke für $1 + \varepsilon$ interessiert, dann lässt sich aus einem gefundenen minimalen Schnitt S in H die Schranke im Allgemeinen noch verbessern, indem man $\varepsilon = \max_{i \in \{1,\ldots,k\}} c(S, G_i) - 1$ setzt. Die Berechnung des Maximumsgraphen erfordert $O(n + km)$ Schritte, die Schnittberechnungen sind in $O(kn^2 m \log(m/n^2))$ möglich.

Nagamochi et al. Eine andere Möglichkeit einer Heuristik bietet der Algorithmus von Nagamochi in der angepassten Version mit dynamischer Schranke. Er kann als Heuristik verwendet werden, indem nicht alle kleinen Schnitte berechnet werden, sondern der Algorithmus nach einer vorgegebenen Zeit abgebrochen und der bis dahin beste Schnitt ausgegeben wird. Dabei können bevorzugt Berechnungspfade ausgewählt werden, die die Schnittgröße nicht oder nur wenig erhöhen.

3.3 Nicht-Zusammenhängende Graphen

Auf Graphen G die nicht zusammenhängend sind, ist die Problemdefinition nicht anwendbar, da $\lambda(G) = 0$ gilt und sich kleine Schnitte damit nicht über einen Faktor $(1 + \varepsilon)$ definieren lassen. Die Frage nach kleinen Schnitten in Familien, die nichtzusammenhängende Graphen enthalten, ist jedoch praxisrelevant. Betrachtet man beispielsweise Graphen, die sich über die Zeit verändern, so wird man im Allgemeinen nicht erwarten können, dass diese zu jedem Zeitpunkt oder in jedem Zeitfenster zusammenhängend sind. Neben der fehlenden Definition kleiner Schnitte, werfen mehrere Zusammenhangskomponenten auch das Problem auf, dass die Zahl minimaler Schnitte exponentiell in der Anzahl der Zusammenhangskomponenten ist.

Durch Berechnung des Maximumsgraphen und seiner Zusammenhangskomponenten, lässt sich leicht prüfen, ob es einen gemeinsamen Schnitt der Größe 0 in der Graphfamilie gibt. Ist das nicht der Fall, so kann man $\lambda(G_i) = \min_{Z \in \mathcal{Z}(G_i)} \lambda(Z)$ definieren, wobei $\mathcal{Z}(G_i)$ die Menge der durch die Zusammenhangskomponenten von G_i induzierten Graphen ist. Kleinere Werte hätten den Nachteil, dass sich die Zahl der infrage kommenden Schnitte stark erhöht.

3.4 Top-Down-Clusterung durch simultane Schnitte

Jeder Schnitt S in einer Graphfamilie (G_i) mit $G_i = (V, E_i)$ induziert zwei Teilgraphfamilien $(G_{S,i})$ und $(G_{\overline{S},i})$, mit $G_{S,i} = (S, E_{S,i})$ und $E_{S,i} = \{\{u, v\} \in E_i \mid u, v \in S\}$. Die Algorithmen lassen sich damit rekursiv auf Graphen anwenden, indem sie jeweils auf die durch einen kleinsten gefundenen Schnitt induzierten Teilgraphfamilien angewandt werden, bis sie nur noch aus einer festgelegten Anzahl κ von Knoten bestehen. Das kann man algorithmisch umsetzen, indem man die beiden durch einen Schnitt induzierten Teilgraphfamilien in eine Warteschlange einfügt, falls $|V| \geq \kappa$ gilt. Initialisiert wird die Warteschlange mit der Graphfamilie (G_i). Es werden dann so lange Graphen aus ihr entnommen und kleine Schnitte berechnet, bis sie leer ist. Damit erhält man eine rekursive Clusterung einer Graphfamilie, die die gemeinsame Struktur ihrer Graphen repräsentiert (siehe Algorithmus 9).

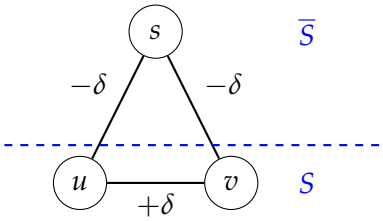
Algorithmus 9 : Clustering

Input : Graphfamilie (G_i), Schranke κ
Output : Clusterung W

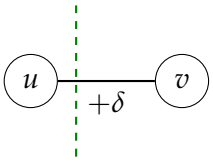
```
1  begin
2      normalize((G_i));
3      Q ← {V} ;
4      W ← ∅;
5      while Q ≠ ∅ do
6          S ← Q.getCut() ;
7          (H_i) ← durch S induzierten Teilgraphen von (G_i) ;
8          T_1 ← ein kleinster Schnitt T mit maximalen min(|T| , |S| − |T|);
9          T_2 ← S \ T_1 ;
10         if |T_1| ≥ κ then Q ← Q ∪ {T_1};
11         else W ← W ∪ T_1;
12         if |T_2| ≥ κ then Q ← Q ∪ {T_2};
13         else W ← W ∪ T_2
14     return W;
15 end
```

Da es im Allgemeinen nicht nur einen einzigen kleinsten Schnitt gibt, hat man die Freiheit ein zusätzliches Kriterium für die Auswahl eines kleinsten Schnitts zu definieren. Für eine Clusterung wäre es ein gutes Ergebnis, wenn ein Schnitt den Graphen in zwei möglichst gleich große Teile zerlegt. Daher wird in der Rekursion jeweils ein kleinster Schnitt S ausgewählt, für den $\min\{|S|, |\overline{S}|\}$ maximiert wird. Für die rekursive Berechnung unter Berücksichtigung dieses Kriteriums bietet sich Nagamochis Algorithmus an, da dieser alle kleinsten Schnitte berechnet. Der Algorithmus von Vazirani und Yannakakis könnte das nur mit zusätzlichem Zeitaufwand.

Bei der Verwendung des Algorithmus von Nagamochi stellt sich dabei die Frage, ob man die einmal berechneten Split-Operationen in den rekursiven Aufrufen wiederverwenden



(a) Die Operation $\texttt{split}(s, u, v, \delta)$ in einem Graphen G und ein Schnitt S in G.



(b) Im durch S induzierten Teilgraphen, erhöht sich die Größe von u-v-Schnitten um δ, da die ausgleichenden Kanten $\{s, u\}, \{s, v\}$ in diesem Graphen fehlen.

Abbildung 3.2: Die berechneten $\texttt{split}(s, u, v, \delta)$-Operationen können nicht in Teilgraphen wiederverwendet werden.

kann. Leider ist das nicht möglich; denn liefert ein Aufruf von Nagamochis Algorithmus den Schnitt S und gilt für eine Split-Operation $\{u, v\} \subset S$ und $\{s\} \subset \overline{S}$, so gilt für u-v-Schnitte im durch S induzierten Teilgraphen, dass diese durch die Split-Operation größer würden (siehe Abbildung 3.2). Umgekehrt bedeutet das, dass in Phase 2 die Größe von u-v-Schnitten während der Berechnung kleiner würde, womit diese nicht mehr aussortiert werden können, bevor sie vollständig berechnet sind. Das erfordert die Berechnung aller $2^{n-1} - 1$ Schnitte und ist deshalb nicht praktikabel.

4

Erzeugung von Graphen mit vorgegebener minimaler Schnittstruktur

Das simultane Schnitt-Problem für minimale Schnitte kann nur sinnvoll auf Familien von Graphen angewendet werden, in denen die Graphen jeweils über mehrere minimale Schnitte verfügen. Da es in verbreiteten Benchmarkdatensätzen keine derartigen Graphen gibt, soll in diesem Kapitel ein Generator entwickelt werden, der solche Graphen erzeugen kann. In Abschnitt 2.3 wurde bereits der Kaktus $\mathcal{R} = (W, F)$ als Repräsentation minimaler Schnitte vorgestellt, der die Struktur der minimalen Schnitte eines Graphen $G = (V, E)$ widerspiegelt. Zusammen mit einer Abbildung $\varphi : V \to W$ bildet er die Kaktus-Repräsentation. Es wird gezeigt, dass sich auf Basis von Kakteen Graphen erzeugen lassen, deren minimale Schnitte durch den vorgegebenen Kaktus repräsentiert werden. Ziel des Kapitels ist es, einen vollständigen Graphgenerator zu erhalten, der als Eingabe den Kantenzusammenhang, die Knoten- und Kantenzahl sowie gegebenenfalls weitere Parameter bekommt. Im ersten Schritt wird mit Hilfe der Parameter ein Kaktus erzeugt aus dem im zweiten Schritt ein Graph konstruiert wird (siehe Abbildung 4.1).

Der erste Abschnitt beschreibt die Konstruktion von Kakteen und im zweiten Abschnitt wird ein Graphgenerator präsentiert, der einen Graph erzeugt, dessen minimale Schnitte durch einen vorgegebenen Kaktus und eine zu bestimmende Funktion φ repräsentiert werden.

4.1 Konstruktion eines zufälligen Kaktus

In diesem Abschnitt wird ein Kaktus-Generator vorgestellt. Zu einer vorgegebenen Knotenzahl n und einer Kantenzahl m erzeugt er einen zufälligen Kaktus, indem erst

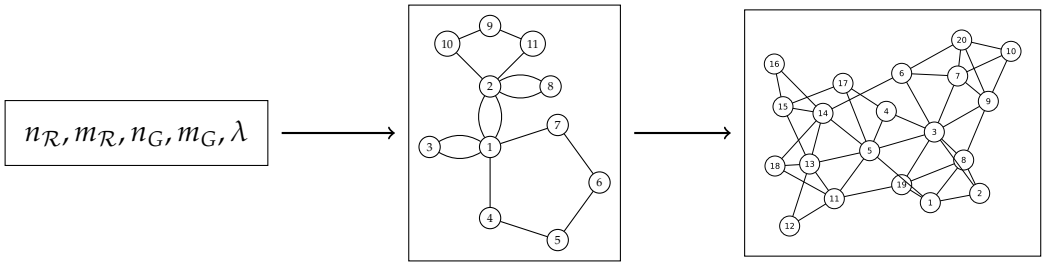


Abbildung 4.1: Das Arbeitsschema des Graphgenerators. Aus gegebenen Parametern wird erst ein Kaktus und daraus ein Graph erzeugt.

Kreise erzeugt und diese dann zu einem Kaktus zusammengesetzt werden. Die Anzahl der benötigten Kreise ergibt sich aus folgendem Lemma:

Lemma 4.1. Ein zusammenhängender Kaktus mit n Knoten und m Kanten hat genau $p = m - n + 1$ Kreise.

Beweis. Der Kaktus ist ein planarer Graph, dessen Facettenmenge aus den Kreisen und der äußeren Facette besteht. Es gilt also für die Anzahl der Facetten $f = p + 1$. Aus der Eulerformel $n - m + f = 2$ folgt damit die Behauptung. □

Die Konstruktion eines Kaktus mit p Kreisen kann somit wie folgt vorgenommen werden. Es wird ein Kreis K_1 erzeugt, der den initialen Kaktus $\mathcal{R}_1$ darstellt. Sukzessive wird dann für $i = 2, \ldots, p$ ein weiter Kreis K_i erzeugt und der Kaktus $\mathcal{R}_i$ konstruiert, indem K_i dem Kaktus $\mathcal{R}_{i-1}$ hinzugefügt wird und ein beliebiger Knoten u_i aus $\mathcal{R}_{i-1}$ mit einem beliebigen Knoten v_i aus K_i identifiziert wird (siehe Pseudocode in Algorithmus 10). Für $\mathcal{R}_i$ gilt:

Lemma 4.2. Sei $\mathcal{R} = (W, F)$ ein Kaktus und $\mathcal{R}' = (W', F')$ der Graph, der entsteht, indem zu $\mathcal{R}$ ein Kreis K hinzugefügt wird und ein Knoten $v \in K$ mit einem Knoten $u \in F$ identifiziert wird. Dann gilt: $\mathcal{R}'$ ist ein Kaktus.

Beweis. Zu zeigen ist, dass in $\mathcal{R}'$ jede Kante zu genau einem Kreis gehört. Der Kreis K enthält genau einen Verbindungsknoten in $\mathcal{R}'$, das heißt einen Knoten, der zu mehr als einem Kreis gehört. Die Kanten F' können in $\mathcal{R}'$ den gleichen Kreisen zugeordnet werden wie in $\mathcal{R}$, denn die Identifizierung von u und v lässt alle diese Kanten unverändert. Die Kanten aus K lassen sich dem neuen Kreis K zuordnen. Es gibt keinen weiteren Kreis, der sich aus Kanten aus F' bilden lässt: So ein Kreis müsste Kanten aus K und aus F' enthalten. Da es nur einen Verbindungsknoten $u = v$ in $\mathcal{R}$ gibt, kann es so einen Kreis aber nicht geben. Damit gehört jede Kante zu genau einem Kreis und $\mathcal{R}'$ ist ein Kaktus. □

Damit wurde also ein korrekter Kaktus konstruiert. Um zu zeigen, dass das Verfahren vollständig ist, wird nun umgekehrt gezeigt, dass sich jeder Kaktus in Kreise zerlegen lässt. Dazu wird folgendes Lemma für Kreise in einem Kaktus benötigt:

Algorithmus 10 : Kaktusgenerator

Input : Knotenzahl n, Kantenzahl m
Output : Kaktus $\mathcal{R}$

```
p ← m − n + 1 ;
K_1, ... K_p ← Kreise zufälliger Größe mit insgesamt m Knoten ;
R ← K_1 ;
for i = 2 to p do
    x ← zufälliger Knoten aus R ;
    u ← zufälliger Knoten aus K_1 ;
    identifiziere x mit u ;
return R ;
```

Lemma 4.3. Sei $\mathcal{R} = (W, F)$ ein zusammenhängender Kaktus, der aus mindestens zwei Kreisen besteht. Dann gibt es mindestens einen Kreis, der genau einen Verbindungsknoten enthält.

Beweis. Da $\mathcal{R}$ zusammenhängend ist, muss jeder Kreis mindestens einen Verbindungsknoten enthalten. Angenommen alle Kreise $K_1, \ldots, K_p$ in $\mathcal{R}$ hätten mindestens zwei Verbindungsknoten. Dann lässt sich ein Weg konstruieren, der bei einem beliebig gewählten Verbindungsknoten s_1 beginnt und über die Kanten eines Kreises K_1 zu einem anderen Verbindungsknoten t_1 führt. Für $i > 1$ gilt dann: Da t_{i-1} Verbindungsknoten in mindestens einem weiteren Kreis $K_i \neq K_{i-1}$ ist, lässt sich der Weg über Kanten aus K_i fortsetzen bis zu einem anderen Verbindungsknoten t_i. Dann muss es $j, k \in \{1, \ldots, |W| + 1\}$ mit $j \neq k$ geben, so dass $t_j = t_k$ und $t_i \neq t_\ell$ für $t, \ell \in \{j + 1, \ldots, k - 1\}$ gilt. Damit ist der Teilweg $(t_j, \ldots, t_k)$ ein Kreis, dessen Kanten zu verschiedenen Kreisen gehören. Das ist ein Widerspruch zur Definition des Kaktus. □

Daraus folgt, dass sich jeder Kaktus sukzessive zerlegen lässt, indem jeweils ein Kreis K mit genau einem Verbindungsknoten x abgetrennt wird, was dem umgekehrten Vorgehen bei der Konstruktion entspricht. Zu klären ist noch die Frage nach der Größe der Kreise, damit ein Kaktus mit n Knoten m Kanten und p Kreisen entsteht. Da bei der Konstruktion eines Kaktus mit m Kanten nie Kanten der hinzugefügten Kreise verändert werden und in Kreisen die Knotenzahl der Kantenzahl entspricht, gilt für die Summe der Kreisgrößen $\sum_{i=1}^{p} |K_i| = m$. Erzeugt man also zufällig p Kreise mit einer Gesamtgröße von m und identifiziert bei der Konstruktion zufällige Knoten miteinander, so ergibt sich mit Algorithmus 10 ein vollständiger Kaktusgenerator.

Eine Alternative wäre, statt einer gegebenen Knoten- und Kantenzahl die Zahl der minimalen Schnitte vorzugeben. In einem Kaktus gilt, dass für jeden seiner Kreise jeweils zwei Kreiskanten einen minimalen Schnitt induzieren. Jeder Kreis der Größe k trägt $\binom{k}{2}$ Schnitte zur Anzahl aller minimalen Schnitte bei. Für die Anzahl minimaler Schnitte $|\mathcal{C}(\mathcal{R})|$ gilt damit in Abhängigkeit von der Menge der Kreise $|\mathcal{C}(\mathcal{R})| = \sum_{K \in \mathcal{K}(\mathcal{R})} \binom{|K|}{2}$. Da ein Kreis der Größe 2 genau einen Schnitt beiträgt, kann so jede Zahl minimaler Schnitte erreicht werden. Die Anzahl minimaler Schnitte, die $\mathcal{R}$ repräsentiert, hängt allerdings

zusätzlich vom Vorkommen leerer Knoten ab, das heißt es gilt $|\mathcal{C}(G)| \leq |\mathcal{C}(\mathcal{R})|$, falls $\mathcal{R}$ die minimalen Schnitte von G repräsentiert.

4.2 Generierung von Graphen aus einem Kaktus

Ziel dieses Abschnitts ist es aus einem Kaktus $\mathcal{R} = (W, F)$ einen ungewichteten Graph $G = (V, E)$ und eine Abbildung $\varphi : V \to W$ zu konstruieren, so dass $(\mathcal{R}, \varphi)$ die Kaktus-Repräsentation der minimalen Schnitte von G ist. Parameter, die vorgegeben werden können sind die Anzahl der Knoten n und Kanten m sowie der Kantenzusammenhang λ. Dazu wird der Kaktus zunächst kopiert und daraus durch Hinzufügen und Verändern der Kanten und Knoten schrittweise ein Graph konstruiert. Dieses Vorgehen ist Thema des ersten Unterabschnitts; im zweiten Unterabschnitt werden einige Rahmenbedingungen festgelegt, die bei der Konstruktion einzuhalten sind und schließlich wird bewiesen, dass der Generator vollständig ist.

4.2.1 Der Graphgenerator

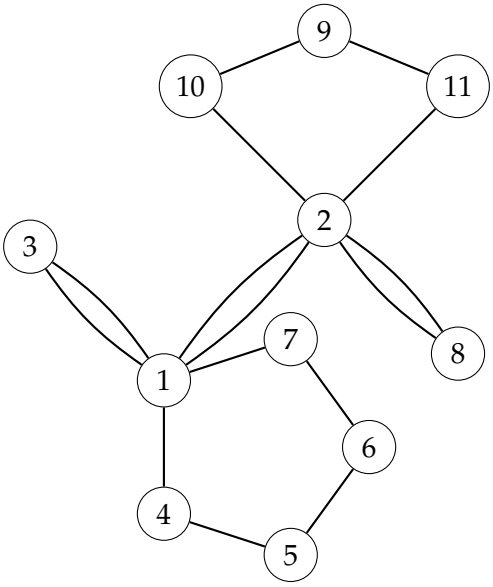
Bevor der Generator beschrieben werden kann, werden noch einige Definitionen benötigt.

Sei $G = (V, E)$ ein Graph und $(\mathcal{R} = (W, F), \varphi)$ die zugehörige Kaktusrepräsentation seiner minimalen Schnitte. Die zu einem Kaktusknoten x gehörenden Graphknoten $\varphi^{-1}(x)$ bilden eine *Komponente*. Eine Graphkante $\{u, v\} \in E$ ist entweder eine *Intrakante*, wenn sich beide Endknoten in der gleichen Komponente befinden, das heißt $\varphi(u) = \varphi(v)$ oder eine *Interkante*, wenn die Endknoten zu verschiedenen Komponenten gehören, also $\varphi(u) \neq \varphi(v)$ gilt. Interkanten sind also genau die Schnittkanten minimaler Schnitte in G. Eine Kante $\{u, v\} \in E$ *gehöre zu einem Kreis* $K \in \mathcal{K}(\mathcal{R})$, wenn K eine Kante $e \in F$ enthält, die Teil des kürzesten Pfades zwischen $\varphi(u)$ und $\varphi(v)$ im Kaktus ist. Jede Graphkante gehört also zu mindestens einem Kreis.

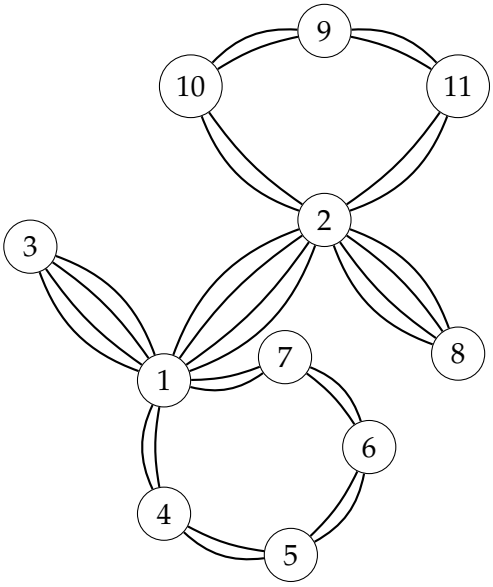
Mit dem *Verbinden* der Kanten $\{u, v\}$ und $\{v, w\}$ ist die Operation auf einem Graphen $G = (V, E)$ gemeint, die E durch $E' = E \setminus \{\{u, v\}, \{v, w\}\} \cup \{\{u, w\}\}$ ersetzt.

Jeder ungewichtete Graph mit ungeradem Kantenzusammenhang λ wird durch einen Kaktus repräsentiert, der nur Kreise der Größe 2 besitzt [FF09]. Daher muss, falls ein Graph mit ungeradem λ erzeugt werden soll, der Ausgangskaktus aus Kreisen der Größe 2 bestehen. Das wird erreicht, indem in diesem Fall bei der Kaktuskonstruktion $m = 2n - 2$ gewählt wird.

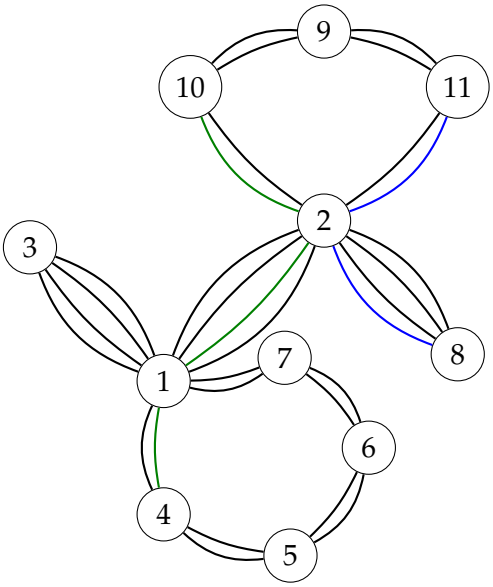
Der Graphgenerator kann nun vorgestellt werden. Er erzeugt einen zufälligen Graphen ausgehend von einem Kaktus und arbeitet in 4 Schritten. In Schritt 1 wird der Ausgangskaktus als Grundlage für den Graphen kopiert. Im zweiten Schritt werden Kanten miteinander verbunden; damit werden die Interkanten festgelegt. Der dritte Schritt fügt die gewünschte bzw. erforderliche Anzahl von Knoten hinzu. Schritt 4 fügt



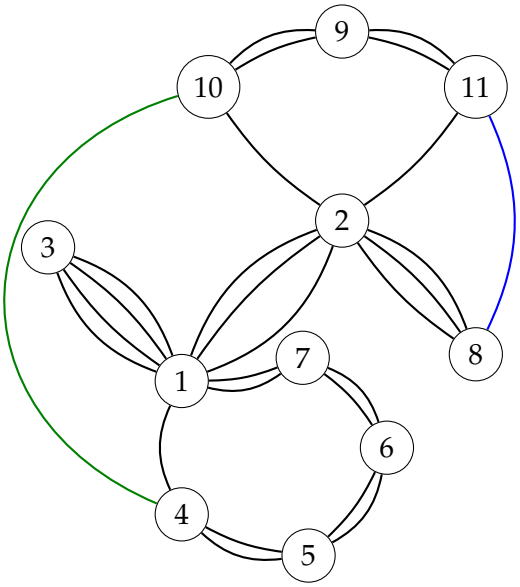
(c) Ein zufälliger Kaktus.



(d) Jede Kante wird durch $\lambda/2 = 2$ Kopien ersetzt.



(e) Es werden so lange Pfade gesucht, deren Kanten in paarweise verschiedenen Kreisen liegen ...



(f) ... und zu einzelnen Kanten zusammengefügt bis die gewünschte Interkantenzahl erreicht ist.

Abbildung 4.2: Graphgenerator: Erzeugung der Interkanten für $\lambda = 4$.

schließlich die Intrakanten ein. Dabei gilt jeweils nach den Schritten 1, 2 und 4, dass der Ausgangskaktus die minimalen Schnitte des Graphen G repräsentiert.

Schritt 1. Ausgangspunkt ist ein Graph G, der die gleiche Knotenmenge wie der gegebene Kaktus $\mathcal{R} = (W, F)$ enthält und $\frac{\lambda}{2}$ Kopien der Kanten, das heißt $G = (V, E)$ mit $V = W$ und der Multimenge $E = \frac{\lambda}{2} \cdot F$ (jedes Element aus F kommt $\frac{\lambda}{2}$ mal in E vor). Da für Kakteen, aus denen Graphen mit ungeradem λ erzeugt werden sollen, verlangt ist, dass sie nur aus Kreisen der Größe 2 bestehen, ist das auch für ungerade λ möglich; in diesem Fall werden jeweils zwei Kaktuskanten durch λ Kanten ersetzt. Damit erhält man bereits einen Graphen G, der durch $(\mathcal{R}, \varphi)$ repräsentiert wird und den vorgegebenen Kantenzusammenhang λ hat, wobei $\varphi : V \to W$ die Identität ist (siehe Abbildungen 4.2c und 4.2d). Eine Komponente im Graph G besteht anfangs also aus jeweils einem Knoten. Dieser Graph enthält allerdings noch Mehrfachkanten; diese werden im Laufe des Verfahrens noch beseitigt.

Schritt 2. Es werden so lange zwei inzidente Kanten in G miteinander verbunden, bis die gewünschte Anzahl Kanten übrig bleibt (siehe Abbildungen 4.2e und 4.2f). Dabei müssen bestimmte Bedingungen eingehalten werden; diese werden später in Lemma 4.4 definiert. Zunächst werden durch das Verbinden von Kanten leere Knoten hergestellt, indem in einem Schritt alle zu einem Graphknoten u inzidenten Kanten paarweise miteinander verbunden werden und u anschließend entfernt wird. Anschließend werden zufällig ausgewählte Kanten verbunden.

Schritt 3. Nachdem alle Interkanten erzeugt wurden, werden nun Knoten eingefügt. Es werden erst gezielt Knoten dort eingefügt, wo mehrere Kanten zwei Komponenten $\varphi^{-1}(x), \varphi^{-1}(y)$ mit $x, y \in W$ verbinden, um anschließend Mehrfachkanten vermeiden zu können. Es werden so lange zufällig in einer dieser Komponenten Knoten hinzugefügt bis $|\varphi(x)| \cdot |\varphi(y)| \geq |\{\{u, v\} \in E \mid \varphi(u) = x \wedge \varphi(v) = y\}|$ gilt. Weitere Knoten werden schließlich in zufälligen Komponenten eingefügt, bis die gewünschte Mindestknotenzahl erreicht ist und bestimmte Bedingungen eingehalten werden, die in Lemma 4.7 präzisiert werden. Die Knotenmenge V wird durch $V' = V \cup \{v\}$ mit dem neuen Knoten v ersetzt und die Abbildung φ wird ergänzt durch $v \mapsto x$, mit zufälligem $x \in W$.

Die Interkanten werden dabei zufällig auf die Knoten der jeweiligen Komponente verteilt. Das heißt die Kantenmenge E wird ersetzt durch $E' = \bigcup_{\{u_1, v_1\} \in E} \{\{u_2, v_2\}\}$, wobei $u_2 \in \varphi^{-1}(\varphi(u_1)), v_2 \in \varphi^{-1}(\varphi(u_1))$ zufällig gewählt werden. In bestimmten Fällen muss durch Interkanten der Kantenzusammenhang in den Komponenten erhöht werden; dann muss die Verteilung so vorgenommen werden, dass die Anzahl der inzidenten Interkanten von zwei Knoten einer Komponente maximal um 1 voneinander abweicht.

Schritt 4. Schließlich werden die Intrakanten eingefügt. Dazu wird zunächst ein Kantenzusammenhang von $\lambda + 1$ innerhalb der Komponenten hergestellt. Das erfolgt in drei Teilschritten.

(a) Erst wird der Knotengrad jedes Knotens auf mindestens $\lambda + 1$ erhöht. Erreicht wird das, indem solange ein Knoten v mit $\deg(v) < \lambda + 1$ gewählt wird und mit einem

zufälligen zweiten Knoten aus der gleichen Komponente $\varphi^{-1}(\varphi(v))$ verbunden wird, bis alle Knoten den Mindestgrad haben.

(b) Im zweiten Teilschritt wird eine Kaktusrepräsentation $(\mathcal{R}', \varphi')$ von G berechnet und überprüft, ob alle Knoten einer Komponente auch im gleichen Kaktusknoten liegen, das heißt, ob gilt $\forall u, v \in V : \varphi(u) = \varphi(v) \iff \varphi'(u) = \varphi'(v)$. Ist das für zwei Knoten u, v mit $\varphi(u) = \varphi(v)$ nicht der Fall, so werden so lange zufällig Knoten aus den jeweiligen $\mathcal{R}'$-Komponenten $\varphi'^{-1}(\varphi'(u))$ und $\varphi'^{-1}(\varphi'(v))$ durch eine Kante verbunden, bis diese Bedingung erfüllt ist. Dabei wird der Kaktus $\mathcal{R}'$ nach jeder Kanteneinfügeoperation mit Hilfe der dynamischen Kaktus-Operationen aktualisiert. Teilschritt (b) wird wiederholt bis schließlich die Knoten jeder Komponente in jeweils einem Kaktusknoten liegen.

(c) Es werden weitere Kanten zwischen zufällig gewählten Knoten der gleichen Komponente eingefügt, bis die gewünschte Kantenzahl erreicht ist.

4.2.2 Rahmenbedingungen und Vollständigkeit

In diesem Abschnitt sollen nun die noch fehlenden Bedingungen definiert werden, die beim Verbinden von Kanten oder Einfügen von Knoten einzuhalten sind. Es wird bewiesen, dass diese Bedingungen hinreichend und notwendig sind, um einen korrekten Graphen zu erhalten. Schließlich wird gezeigt, dass der Generator vollständig ist.

Lemma 4.4. Sei $G = (V, E)$ ein ungewichteter Graph und $G' = (V, E')$ der Graph, der aus G entsteht, indem zwei adjazente Kanten $\{u, v\}, \{v, w\} \in E$ miteinander verbunden werden, so dass gilt $E' = E \setminus \{\{u, v\}, \{v, w\}\} \cup \{u, w\}$. Dann ist die Kaktus-Repräsentation $(\mathcal{R}, \varphi)$ der minimalen Schnitte in G genau dann auch die Kaktus-Repräsentation von G', wenn folgende Bedingungen eingehalten werden:

(1) $\{u, v\}$ und $\{v, w\}$ gehören zu unterschiedlichen Kreisen.

(2) $\deg(v, G) \geq \lambda + 3$

(3) Für einen Verbindungsknoten x, der p Kreise verbindet, muss für je $k \leq p$ Kreise $K_1, \ldots, K_k$ in $\mathcal{R}$, die den Verbindungsknoten x gemeinsam haben, gelten: Es darf höchstens $\frac{(k-1)\cdot\lambda}{2} - 1$ Kanten geben, die diese k Kreise verbinden, das heißt Kanten $\{u, v\}$, für die gilt: $\varphi(u) \in K_i \setminus \{x\}, \varphi(v) \in K_j \setminus \{x\}, i \neq j$.

Beweis. Seien $x = \varphi(u), y = \varphi(v), z = \varphi(w) \in W$ die Kaktusknoten in $\mathcal{R}$ sowie $e_1 = \{u, v\}$ und $e_2 = \{v, w\} \in E$ die Kanten, die verbunden werden. Da alle Schnitte, die $\{u, w\}$ in G' schneiden, entweder e_1 oder e_2 in G schneiden, können Schnitte durch die Operation nicht größer werden.

Verbindet man zwei Kanten e_1 und e_2, die zu einem Kreis K' gehören, so muss es auch einen Kreis K geben zu dem beide Kanten gehören und der y enthält, da der kürzeste Weg von K' zu K für beide Kanten über die gleichen Kreise führen muss. Dann gilt für den minimalen Schnitt in G, der durch die beiden zu y inzidenten Kaktuskanten des Kreises K induziert wird, dass er durch das Verbinden von e_1 und e_2 zwei Schnittkanten weniger

hätte. Damit würde dieser Schnitt kleiner, während Schnitte, die u und w trennen, gleich groß blieben. Damit ist gezeigt, dass Bedingung (1) notwendig ist.

Bedingung (2) ist notwendig, da sonst ein neuer minimaler Schnitt $\{v\}$ entstünde.

Falls $\frac{(k-1)\cdot\lambda}{2}$ Kanten zwischen k Kreisen liegen, dann bilden diese k Kreise zusammen einen neuen Schnitt. Damit ist die Bedingungen (3) nur $\frac{k-1}{2} - 1$ Kanten zwischen k Kreisen zu erlauben, notwendig, um sicherzustellen, dass keine neuen Schnitte entstehen.

Die Bedingungen sind auch hinreichend. Denn wegen Bedingungen (2) können keine neuen minimalen Schnitt entstehen, die eine Komponente aufspalten, denn außer $\varphi(v)$ gibt es keine Komponente deren Kantenzusammenhang verändert wurde. Durch das Verbinden von e_1 und e_2 können nur Schnitte kleiner werden, die $S_1 = \varphi^{-1}(\{x, z\})$ und $S_2 = \varphi^{-1}(y)$ trennen. In G gibt es keine minimalen Schnitte, die S_1 und S_2 trennen, da es nach Bedingung (1) keinen Kreis gibt zu dem e_1 und e_2 gehören; nur ein solcher Kreis in $\mathcal{R}$ könnte aber so einen Schnitt repräsentieren. Ein minimaler Schnitt in G' müsste also aus Schnittkanten bestehen, die auch in G Schnittkanten waren. Da die einzigen Kaktuskanten, zu denen e_1 und e_2 gehörten und zu denen e nicht gehört, die Kanten sind, die zu y inzident sind, muss es einen neuen minimalen Schnitt in G' geben, der durch solche Kanten repräsentiert wird, falls es neue minimale Schnitte in G' gibt. Sei nun S_K der Schnitt in G', den die beiden zu $\varphi(v)$ inzidenten und zum Kreis K gehörenden Kanten induzieren. Da Graphkanten nur zwischen Knoten liegen können, die zu verschiedenen Kreisen gehören, müsste ein neuer Schnitt aus der Vereinigung mehrerer solcher S_K bestehen. Seien K_1 und K_2 die Kreise, die $\varphi(u)$ bzw. $\varphi(w)$ enthalten. Angenommen es gäbe einen neuen minimalen Schnitt T, dann müsste er S_{K_1}, S_{K_2} und eventuell weitere Schnitte $S_{K_3}, \ldots S_{K_p}$ enthalten, also $T = \bigcup_{i=1}^{k} S_{K_i}$ gelten (siehe Abbildung 4.3). Da für jedes $i < k$ gilt $c(S_{K_i}) = \lambda$ und auch $c(T) = \lambda$ erfüllt wäre, müssten von den $k\lambda$ Schnittkanten aller S_{K_i} insgesamt $(k-1)\lambda$ Kanten doppelt vorkommen, also zwei Knoten aus verschiedenen dieser Schnitte verbinden. Das ist ein Widerspruch zur Bedingung (3) aus Lemma 4.4.

Die formulierten Bedingungen sind also notwendig und hinreichend. □

Nachdem die Bedingungen für das Verbinden von Kanten geklärt sind, muss noch der Fall leerer Kaktusknoten untersucht werden.

Lemma 4.5. Sei $G = (V, E)$ ein Graph und $\mathcal{R} = (W, F)$ seine Kaktusrepräsentation. Dann gilt für alle leeren Knoten $x \in W$, dass x ein Verbindungsknoten ist und, falls er nur zu zwei Kreisen gehört, keiner dieser Kreise die Größe 2 hat.

Beweis. Ein leerer Knoten x muss Verbindungsknoten sein, denn sonst wäre $\{x\}$ in $\mathcal{R}$ ein minimaler Schnitt, aber $\varphi^{-1}(x)$ wäre kein Schnitt in G, was ein Wiederspruch zur Definition der Kaktusrepräsentation ist. Dass leere Knoten nicht zu zwei Kreisen gehören dürfen, von denen ein Kreis die Größe 2 hat, wird in der Definition der Kaktusrepräsentation gefordert (siehe Abschnitt 2.3). □

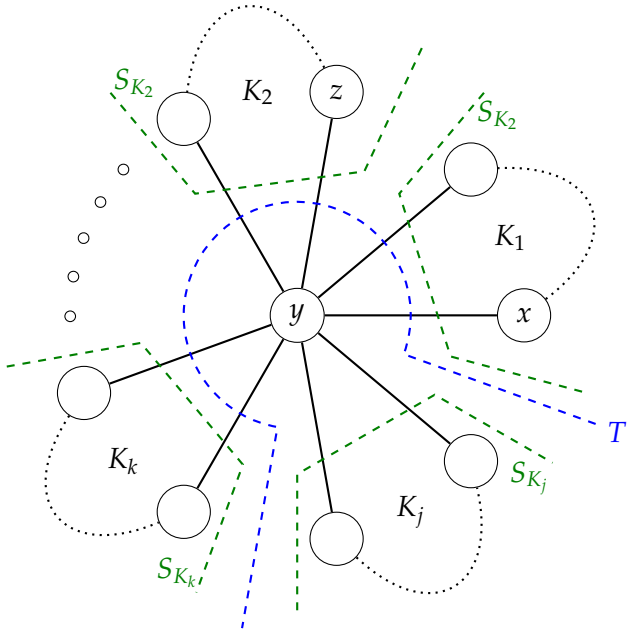


Abbildung 4.3: Alle S_{K_i} sind minimale Schnitt der Größe λ. Wäre auch $T = \bigcup_{i=1}^{k} S_{K_i}$ ein Schnitt dieser Größe, so müssten $(k-1)\lambda$ Kanten je zwei Knoten aus verschiedenen S_{K_i} verbinden. Der Kreis K_j steht stellvertretend für einen oder mehrere Kreise, die nicht in T enthalten sind.

In Schritt zwei werden als erstes leere Knoten erzeugt. Dazu muss die Anzahl der Knoten, die leer sein müssen bekannt sein. Dazu benötigt man folgendes Lemma über den Zusammenhang von Interkantenzahlen und leeren Knoten.

Lemma 4.6. Für die Interkantenzahl m_{inter} in einem Graphen $G = (V, E)$ mit Kaktus-Repräsentation $(\mathcal{R}, \varphi)$, $\mathcal{R} = (W, F)$ und $m_{\mathcal{R}} = |F|$ gilt:

$$\frac{\lambda}{2}\left(m_{\mathcal{R}} - |Z(\mathcal{R})| - \sum_{x \in L(\mathcal{R})} \deg(x)/2\right) + |Z(\mathcal{R})| \leq m_{\text{inter}},$$

wobei $L(\mathcal{R})$ die Menge der Knoten ist, die leer werden kann und $Z(\mathcal{R})$ die Menge der 2-Verbindungsknoten mit mindestens einem Kreis der Größe 2 ist.

Beweis. Es wird die Menge der Verbindungsknoten betrachtet, da nur an Verbindungsknoten Kanten verbunden werden können. Sie ist die disjunkte Vereinigung der 2-Verbindungsknoten mit mindestens einem Kreis der Größe 2, $Z(\mathcal{R})$, und den Verbindungsknoten, die leer werden können $L(\mathcal{R})$. Jede Komponente $\varphi^{-1}(x)$ mit Knoten x aus $L(\mathcal{R})$ ist inzident zu höchstens $\deg(x)\lambda$ Kanten; die Kantenzahl kann hier also um $\sum_{x \in L(\mathcal{R})} \frac{\deg(x)\cdot\lambda}{4}$ Kanten verringert werden. Knoten aus $Z(\mathcal{R})$ verringern wegen Bedingung 2 aus Lemma 4.4 die Kantenzahl jeweils um $\lambda/2 - 1$ Kanten. □

Mit Hilfe dieses Lemmas und der vorgegebenen Interkantenzahl lässt sich dann die Anzahl an Knoten bestimmen, die mindestens leer sein müssen.

Es muss noch sichergestellt werden, dass innerhalb einer Komponente $\varphi^{-1}(x)$ der Kantenzusammenhang $\lambda + 1$ erreicht werden kann. Dabei muss beachtet werden, dass nicht nur Intrakanten, sondern auch Interkanten den Kantenzusammenhang erhöhen können. Insbesondere gilt, dass der Knotengrad jedes Komponentenknoten einer Komponente mit $\left|\varphi^{-1}(x)\right| > 1$ größer oder gleich $\lambda + 1$ sein muss. Aus dieser Überlegung ergibt sich folgendes Lemma:

Lemma 4.7. Sei $\mathcal{R} = (W, F)$ die Kaktusrepräsentation eines ungewichteten Graphen $G = (V, E)$ und $x \in W$ ein Kaktusknoten. Bezeichne $\ell(x) = \left|\varphi^{-1}(x)\right|$ die Anzahl der Komponentenknoten in $\varphi^{-1}(x)$ und $c(S)$ die Größe eines Schnitts S. Dann gilt folgende Ungleichung:

$$\left\lceil \frac{(\lambda(G)+1)\cdot \ell(x) - c(\varphi^{-1}(x))}{2} \right\rceil \leq \binom{\ell(x)}{2} \text{ oder } \ell(x) \leq 1$$

und damit die folgende untere Schranke für $\ell(x)$:

$$\ell(x) \geq \left\lceil \sqrt{\left(\frac{\lambda(G)+2}{2}\right)^2 - c(\varphi^{-1}(x))} + \frac{\lambda(G)}{2} + 1 \right\rceil \text{ oder } \ell(x) \leq 1$$

Beweis. Jeder Knoten einer Komponente mit $\ell(x) > 1$ muss mindestens Grad $\lambda + 1$ haben. Daraus folgt, dass die Summe der Knotengrade aller Komponentenknoten $\varphi^{-1}(x)$ mindestens $(\lambda(G) + 1) \cdot \ell(x)$ sein muss. Jede Interkante erhöht die Knotengradsumme um 1. Die Differenz muss durch Intrakanten, die jeweils 2 zur Knotengradsumme beitragen, gedeckt werden. Maximal sind $\binom{\ell(x)}{2}$ Intrakanten möglich, die die Knotengradsumme jeweils um 2 erhöhen. Damit gilt also die Ungleichung. Aufgelöst nach $\ell(x)$ ergibt sich damit die untere Schranke. □

Lemma 4.7 liefert damit eine untere Schranke für die Anzahl der Knoten in einer Komponente. Mit diesen Bedingungen erhält man insgesamt den Algorithmus 11. Im Folgenden wird gezeigt, dass das Verfahren vollständig ist.

Lemma 4.8. Das Verfahren ist vollständig, das heißt aus einem gegebenen Kaktus kann jeder Graph erzeugt werden, dessen minimale Schnitte er repräsentiert.

Beweis. Der Beweis wird geführt, indem für einen beliebigen Graphen G gezeigt wird, dass dieser durch das beschriebene Verfahren konstruiert werden kann. Dazu werden die einzelnen Schritte rückwärts betrachtet und gezeigt, dass der Graph durch umgekehrte Anwendung der einzelnen Schritte in einen Kaktus überführt werden kann. Dabei wird mit $G_t = (V_t, E_t)$ der Graph bezeichnet, der durch umgekehrte Anwendung von Schritt t entsteht.

Sei also $G = (V, E)$ ein beliebiger Graph. Schritt 4c fügt zufällige Intrakanten ein, die nicht für den Kantenzusammenhang notwendig sind. Sei G_{4c} der Graph aus dem sukzessive so lange wie möglich Kanten mit diesem Kriterium entfernt wurden. Intrakanten

Algorithmus 11 : Graphgenerator

Data : Kaktus $\mathcal{R} = (W, F)$, Interkantenzahl m_{inter}, Intrakantenzahl m_{intra} Knotenzahl n
Result : Graph $G = (V, E)$

$(V, E) \leftarrow (W, \lambda/2 \cdot F)$;
while *weitere leere Knoten notwendig, um geforderte Kantenzahl zu erreichen* **do**
    leereZufälligenKnoten() ;
while $|E| > m_{inter}$ **do**
    verbindeZufälligeKanten() ; `// Bedingungen werden eingehalten`
while $\exists x, y : |\varphi(x)| \cdot |\varphi(y)| < |\{\{u, v\} \in E | \varphi(u) = x \wedge \varphi(v) = y\}|$ **do**
    $v =$ addnewNode();
    $\varphi(v) = random(x, y)$;
while $|V| < n \vee$ *Mindestknotenzahl nicht erreicht* **do**
    $v =$ addnewNode();
    $\varphi(v) = random(V)$;
$E \leftarrow \{\{u_2, v_2\} | \{u_1, v_1\} \in E, u_2 \in \varphi^{-1}(\varphi(u_1)), v_2 \in \varphi^{-1}(\varphi(u_1))\}$;
while $\exists v \in V : \deg(v) < \lambda + 1$ **do**
    $v \leftarrow random(V, \deg(v) < \lambda + 1)$;
    $u \leftarrow random(\varphi^{-1}(\varphi(v)))$;
    addEdge(u,v);
do
    $\mathcal{R} \leftarrow$ berechneKaktus(G); `// Berechne Kaktus, um Kantenzusammenhang`
                                  `// zu überprüfen und ggf. zu korrigieren`
    $b \longleftarrow = false$;
    while $\exists u, v \in V : \varphi(u) = \varphi(v) \nLeftrightarrow \varphi'(u) = \varphi'(v)$ **do**
        $w \leftarrow random(\varphi'^{-1}(\varphi'(u)))$;
        $x \leftarrow random(\varphi'^{-1}(\varphi'(v)))$;
        addEdge(w,x); `// nutze dynamischen Kaktus`
        $b \leftarrow true$;
while b ;
while $m_{G,intra} < m_{intra}$ **do**
    $x \leftarrow random(W)$;
    $u \leftarrow random(\varphi^{-1}(W))$;
    $v \leftarrow random(\varphi^{-1}(W) \setminus \{u\})$;
    addEdge(u,v);

$\{u, v\} \in E_{4c}$ mit $\deg(u) > \lambda + 1$ und $\deg(v) > \lambda + 1$ können sukzessive durch rückwärtige Anwendung von Schritt 4b entfernt werden, denn diese Kanten müssen den Kantenzusammenhang erhöhen und erfüllen damit das Kriterium von Schritt 4b. Alle Intrakanten in G_{4b} müssen nun mindestens einen Knoten v mit $\deg(v) < \lambda + 1$ haben, diese können alle durch Umkehrung von Schritt 4a entfernt werden. Damit gibt es keine Intrakanten mehr in G_{4a}.

Alle Interkanten werden in Schritt 3 von einem Knoten zufällig auf alle Knoten einer Komponente verteilt. Da von der zufälligen Wahl nur abgewichen wurde, wenn die Interkante zur Herstellung des Kantenzusammenhangs erforderlich ist, bedeutet dies keine Einschränkung, die die Konstruktion des Graphen G_{4a} unmöglich machen würde. Wählt man in jeder Komponente einen Knoten, so können durch umgekehrte Anwendung von Schritt 3 die Endknoten aller Interkanten ersetzt werden durch diese ausgewählten Knoten. Entfernt man so lange alle Knoten, so dass keine der Bedingungen aus Lemma 4.7 oder $|\varphi(x)| \cdot |\varphi(y)| \geq |\{\{u, v\} \in E \mid \varphi(u) = x \wedge \varphi(v) = y\}|$ verletzt wird, so entspricht das den zufällig eingefügten Knoten in Schritt 3. Alle anderen Knoten lassen sich durch mindestens eine der beiden Bedingungen begründen, da beide Bedingungen notwendig sind. Damit erhält man den Graphen G_3.

Jede Kante $\{u, v\}$ in G_3, für die gilt, dass $\varphi(u)$ und $\varphi(v)$ im zugehörigen Kaktus nicht benachbart sind, muss zu mehreren Kreisen gehören. Kanten, die zu mehreren Kreisen gehören, können durch umgekehrte Anwendung von Schritt 2 getrennt werden. Da gezeigt wurde, dass alle Bedingungen aus Lemma 4.4 notwendig sind kann es keine Bedingung geben, die verhindern würde, dass die Kante in Schritt 2 nicht verbunden werden könnte. Leere Knoten werden gegebenenfalls entsprechend Schritt 2 durch einen neuen Knoten ersetzt. Der Graph G_2 enthält damit nur noch Kanten $\{u, v\}$ mit $\{\varphi(u), \varphi(v)\} \in F$, wenn $\mathcal{R} = (W, F)$ der zugehörige Kaktus ist.

Damit enthält G_2 nur noch Kanten, die auch im Kaktus vorkommen. Das müssen genau $\lambda/2$ Kopien der Kaktuskante sein, denn sonst hätte der Graph nicht den Kantenzusammenhang λ. Diese lassen sich durch jeweils eine Kante ersetzen, was der umgekehrten Anwendung von Schritt 1 entspricht. G_1 muss damit ein Kaktus sein. □

5

Experimentelle Auswertung

In diesem Kapitel werden die entwickelten Algorithmen getestet. Dabei sollen die Laufzeiten der verschiedenen Algorithmen betrachtet werden und festgestellt werden, inwieweit sich die dynamische Kaktus-Repräsentation gegenüber einer Neuberechnung lohnt. Außerdem werden die Ergebnisse qualitativ untersucht und verglichen. Alle Algorithmen wurden in Java 6 unter Verwendung der Bibliothek yFiles [yWo] implementiert. Die Experimente wurden auf einem Kern eines Intel Xeon E5430 mit 2.66 GHz und 32 GB RAM durchgeführt. Soweit nicht anders angegeben beruht jeder Datenpunkt auf dem Durchschnittswert von 10 unabhängig erzeugten Graph(familien) mit den jeweiligen Parametern.

5.1 Testgraphen

In diesem Abschnitt werden Generatoren von Familien von Graphen betrachtet. Diese Graphfamilien sollen aus verschiedenen Graphen bestehen, die sich hinsichtlich ihrer Schnitte ähnlich sind.

Zeitabhängige Graphen. Ein Typ von Graphfamilien sind zeitabhängige Graphen. Sie sollen den dynamischen Fall widerspiegeln, in dem sich die Kanten eines Graphen mit der Zeit verändern. Zwei aufeinander folgende Graphen einer Familie sollen demnach nur geringe Unterschiede haben, während beispielsweise der erste und der letzte Graph einer solchen Familie nicht mehr viele Kanten gemeinsamen haben müssen. Die Unterschiede sollen mit Hilfe folgender Definition ausgedrückt werden: Für zwei Graphen

$G = (V, E_G), H = (V, E_H)$ sei die *Distanz* $\text{dist}(G, H)$ definiert als die Anzahl der Kanten, die in genau einem der beiden Graphen vorkommen, das heißt

$$\text{dist}(G, H) = |E_G \setminus E_H| + |E_H \setminus E_G|$$

Die Distanz zwischen zwei Graphen G und H entspricht damit der Anzahl der notwendigen Einfüge- und Löschoperationen von Kanten um aus G den Graphen H zu konstruieren.

Bei der Konstruktion einer zeitabhängigen Graphfamilie soll ein beliebiger Graph G Ausgangpunkt sein. Die weiteren Graphen werden dabei generiert, indem ausgehend von $G_1 = G$, Kanten hinzugefügt oder gelöscht werden und so sukzessive die Graphfamilie berechnet wird. Das heißt, dass der Graph G_{i+1} durch Einfügen und Löschen von Kanten aus G_i erzeugt wird. Durch diese schrittweise Veränderung gibt es eine große Ähnlichkeit zwischen aufeinanderfolgenden Graphen wie man sie beispielsweise im dynamischen Fall zu aufeinanderfolgenden Zeitpunkten erwarten würde. Geht man von einem gegebenen Graphen G_1 und einer festgelegten Distanz d aus, so kann die Familie erzeugt werden, indem für jeden weiteren Graphen der Familie d mal zufällig entweder eine Kante eingefügt oder entfernt wird. Im Falle des Einfügens wird die neue Kante zwischen zwei zufällig gewählten Knoten erzeugt. Anderenfalls wird eine zufällig gewählte Kante entfernt.

Konkret werden in den Tests zwei Typen von zeitabhängigen Graphen verwendet

- Bei *Typ-ZK*-Familien (Zeitabhängig, Kaktus) wurden der erste Graph G_1 aus einem zufälligen Kaktus erzeugt. Für $i > 1$ wurde der Graph G_i aus dem Graphen G_{i-1} durch 5 zufällige Kanten-Einfüge- oder Lösch-Operationen konstruiert.
- Graphfamilien vom *Typ ZZ* (Zeitabhängig, Zufall) wurden analog erzeugt ausgehend von einem zufälligem Graphen ohne vorgegebene Schnittstruktur.

Zeitunabhängige Graphen. Betrachtet werden sollen auch Graphen die zeitunabhängig sind. Diese Familien von Graphen teilen sich die gleiche Knotenmenge, werden aber unabhängig voneinander erzeugt:

- Alle Graphen einer *Typ-UK*-Familie (Unabhängig, Kaktus) wurden aus dem gleichen zufälligen Kaktus erzeugt. Die Knotennummern, mit denen gleiche Graphknoten in verschiedenen Graphen identifiziert werden, wurden dabei unabhängig voneinander gewählt.
- Familien vom *Typ UZ* (Unabhängig, Zufall) bestehen aus völlig verschiedenen Graphen, die nur die Knoten- und Kantenzahl gemeinsam haben.

Interessant sind aber auch Familien, deren Graphen sich untereinander ähnlich sind, aber nicht zeitabhängig sind. Um sie zu erzeugen wird eine Ordnung $(x_1, \ldots, x_n)$ für die Kaktusknoten bestimmt. Für jeden Graphen werden nacheinander für $i = 1, \ldots, n$ die Komponenten $\varphi^{-1}(x_i)$ benannt. Die Reihenfolge der Knoten innerhalb einer Komponente ist damit nicht bestimmt, insgesamt gilt aber, dass die Schnitte, die durch den gleichen Schnitt im Kaktus induziert werden, eine ähnliche Knotenmenge in allen Graphen der

Familie repräsentieren. Allerdings enthalten die Graphen aufgrund der unabhängigen Erzeugung völlig unterschiedliche Kantenmengen. Damit vervollständigt sich die Liste der Graphfamilien-Typen:

- Bei *Typ-GK*-Familien (Gleicher Kaktus) werden alle Graphen einer Familie aus dem gleichen zufälligen Kaktus erzeugt. Die Knotennummern wurden hier aber in der gleichen Reihenfolge gewählt.

E-Mail-Graph-Familie Eine weitere verwendete Testgraphfamilie besteht aus Graphen, die die E-Mail-Kommunikation zwischen Mitarbeitern der Fakultät für Informatik abbilden. Dabei repräsentieren die Knoten Mitarbeiter und die gewichteten Kanten repräsentieren die Anzahl an E-Mails zwischen zwei Mitarbeitern. Erzeugt wurde daraus eine Familie aus 9 Graphen, in denen jeder Graph den E-Mail-Verkehr eines Tages abdeckt.

5.2 Dynamischer Kaktus

In diesem Abschnitt wird die dynamische Kaktus-Repräsentation untersucht.

Zunächst soll getestet werden, inwieweit sich die simultane Schnittberechnung bei einer gegebenen Graphfamilie lohnt. Dazu wurden Typ-ZK-Familien aus jeweils 5 Graphen mit 1000 Knoten und unterschiedlichem Kantenzusammenhang λ als Testfamilien verwendet, da diese Familie die notwendige Ähnlichkeit der Graphen untereinander sicherstellt und die Graphen mehrere minimale Schnitte haben. Es wurde dabei keine Kantenzahl vorgegeben, sondern so viele Kanten verwendet, wie für einen Graph mit der vorgegebenen Kaktus-Struktur notwendig sind. Die Zahl der Kanten verhält sich proportional zu λ.

Die Ergebnisse des Tests sind in Abbildung 5.1 zu sehen. Es ergibt sich bei allen λ ein deutlicher Vorteil für die Berechnung mit Hilfe des dynamischen Kaktus. Die einzelne Berechnung der Kakteen benötigt hier zwischen $3,1$ und $3,6$ mal so viel Zeit, wie im dynamischen Fall.

Interessant ist auch die Frage, wie ähnlich sich Graphen sein müssen, damit sich eine simultane Berechnung überhaupt lohnt. Dazu soll getestet werden, wie viele Aktualisierungen des Kaktus in der Zeit einer Neuberechnung möglich sind. Das Ergebnis ist in Abbildung 5.2 dargestellt. Im Fall der Typ-ZK-Graphen ergibt sich bei allen getesteten Knotenzahlen und Kantenzusammenhängen etwa ein Wert von $0,3$, das heißt in der Zeit einer Kaktusberechnung sind $0,3n$ Updates möglich.

Bei den Typ-ZK-Graphen wurde statt dem Kantenzusammenhang die Anzahl der Kanten vorgegeben. Als Mindestkantenzahl wurde $5n$ gewählt, da bei diesem Verhältnis von Kanten zu Knoten mit hoher Wahrscheinlichkeit ein zusammenhängender Graph erzeugt wird. Als Höchstkantenzahl wurde $5n + n^2/4$ gewählt, was einem Graphen entsprich, in dem etwa die Hälfte aller Knotenpaare durch eine Kante verbunden ist. Die Messwerte liegen hier zwischen $0,2$ und $0,8$; sie variieren also stärker, als bei Typ-ZK-Familien.

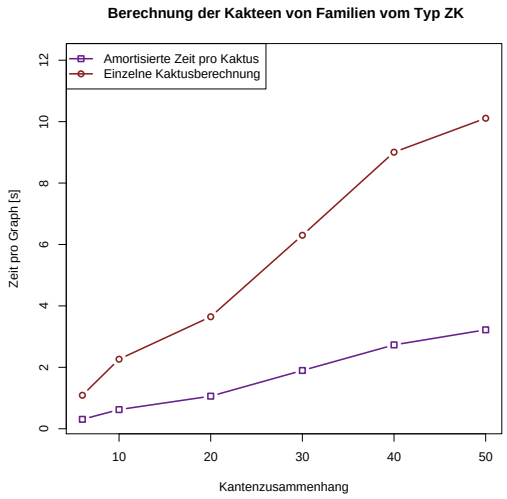


Abbildung 5.1: Zeit pro Graph für die Berechnung einer Familie von Kakteen für Typ-ZK-Familien mit jeweils 5 Graphen.

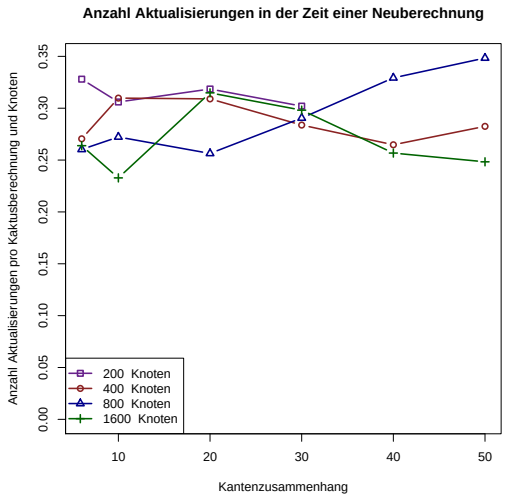
Doch auch hier zeigt sich, dass die Anzahl der Aktualisierungen etwa linear in n ist; man spart also einen linearen Faktor gegenüber einer Neuberechnung.

5.3 Vergleich minimaler Schnitte

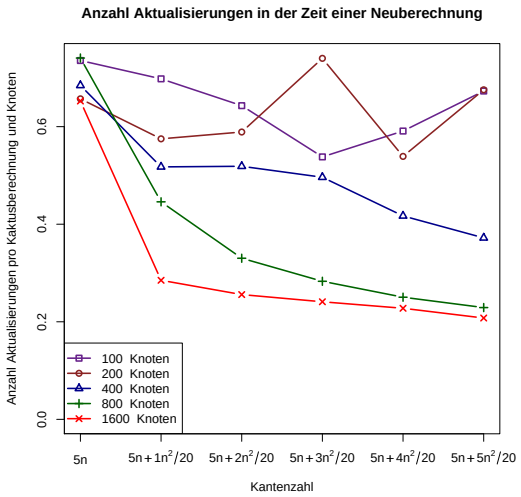
In diesem Abschnitt wird die benötigte Zeit für den Vergleich minimaler Schnitte einer Familie von Graphen untersucht. Da sichergestellt werden soll, dass die Graphen mehrere nicht-triviale minimale Schnitte enthalten, wurden Familien, die auf Kakteen basieren, verwendet. Getestet wurden also Familien der Typen ZK, UK und GK jeweils mit unterschiedlichen λ. Dabei wurde im Fall des Algorithmus zur optimalen Lösung jeweils das Ergebnis der Heuristik als Startwert verwendet und die dazu benötigte Zeit dem optimalen Lösungsalgorithmus zugeschlagen.

Bei Typ-ZK-Familien zeigt sich, dass die Berechnung verhältnismäßig schnell läuft (siehe Abbildung 5.3a). Der Grund hierfür ist, dass es sich um eine zeitabhängige Familie handelt, bei der von Graph zu Graph immer weniger minimale Schnitte vorhanden sind, das heißt, dass $\prod_i \mathcal{C}(G_i)$ relativ klein ist. Der zusätzliche Zeitbedarf, um das Problem optimal lösen zu können, ist gering. Die beiden Berührungspunkte bei $\lambda = 10$ und $\lambda = 50$ ergeben sich daraus, dass alle betreffenden Testfamilien jeweils einen Schnitt gemeinsam haben, das Ähnlichkeitsmaß also den optimalen Wert 0 liefert und der Algorithmus zur optimalen Berechnung mit Startwert 0 nicht gestartet wird.

Typ-GK-Familien benötigen deutlich mehr Zeit (siehe Abbildung 5.3b). Doch auch auch hier liegt die Laufzeit für die optimale Lösung in der gleichen Größenordnung, wie



(a) Anzahl der Kaktus-Aktualisierungen in der Zeit einer Kaktusberechnung in Graphen vom Typ ZK, $n = 1000$.



(b) Anzahl der Kaktus-Aktualisierungen in der Zeit einer Kaktusberechnung in Graphen vom Typ ZK.

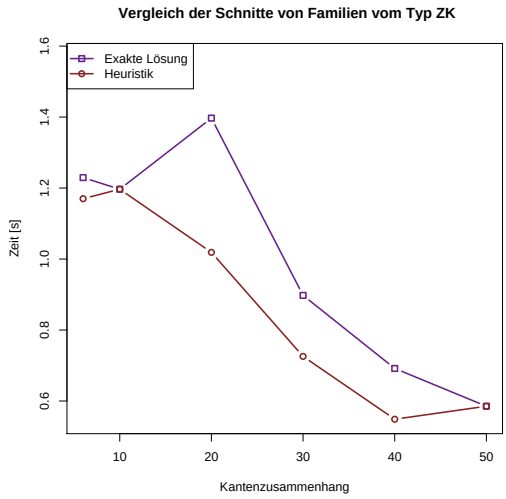
Abbildung 5.2: Effizienz der Kaktus-Aktualisierungen.

die der Heuristik. Der erhöhte Zeitbedarf ist zum einen auf die größere Zahl minimaler Schnitte und zum anderen auf die geringere Ähnlichkeit zwischen den Graphen zurückzuführen.

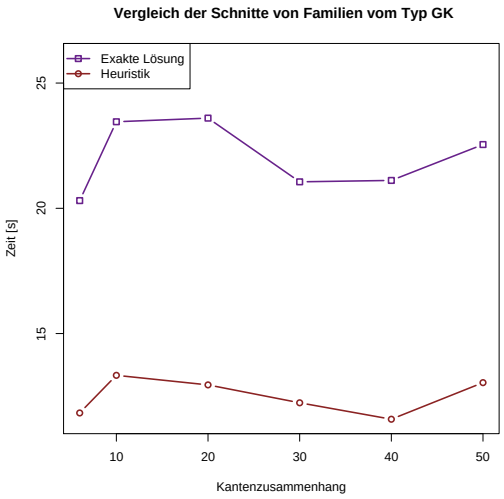
Bei Typ-UK-Familien ergibt sich dagegen ein völlig anderes Bild (siehe Abbildung 5.3c). Hier benötigt die Berechnung der optimalen Lösung mehrere Stunden, während die Heuristik nach wenigen Sekunden fertig ist.

Alle Testgraphen haben gemeinsam, dass bereits die Heuristik stets die optimale Lösung gefunden hat. Gerade bei Graphfamilien mit geringerer Ähnlichkeit, wie den Typ-GK-Familien, ist die schnelle Heuristik eine gute Alternative zur optimalen Berechnung.

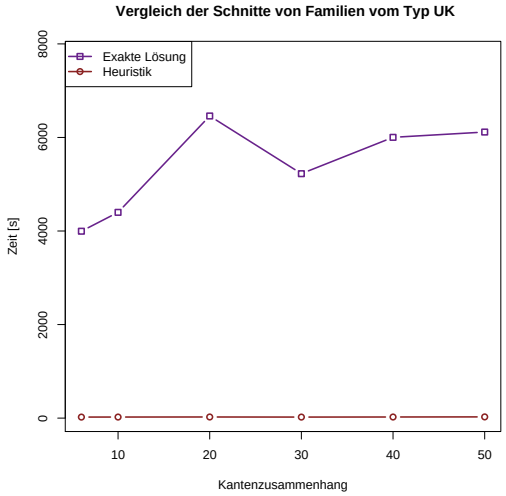
In Abbildung 5.4 kann man ein Beispiel einer Typ-ZK-Familie sehen, in denen die Graphen Distanz 1 haben. Rechts neben den Graphen ist die Kaktusrepräsentation ihrer minimalen Schnitte zu sehen; die ähnlichsten minimalen Schnitte sind eingefärbt. Aus den Kaktusrepräsentationen ergibt sich, dass es keinen Schnitt gibt, der in allen Graphen minimal ist. Es gibt allerdings einen minimalen Schnitt $S = \{5, 12, 13, 14\}$, der nur in G_5 nicht enthalten ist; dort wurde der Schnitt $T = \{13\}$ gewählt. Statt der Schnitt-Kombination 4 mal S und 1 mal T, wäre auch die Kombination 2 mal S und 3 mal T (in G_1, G_4 und G_5) möglich gewesen, ohne dass sich die Anzahl verschiedener Schnitte erhöht hätte. Hier zeigt sich, dass die Bewertungfunktion eher eine einzelne Abweichung zulässt, als mehrere, wenn auch gleiche Abweichungen. Gut zu sehen ist in der Abbildung auch, wie empfindlich minimale Schnitte gegenüber Veränderungen im Graph sind.



(a) Typ ZK

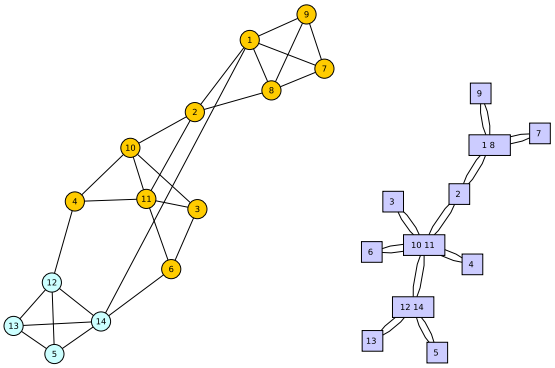


(b) Typ GK



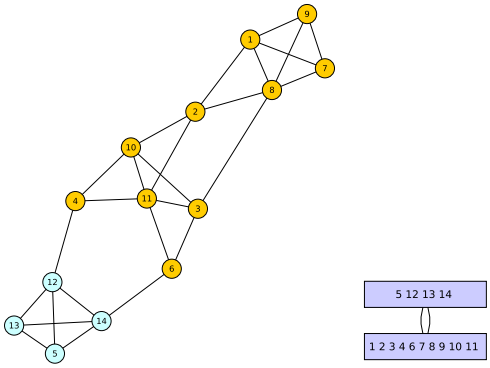
(c) Typ UK

Abbildung 5.3: Laufzeit für den Vergleich minimaler Schnitte für unterschiedliche Graphtypen.

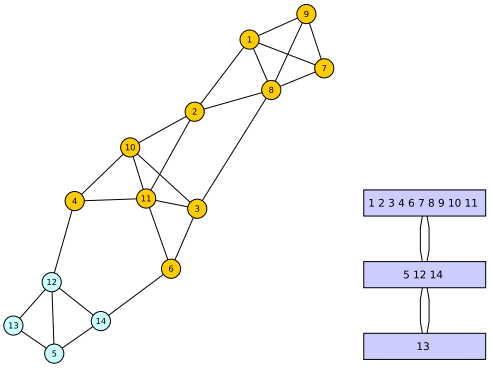


(a) $G_1 = (V, E_1)$ und $\mathcal{R}_1$

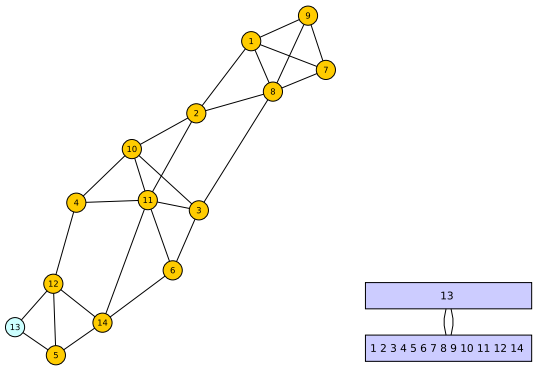
(b) $G_2 = (V, E_1 \setminus \{\{1, 14\}\})$ und $\mathcal{R}_2$



(c) $G_3 = (V, E_2 \cup \{\{3, 8\}\})$ und $\mathcal{R}_3$



(d) $G_4 = (V, E_3 \setminus \{\{13, 14\}\})$ und $\mathcal{R}_4$



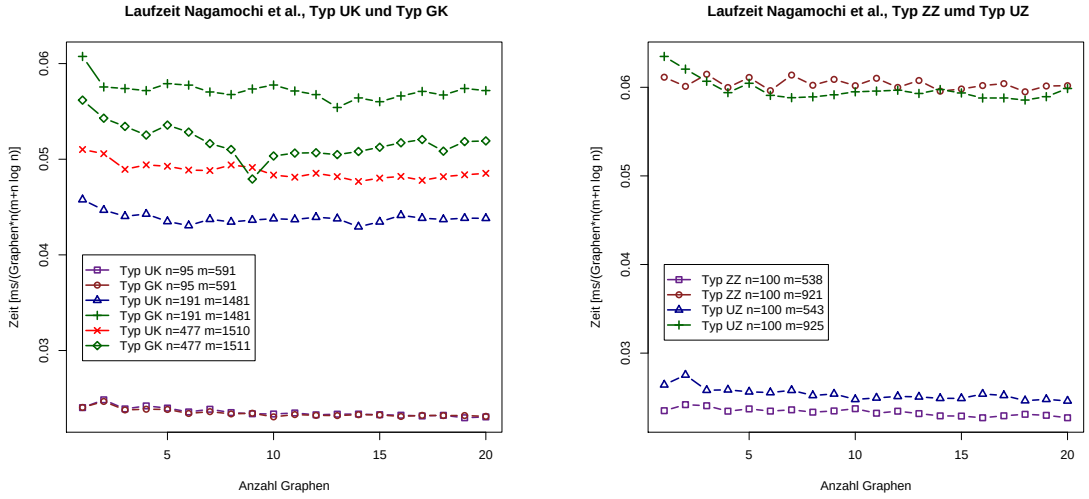
(e) $G_5 = (V, E_4 \cup \{\{14, 11\}\})$ und $\mathcal{R}_5$

Abbildung 5.4: Beispielfamilie (G_i) mit den Kaktusrepräsentationen $(\mathcal{R}_i, \varphi_i)$ ihrer minimaler Schnitte; zwischen zwei Graphen G_i und G_{i+1} wurde jeweils eine Kante eingefügt oder entfernt. Eingefärbt sind in den Graphen die minimalen Schnitte, die bezüglich der Bewertungsfunktion die größte Ähnlichkeit haben.

5.4 Kleine Schnitte

Zunächst ist soll die Laufzeit der beiden Algorithmen einzeln untersucht werden, bevor beide miteinander verglichen werden.

Algorithmus auf Basis von Nagamochi et al. Der Algorithmus $\mathcal{A}_{\text{NNI}}$ zur Berechnung kleiner Schnitte auf Basis des Algorithmus von Nagamochi, Nishimura und Ibaraki wurde mit Familien der Typen UK, GK, ZZ und UZ getestet. Familien vom Typ ZK sind hier ungeeignet, da diese häufig gleiche minimale Schnitte enthalten. Die Ergebnisse sind in Abbildung 5.5 zu sehen. Angegeben ist jeweils die durchschnittliche Knoten- und Kantenzahl der getesteten Graphen.



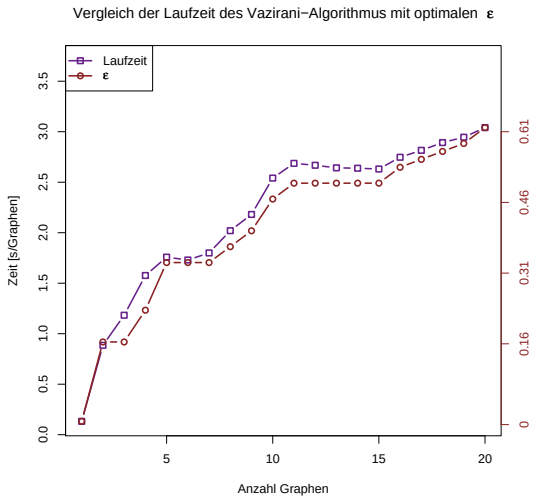
(a) Vergleich verschiedener Familien, die auf Kakteen basieren.

(b) Vergleich verschiedener zufälliger Familien.

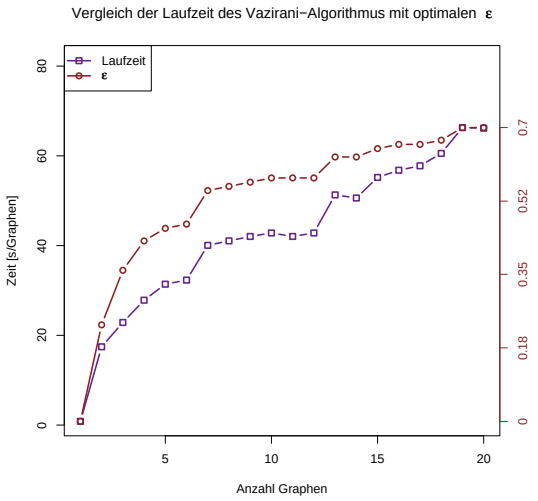
Abbildung 5.5: Laufzeit des zur Berechnung kleiner Schnitte von Algorithmus $\mathcal{A}_{\text{NNI}}$.

Die Anzahl der Schnittkandidaten während der Berechnung von Phase 2 ist entscheidend für die Laufzeit. In allen getesteten Graphfamilien der Typen ZK, UK, GK, ZZ und UZ gab gab es im Durchschnitt maximal 8,4 Schnittkandidaten während der Berechnung. Das Maximum waren 101 Schnittkandidaten; damit dominierte in allen Testfällen die Laufzeit der ersten Phase des Algorithmus deutlich die Gesamtlaufzeit. Der Algorithmus $\mathcal{A}_{\text{NNI}}$ verhält sich sehr robust. Seine Laufzeit hängt nur unwesentlich davon ab, wie ähnlich sich die Schnitte sind und wie die Struktur der Graphen aussieht. Die entscheidenden Faktoren für die Laufzeit sind die Knoten- und Kantenzahl; die Vorhersagbarkeit des Zeitbedarf ist also ein Vorteil dieses Algorithmus.

Algorithmus auf Basis von Vazirani und Yannakakis. Die Laufzeit des Algorithmus $\mathcal{A}_{\text{VY}}$ auf Basis des Algorithmus von Vazirani und Yannakakis ist bestimmt durch die Anzahl der Schnitte, die berechnet werden müssen, bis ein gleicher Schnitt in allen



(a) Familie vom Typ UK, $n = 100$, $\lambda = 6$, $m \approx 400$



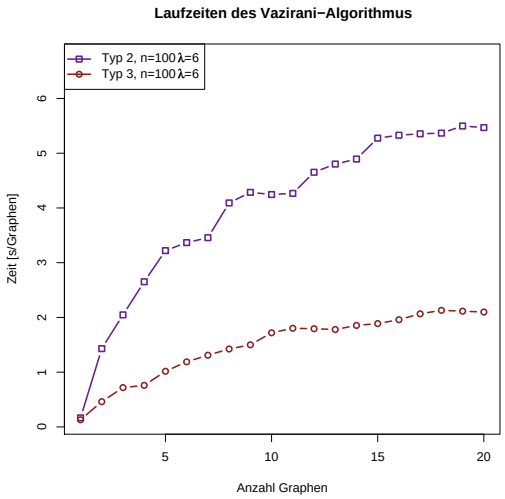
(b) Familie vom Typ UK, $n = 200$, $\lambda = 10$, $m \approx 1480$

Abbildung 5.6: Vergleich der Laufzeit des Algorithmus $\mathcal{A}_{\mathrm{VY}}$ mit dem optimalen ε.

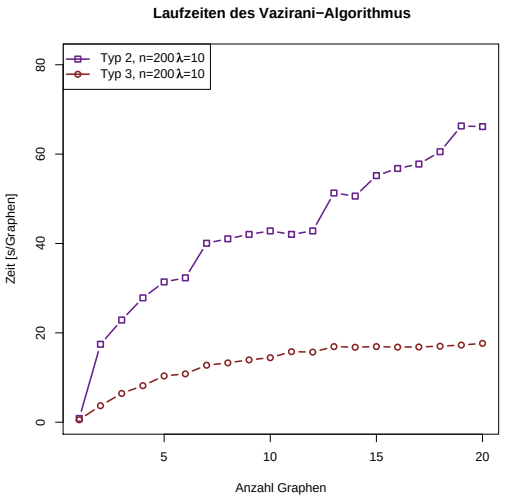
Graphen gefunden wurde. Die Anzahl der Schnitt ist durch das ε (siehe Definition in Abschnitt 1.2) bestimmt. Das zeigt sich auch in den Ergebnissen. Das gefundene ε ist der Laufzeitkurve sehr ähnlich, in vielen Fällen sogar nahezu identisch (siehe Abbildung 5.6).

Wie in Abbildung 5.7 zu sehen ist hängt die Laufzeit stark von der Ähnlichkeit der Graphen einer Familie ab. Die Ausführungszeit bei Typ-GK-Familien, in denen die Knoten mit großer Wahrscheinlichkeit den gleichen Kaktusknoten zugeordnet werden, liegt bei etwa einem Drittel der Zeit, die Typ-UK-Familien benötigen, in denen die Knoten zufällig auf die Kaktusknoten verteilt werden. Das zeigt sich im Vergleich von Typ-ZZ und Typ-UZ-Familien noch viel deutlicher: Während die kleinen Schnitte bei zeitabhängigen Graphen vom Typ ZZ in Zeiten von deutlich unter einer Sekunde gefunden werden, benötigt die Berechnung bei Typ-UZ-Graphen, die zeitunabhängig sind bis zu 300 Sekunden pro Graph. Auffällig ist, dass die Kurve bei den zufälligen Graphen sehr flach beginnt und die Steigung am Ende stark zunimmt, während es sich bei den Kurven der Kaktus-Typen andersherum verhält. Grund für den unterschiedlichen Verlauf ist, dass bei zufällig erzeugten Graphen viele kleine triviale Schnitte, also Schnitte, die aus einem Knoten bestehen, existieren; diese werden schnell gefunden. Ab einer bestimmten Anzahl Graphen gibt es dann keine übereinstimmenden trivialen Schnitte mehr, was zu einer erheblichen Laufzeitsteigerung führt. Die Kaktusfamilien werden so erzeugt, dass es nur sehr wenige triviale kleine Schnitte gibt.

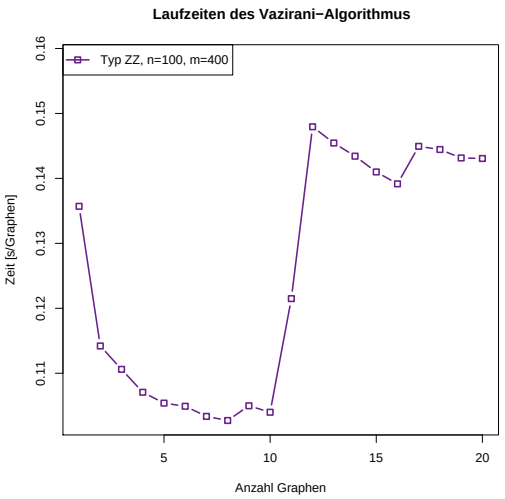
Vergleich beider Algorithmen. Im Vergleich zeigt sich, dass es sowohl Graphfamilien gibt, in denen der Algorithmus $\mathcal{A}_{\mathrm{NNI}}$ besser ist, als auch Graphfamilien, in denen der Algorithmus $\mathcal{A}_{\mathrm{VY}}$ schneller läuft. Algorithmus $\mathcal{A}_{\mathrm{VY}}$ profitiert sehr stark davon, wenn



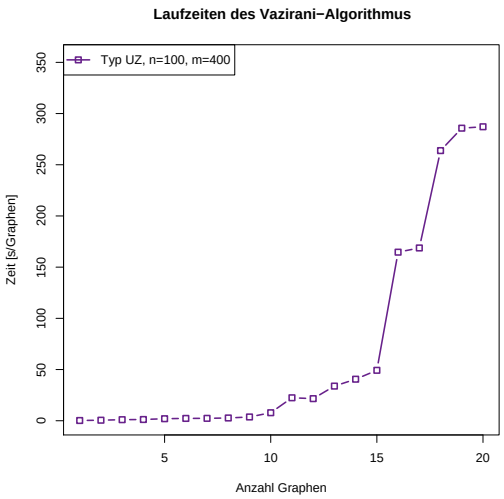
(a) Familien vom Typ UK und GK, $n = 100$, $\lambda = 6$, $m \approx 400$



(b) Familien vom Typ UK und GK, $n = 200$, $\lambda = 10$, $m \approx 1480$



(c) Familie vom Typ ZZ, $n = 200$, $m = 400$



(d) Familie vom Typ UZ, $n = 200$, $m = 400$[a]

[a] Jeder Datenpunkt ist hier das Ergebnis von nur drei Iterationen.

Abbildung 5.7: Vergleich der Laufzeiten von Algorithmus $\mathcal{A}_{\mathrm{VY}}$ mit verschiedenen Graphfamilien.

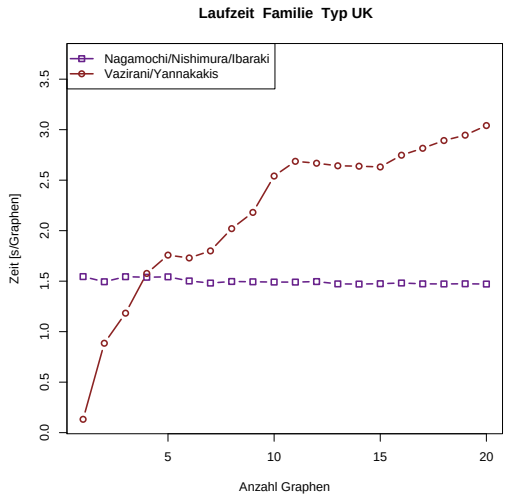
sich die Graphen einer Familie ähnlich sind, während die Laufzeit von Algorithmus $\mathcal{A}_{\text{NNI}}$ im Wesentlichen durch Knoten- und Kantenzahl bestimmt ist. Die Ausführungszeit des letztgenannten Algorithmus ist linear in der Anzahl der Graphen; er eignet sich daher auch besonders für große Graphfamilien. Ein typisches Bild im Vergleich zeigt Abbildung 5.8; die ähnlicheren Typ-GK-Familien liegen dem Algorithmus $\mathcal{A}_{\text{VY}}$, während er sich bei Typ-UK-Familien schwächer zeigt.

Bei den Testläufen zeigte das Ergebnis der Heuristik eine gute Qualität; von 3600 Tests hat die Heuritik nur in 17 Fällen (0,5 %) keinen kleinsten Schnitt gefunden. Getestet wurden Familien der Typen UK, GK, ZZ und UZ. Dabei traten alle 17 Fälle, in denen die Heuristik kein optimales Ergebnis geliefert hat, bei Familien vom Typ GK auf.

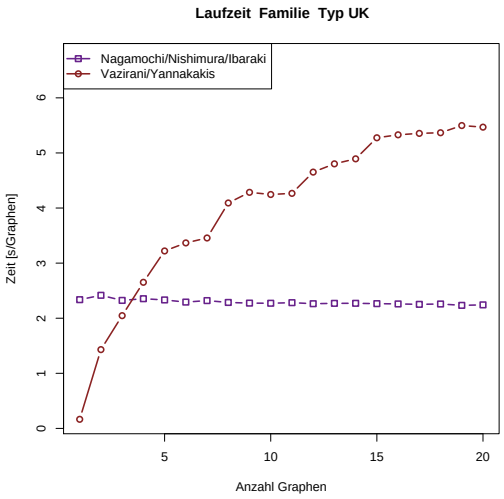
Clusterung. Die in Abschnitt 3.4 vorgestellt Clusterung wurde auf dem E-Mail-Graphen getestet. Da nicht alle Mitarbeiter in jedem Zeitfenster an der E-Mail-Kommunikation beteiligt waren, bilden diese in den anderen Graphen jeweils eine eigene Zusammenhangskomponente. Damit erhöht sich die Anzahl kleiner Schnitte drastisch, da schon die Zahl minimaler Schnitte exponentiell in der Anzahl der Zusammenhangskomponenten ist. In diesem Fall dominierte Phase 2 des Algorithmus $\mathcal{A}_{\text{NNI}}$ die Laufzeit deutlich; aufgrund der großen Anzahl an Schnittkandidaten war die Berechnung nur mit der modifizierten Variante mit Speicherkomplexität $O(km)$ möglich.

Die neun E-Mail-Graphen der Familie und das Ergebnis der Clusterung sind in Abbildung 5.9 zu sehen. Es sind mehrere größere Cluster in den verschiedenen Graphen zu sehen deren Knoten untereinander relativ stark zusammenhängen. Einige Cluster werden aber auch aus nicht zusammenhängenden Knoten gebildet; das liegt daran, dass vor allem zu Beginn des Verfahrens kleine Schnitte häufig zwei nicht zusammenhängende Knoten abtrennen. Insgesamt ergibt sich aus dem Verfahren eine Clusterung, die über die Zeit stabil ist.

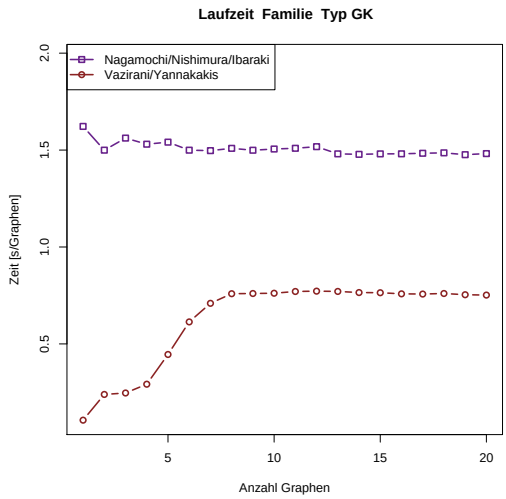
In Abbildung 5.10a sieht man die Clusterung dargestellt im Vereinigungsgraphen; in Abbildung 5.10b ist zusätzlich durch Einfärben der Knoten eine Clusterung der Vereinigungsgraphen mit dem Newman-Verfahren [New04] dargestellt. Hier zeigt sich in einigen Komponenten eine große Übereinstimmung, auch wenn manche Knoten die nach einem Verfahren einem Cluster zugeordnet werden im anderen Verfahren auf zwei Cluster verteilt werden.



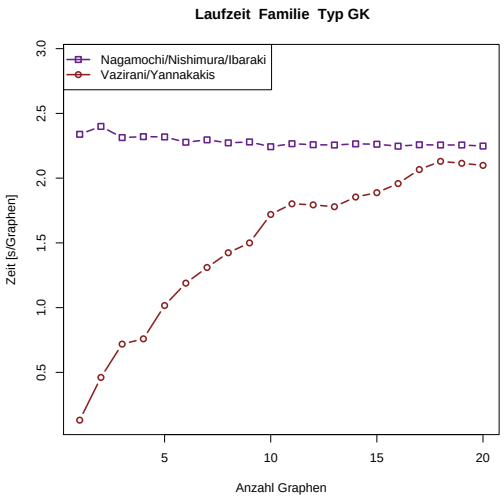
(a) Typ UK, $n = 100$, $m \approx 425$



(b) Typ UK, $n = 100$, $m \approx 619$

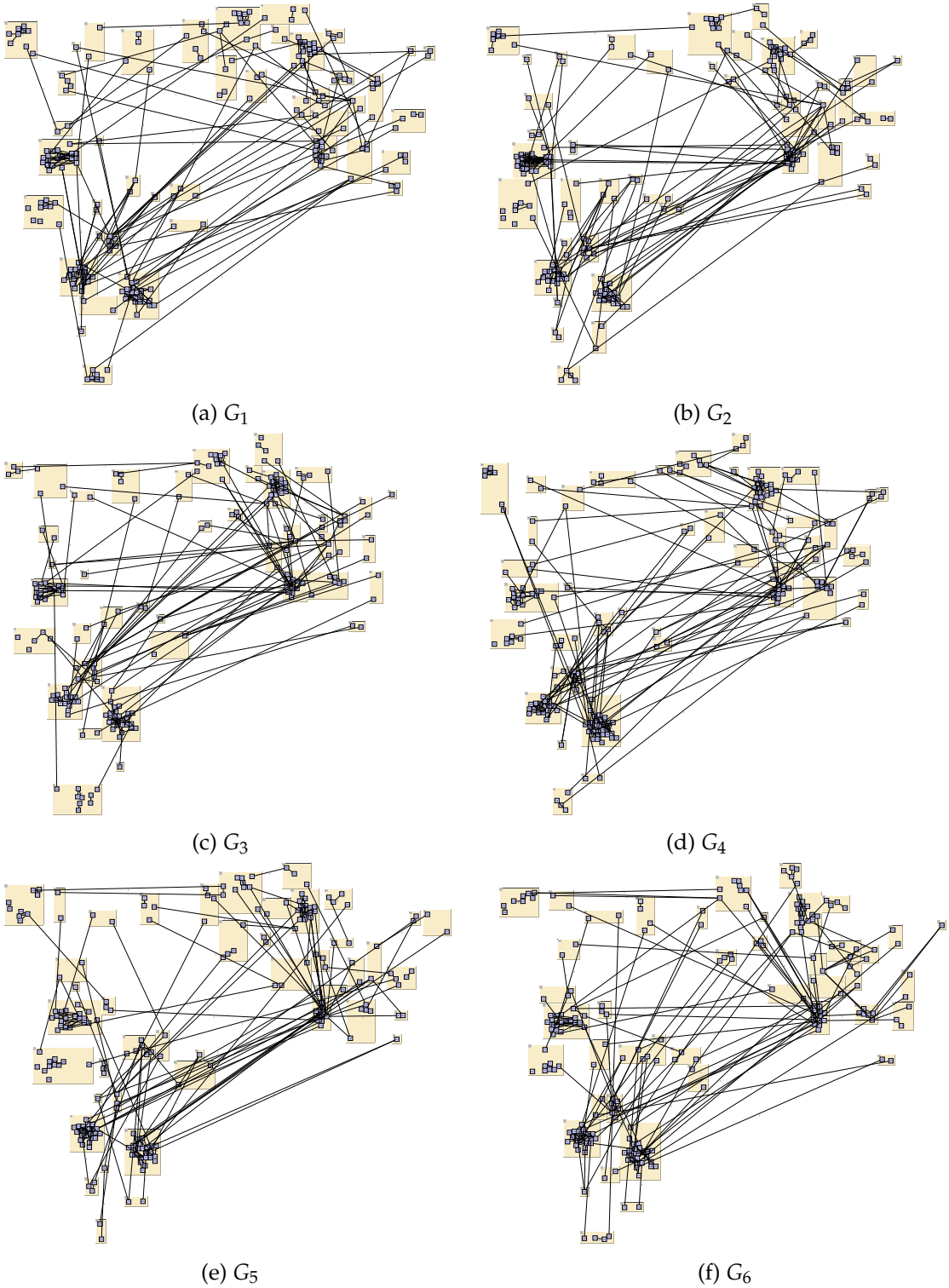


(c) Typ GK, $n = 100$, $m \approx 425$



(d) Typ GK, $n = 100$, $m \approx 619$

Abbildung 5.8: Laufzeit der Algorithmen für Familien mit 100 Knoten und $\lambda = 6$.

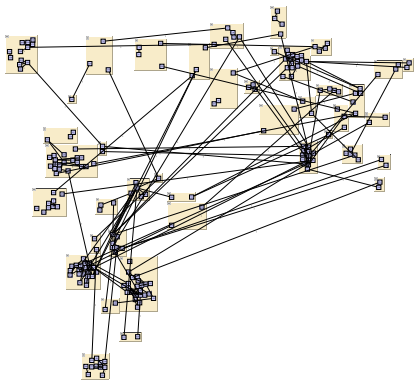


(a) G_1 (b) G_2

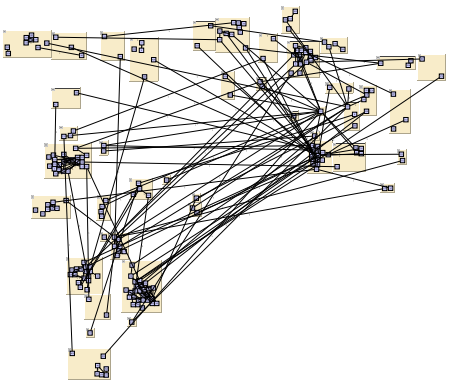
(c) G_3 (d) G_4

(e) G_5 (f) G_6

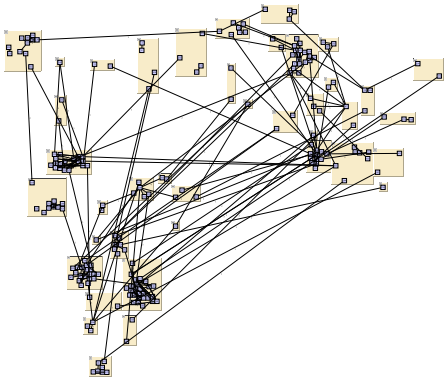
Abbildung 5.9: E-Mail-Graphen $G_1, \ldots, G_6$



(g) G_7

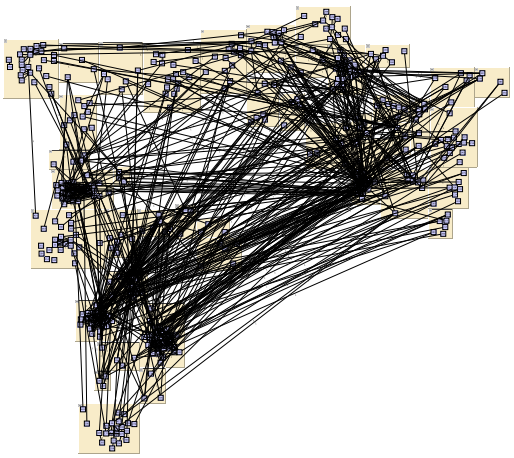


(h) G_8

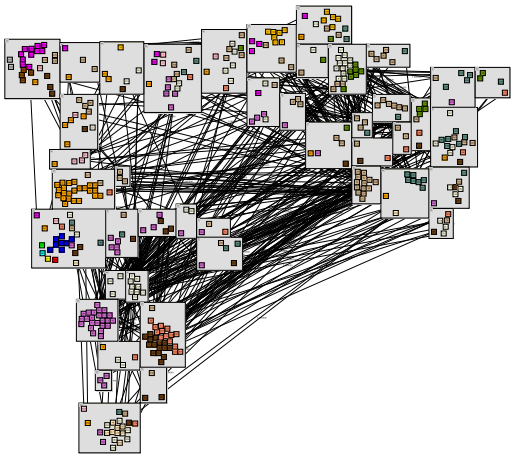


(i) G_9

Abbildung 5.9: E-Mail-Graphen $G_7, \ldots, G_9$



(a) Clusterung im Vereinigungsgraphen.



(b) Die Clusterung durch simultane kleine Schnitte (Gruppierung) im Vergleich zur Newman-Clusterung (Farben).

Abbildung 5.10: Clusterung der E-Mail-Graphfamilie.

6 Zusammenfassung und Ausblick

Im Folgenden werden die wichtigsten Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick über weiterführende Problemstellungen gegeben.

6.1 Zusammenfassung

In dieser Arbeit wurde das Problem behandelt simultane minimale Schnitte in Familien von Graphen zu berechnen. Dabei wurden zwei Problemvarianten identifiziert; die erste Variante fragt nach minimalen Schnitten, die sich in allen Graphen der Familie ähnlich sind und die zweite Variante sucht einen gemeinsamen Schnitt, der in allen Graphen gleich ist. Außerdem wurde ein Graphgenerator zur Erzeugung von Graphen mit vorgegebener minimaler Schnittstruktur vorgestellt.

Minimale Schnitte. Die Kaktus-Repräsentation wurde als geeignete Darstellung für minimale Schnitte verwendet. Auf ihrer Grundlage wurde eine dynamische Kaktus-Repräsentation entwickelt, das heißt Algorithmen, die die Kaktus-Repräsentation nach dem Einfügen oder Entfernen von Graphkanten aktualisieren. Es wurde gezeigt, dass diese Aktualisierung in vielen Fällen asymptotisch schneller ist, als eine vollständige Neuberechnung. Die experimentelle Untersuchung hat diese Ergebnisse bestätigt. Die dynamische Kaktus-Repräsentation konnte verwendet werden, um effizient die minimalen Schnitte von Graphfamilien zu berechnen. Die Experimente haben gezeigt, dass der Vorteil bei ähnlichen bzw. zeitabhängigen Graphen sehr groß ist. Es wurden Ähnlichkeitsmaße für Schnitte vorgestellt und Algorithmen präsentiert, die mit Hilfe der Kaktus-Repräsentation ähnliche minimale Schnitte in Graphfamilien finden. Die vorgestellte schnelle Heuristik konnte das Problem in allen Testfällen optimal lösen.

Kleine Schnitte. Zwei bekannte Algorithmen zur Berechnung kleiner Schnitte wurden für den Fall von Graphfamilien adaptiert, so dass sie das simultane Schnitt-Problem für kleine Schnitte lösen können. Dabei konnte der ursprünglich exponentielle Speicherbedarf auf linearen Platzverbrauch reduziert werden, womit die Voraussetzung geschaffen wurde, auch größere Probleminstanzen lösen zu können. Die experimentelle Auswertung hat ergeben, dass es für beide Algorithmen Graphklassen gibt, in denen sie einen deutlichen Laufzeitvorteil gegenüber dem anderen Algorithmus haben; das Verfahren auf Basis von Nagamochi et al. eignet sich eher für Graphklassen mit geringerer Ähnlichkeit, während das Verfahren auf Basis des Algorithmus von Vazirani und Yannakakis für ähnliche Graphen schneller ist. Als Anwendungsfall wurde ein Top-Down-Clusterungsverfahren für Graphfamilien vorgestellt, das eine Clusterung findet, die über die Zeit stabil ist.

Graphgenerator. Es wurde ein vollständiger Graphgenerator vorgestellt, der beliebige Kakteen erzeugen kann und daraus mit Hilfe der dynamischen Kaktus-Repräsentation einen Graphen konstruiert, dessen minimale Schnitte durch diesen Kaktus repräsentiert werden. Somit ist es möglich Graphen mit beliebiger minimaler Schnittstruktur zu erzeugen, um Testfälle für Schnittprobleme zu erhalten.

6.2 Ausblick

Der Algorithmus zur Berechnung kleiner Schnitte auf Basis von Vazirani und Yannakakis kann vermutlich durch die Weiterentwicklung von Yeh und Wang [YW08] noch beschleunigt werden. Das Clusterungsverfahren auf Basis simultaner kleiner Schnitte wirft noch das Problem auf, dass zwei unzusammenhängende Knoten (oder Komponenten) abgeschnitten und damit einem Cluster zugeordnet werden. Eine Möglichkeit das Problem zu lösen wäre bevorzugt Schnitte zu wählen, die in allen Graphen der Familie in beiden Teilen zusammenhängend sind.

Die Intrakanten der durch den Graphgenerator erzeugten Graphen werden noch völlig zufällig gewählt. In einem Graphen mit einer komplexen minimalen Schnittstruktur würde man aber auch innerhalb einzelner Komponenten eine Struktur erwarten. Hier wäre also eine sinnvolle Erweiterung die Bestimmung zusätzliche Parameter, die die Graphstruktur innerhalb der einzelnen Zusammenhangskomponenten vorgeben.

Literaturverzeichnis

[AM99] Srinivasa R. Arikati and Kurt Mehlhorn. A correctness certificate for the Stoer-Wagner min-cut algorithm. *Information Processing Letters*, 70(5):251–254, 1999.

[Bix75] Robert E. Bixby. The minimum number of edges and vertices in a graph with edge connectivity n and m n-bonds. *Networks*, 5(3):253–298, 1975.

[CGK+97] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 324–333, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[Che79] Boris V. Cherkassky. A fast algorithm for computing maximum flow in a network. In A. V. Karzanov, editor, *Combinatorial Methods for Flow Problems*, volume 3 of *Collected Papers*, pages 90–96. The Institute for Systems Studies, Moscow, 1979.

[Din70] Efim A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics-Doklady*, 11(5):1277–1280, September 1970.

[DKL76] Efim A. Dinic, Alexander V. Karzanov, and Michael V. Lomonosov. A structure of the system of all minimum cuts of a graph. In A.A. Fridman, editor, *Studies in Discrete Optimization*, pages 290–306. Nauka, Moscow, 1976.

[DM89] Ulrich Derigs and Wolfgang Meier. Implementing Goldberg's max-flow-algorithm - a computational investigation. *Mathematical Methods of Operations Research*, 33(6):383–403, November 1989.

[EFS56] Peter Elias, Amiel Feinstein, and Claude E. Shannon. A note on the maximum flow through a network. *IRE Transactions on Information Theory*, 2(4):117–119, December 1956.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[FF56] Lester R. Ford, Jr. and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FF09] Tamás Fleiner and András Frank. A quick proof for the cactus representation of mincuts. Technical Report QP-2009-03, Egerváry Research Group, Budapest, 2009. `www.cs.elte.hu/egres`.

[Fle99] Lisa Fleischer. Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same asymptotic run time. *Journal of Algorithms*, 33(1):51–72, 1999.

[Fra94] András Frank. On the edge-connectivity algorithm of Nagamochi and Ibaraki. Laboratoire Artemis, IMAG, Université Joseph Fourier, Grenoble, 1994.

[GH61] Ralph E. Gomory and Te Chiang Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[Gol85] Andrew V. Goldberg. A new max-flow algorithm. Technical Memo MIT/LCS/TM, MIT Laboratory for Computer Science, November 1985.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[Ham82] Horst W. Hamacher. An $O(Kn^4)$ algorithm for finding the K best cuts in a network. *Operations Research Letters*, 1(5):186–189, 1982.

[HO92] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 165–174, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[HPQ84] Horst W. Hamacher, Jean-Claude Picard, and Maurice Queyranne. On finding the K best cuts in a network. *Operations Research Letters*, 2(6):303–305, 1984.

[HW96] Monika Henzinger and David P. Williamson. On the number of small cuts in a graph. *Information Processing Letters*, 59(1):41–44, 1996.

[Kar74] Alexander V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics-Doklady*, 15(2):434–437, 1974.

[Kar93] David R. Karger. Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-out algorithm. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[Kar00] David R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000.

[KRT94] Valerie King, Satish Rao, and Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[KS96] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

[KT86] Alexander V. Karzanov and Eugeniy A. Timofeev. Efficient algorithm for finding all minimal edge cuts of a nonoriented graph. *Cybernetics and Systems Analysis*, 22:156–162, 1986.

[Men27] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[New04] Mark E. J. Newman. Fast algorithm for detecting community structure in networks. *Physical Review E*, 69(6):066133, Jun 2004.

[NI92a] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[NI92b] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5–6):583–596, 1992.

[NI08] Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Number 123 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, New York, NY, USA, 2008.

[NK94] Hiroshi Nagamochi and Tiko Kameda. Canonical cactus representation for all minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 11:343–361, 1994.

[NNI97] Hiroshi Nagamochi, Kazuhiro Nishimura, and Toshihide Ibaraki. Computing all small cuts in an undirected network. *SIAM Journal on Discrete Mathematics*, 10(3):469–481, 1997.

[NNI03] Hiroshi Nagamochi, Shuji Nakamura, and Toshimasa Ishi. Constructing a cactus for minimum cuts of a graph in $O(mn + n^2 \log n)$ time and $O(m)$ space. *IEICE Transactions on Information and Systems*, 86(2):179–185, 2003.

[NOI94] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67(3):325–341, 1994.

[PQ80] Jean-Claude Picard and Maurice Queyranne. On the structure of all minimum cuts in a network and applications. In T. Brian Boffey, editor, *Mathematical Programming Study*, volume 13, pages 8–16. North-Holland, 1980.

[PQ82] Jean-Claude Picard and Maurice Queyranne. Selected applications of minimum cuts in networks. *Information Systems and Operational Research*, 20(4):394–422, 1982.

[PR90] Manfred Padberg and Giovanni Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47(1):19–36, 1990.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[SW97] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[VY92] Vijay V. Vazirani and Mihalis Yannakakis. Suboptimal cuts: Their enumeration, weight and number (extended abstract). In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 366–377, London, UK, 1992. Springer-Verlag.

[YW08] Li-Pu Yeh and Biing-Feng Wang. Efficient algorithms for the k smallest cuts enumeration. In *COCOON '08: Proceedings of the 14th annual international conference on Computing and Combinatorics*, pages 434–443, Berlin, Heidelberg, 2008. Springer-Verlag.

[yWo] yWorks. yfiles. http://www.yworks.com.