Datenstrukturen und Algorithmen

Theorieseminar des Fachbereichs Informatik

Johann Wolfgang Goethe-Universität Frankfurt am Main, Sommersemester 2000

Professor Hagerup, Professur für Komplexität und Algorithmen

Minimale Schnitte in Graphen

Autoren:

Ernst-Joachim Preussler

Email: erps@cs.uni-frankfurt.de

Patrick Klein

Email: patrick@cs.uni-frankfurt.de

Inhaltsübersicht

1 EINLEITUNG ... 2

1.1 PROBLEM ... 2

1.2 BISHERIGE VERFAHREN ... 3

1.3 ANWENDUNGSGEBIETE ... 4

2 DER ALGORITHMUS MINIMUMCUT ... 7

2.1 BEISPIEL ... 7

2.2 DER ALGORITHMUS ... 8

2.3 KORREKTHEITSBEWEIS ... 10

2.4 LAUFZEITANALYSE ... 13

2.5 ABSCHLIEßENDE BEMERKUNGEN ... 13

2.6 AUSFÜHRLICHES BEISPIEL ... 14

3 EIN STOCHASTISCHER ANSATZ ZUR MIN-CUT-BERECHNUNG ... 15

3.1 DER CONTRACT-ALGORITHMUS ... 15

3.2 IMPLEMENTIERUNG DES CONTRACT-ALGORITHMUS ... 19

3.3 DER REKURSIVE CONTRACT‘-ALGORITHMUS ... 24

3.4 STRENG POLYNOMIELLE ZUFÄLLIGE AUSWAHL ... 30

3.5 AUSBLICK ... 32

3.6 AUSFÜHRLICHES BEISPIEL VON RECURSIV-CONTRACT ... 34

4 LITERATURVERWEISE ... 35

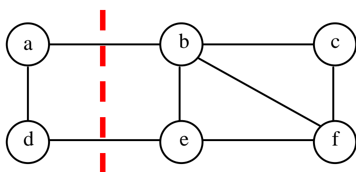
1 Einleitung

1.1 Problem

Die dem Seminarvortrag zugrundeliegenden Papers beschreiben zwei neue Algorithmen zum Berechnen minimaler Schnitte in ungerichteten kantengewichteten Graphen. Im Gegensatz zu vielen klassischen Verfahren beruhen diese Algorithmen nicht mehr auf Flußalgorithmen (Bestimmung des maximalen Flusses in *s*-*t*-Netzwerken).

Was ist ein minimaler Schnitt in einem ungerichteten kantengewichteten Graphen? Zuerst soll geklärt werden, was ein Schnitt ist. Ein **Schnitt** ist eine Aufteilung der Knotenmenge des Graphen in zwei nicht leere Partitionen.

In der Literatur findet man zwei verschiedene Definitionen für den Schnitt: Einerseits wird der Schnitt durch die beiden entstehenden Knotenmengen definiert: Der Schnitt eines Graphen ist dann die Menge der Knoten einer Partition. Andererseits kann der Schnitt auch die durchtrennten Kanten bezeichnen: Dies sind also alle Kanten, bei denen ein Endknoten in der ersten Partition und der andere Endknoten in der zweiten Partition liegt. Beide Beschreibungen sind gleichwertig, weswegen im folgenden je nach Situation beide Konventionen benutzt werden.



Ein zusammenhängender ungerichteter Graph mit den Knoten {a, b, c, d, e, f} und den Kanten (a,b), (a,d), (b,c), (b,e), (b,f), (c,f), (d,e) und (e,f). Die gestrichelte Linie trennt den Graphen in die beiden Partitionen {a, d} und {b, c, e, f}. Der Schnitt kann sowohl mit {a, d} als auch {b, c, e, f} oder mit den Kanten (a,b) und (d,e) bezeichnet werden.

Das Finden eines beliebigen Schnittes ist eine triviale Aufgabe, weswegen das viel interessantere Problem des minimalen Schnittes betrachtet wird. Ein **minimaler Schnitt** kann auf zwei verschiedene Arten definiert werden.

Einerseits ist der minimale Schnitt der Schnitt mit den wenigsten Kanten zwischen den beiden entstehenden Partitionen, d.h. es werden (anschaulich wie im obigen Beispiel) weniger Kanten „durchtrennt“ als bei allen anderen Schnitten des Graphen. Diese Definition gilt für alle ungewichteten Graphen, d.h. Graphen, bei denen die Kanten nicht mit Werten versehen sind.

Andererseits ist der minimale Schnitt der Schnitt mit dem geringsten Kantengewicht zwischen beiden Partitionen (also der Summe der Kantengewichte aller schneidenden Kanten) in einem gewichteten Graphen. Dabei werden nur nichtnegative reelle Zahlen zugelassen,

da das Problem ansonsten *NP*-vollständig ist. Dies kann gezeigt werden, indem das Problem durch eine einfache Transformation auf das Problem des maximalen Schnittes zurückgeführt wird, das *NP*-vollständig ist (Garey und Johnson [4]).

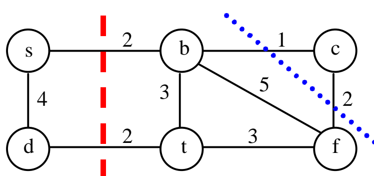
Es ist einfach, die Analogie beider Varianten zu erkennen, indem im ungewichteten Fall das Gewicht jeder Kante auf Eins gesetzt wird. Da es für die Lösung des ungewichteten Falles spezielle, schnelle Algorithmen gibt, wird im folgenden nur der gewichtete Fall angenommen.

Da der minimale Schnitt eines aus mehreren Zusammenhangskomponenten bestehenden Graphen trivialerweise 0 ist, stellt sich das Problem nur für **zusammenhängende Graphen**. Im folgenden wird stets angenommen, daß alle behandelten Graphen zusammenhängend sind, jeder Knoten des Graphen also von jedem anderen Knoten über ungerichtete Kanten erreicht werden kann.

1.2 Bisherige Verfahren

Bisher wurde zur Berechnung eines minimalen Schnittes (Min-Cut) auf die Berechnung eines maximalen Flusses (Maxflow) in einem *s*-*t*-Netzwerk zurückgegriffen. Ein ***s*-*t*-Netzwerk** ist ein kantengewichteter Graph, der über zwei ausgezeichnete Knoten *s* und *t* verfügt, wobei *s* die Quelle und *t* die Senke eines potentiellen Netzwerkflusses markiert. Es wird dabei ausgenutzt, daß der Wert eines maximalen Flusses in einem *s*-*t*-Netzwerk dem Wert eines minimalen Schnittes dieses *s*-*t*-Netzwerkes (minimaler *s*-*t*-Schnitt) entspricht (Max-Flow-Min-Cut-Theorem von Ford und Fulkerson [2]), wobei ein ***s*-*t*-Schnitt** ein Schnitt ist, bei dem sich *s* und *t* zwingend in verschiedenen Partitionen des Graphen befinden müssen. Dazu wird nach dem Berechnen des maximalen Flusses überprüft, welche Knoten des Residualgraphen von *s* aus zu erreichen sind. Diese Knotenmenge ist dann eine Seite des Schnittes.

Das eigentliche Problem des minimalen Schnittes beschränkt sich jedoch nicht auf ein *s*-*t*-Netzwerk, sondern muß den minimalen Schnitt im ganzen Graphen finden. Das folgende Beispiel zeigt einen minimalen *s*-*t*-Schnitt, der aber nicht der minimale Schnitt des kompletten Graphen ist.



Ein kantengewichteter Graph mit den Knoten {s, b, c, d, t, f}. Der Knoten s soll die Quelle und der Knoten t die Senke des Flusses sein, der durch das s-t-Netzwerk fließt. Die gestrichelte Linie kennzeichnet den minimalen s-t-Schnitt, welcher im Wert dem maximalen Fluß (4) entspricht, welcher von s nach t fließen kann. Trotzdem ist der minimale Schnitt des Graphen die Knotenmenge {c}, im Bild durch eine gepunktete Linie abgetrennt (Wert 3).

Eine Lösung könnte darin bestehen, einen Maxflow-Algorithmus auf jede mögliche *s-t*-Kombination anzuwenden, was quadratische Laufzeit benötigt. Es geht aber auch einfacher. Es ist möglich, einen beliebigen Knoten als s auszuzeichnen und dann für jeden anderen Knoten als t einen Maxflow (und damit Min-Cut) zu bestimmen. Damit muß der Maxflow-Algorithmus für n Knoten nur noch n-1 mal durchgeführt werden, das Minimum der gefundenen Werte ermittelt werden und das Problem ist in stark polynomieller Zeit lösbar (da es stark polynomielle Algorithmen für das Maxflow-Problem gibt). Besonders wichtig in diesem Zusammenhang ist die Arbeit von Gomory und Hu [5]. Trotz aller Verbesserungen der Verfahren, um aus maximalen Flüssen minimale Schnitte zu berechnen, benötigt ein Min-Cut-Algorithmus dieser Art $\Omega(mn^2)$ Zeit für m Kanten und n Knoten. Unter Ausnutzung aller Tricks (wie annähernd gleichzeitiger Berechnung aller benötigten Maxflows) können mit Maxflow-Algorithmen minimale Schnitte in $O(mn \log(n^2/m))$ Zeit berechnet werden (Hao und Orlin [7]).
Die Anstrengungen konzentrierten sich also darauf, immer bessere Maxflow-Algorithmen zu finden, um daraus (mit Faktor n) ein gleichartiges Min-Cut-Problem zu machen. In der neueren Literatur und auch in den behandelten Papers werden völlig andere Lösungsversuche unternommen. Dabei wird das Min-Cut-Problem vom Maxflow-Problem gelöst, um effektivere Algorithmen zu finden. Nach neuesten Erkenntnissen zeichnet sich sogar ab, daß das Min-Cut-Problem leichter als das Maxflow-Problem zu behandeln ist, die verwendeten Algorithmen sogar schneller sein können. Leider folgt damit keine schnellere Lösung für das Maxflow-Problem, da es wesentlich schwieriger ist, aus einem minimalen Schnitt einen maximalen Fluß zu berechnen als umgekehrt. Einen guten Überblick über maximale Flüsse und minimale Schnitte gibt das Skript *Flüsse in Netzwerken* von Professor Hagerup [6].
Der erste vorgestellte Algorithmus MINIMUMCUT hat eine Laufzeit von $O(nm + n^2 \log n)$ und findet sicher einen minimalen Schnitt. Der randomisierte zweite Algorithmus von Karger und Stein [B] hat eine Laufzeit von $O(n^2 \log^3 n)$ und findet dabei alle minimalen Schnitte eines Graphen mit hoher Wahrscheinlichkeit.

1.3 Anwendungsgebiete

Das Min-Cut-Problem taucht in zahlreichen Anwendungen auf, von denen hier nur einige kurz aufgeführt werden sollen.

Die Ermittlung minimaler Schnitte in Netzwerken jeglicher Art ist ein wichtiges Gebiet. So ist es für Netzwerkdesign und Netzüberprüfung essentiell, minimale oder annähernd minimale Schnitte zu kennen, um z.B. zufällige Zusammenbrüche im Netzwerk zu erkennen. Bei zufälligen Kantenfehlern mit Wahrscheinlichkeit p pro Kante ist die Wahrscheinlichkeit einer Netzwerktrennung:

$$\sum_k f_k p^k (1-p)^{m-k}$$

Dabei ist m die Kantenanzahl des Netzwerks, und f_k die Anzahl der Kantenmengen der Kardinalität k, welche das Netzwerk trennen. Bei einer geringen Wahrscheinlichkeit p kann dieser Wert approximiert werden, indem nur f_k für kleine k eingerechnet werden. Für diese Anwendung bietet der randomisierte Algorithmus Vorteile, da er alle minimalen oder zumindest annähernd minimalen Schnitte findet.

Eine andere Anwendung besteht darin, Relationen zwischen Gruppen von Hypertextdokumenten zu finden. Dabei können Links zwischen den Texten als Kanten betrachtet werden. Sehr kleine Schnitte zwischen Dokumentgruppen (als Knotenmengen) deuten dann auf geringe Verwandtschaft der Themen hin. Auch hier lohnt es sich auf den randomisierten Algorithmus zurückzugreifen, da das Ergebnis der Berechnung sowieso nur die Wahrscheinlichkeit einer Verwandtschaft minimiert. Selbst bei einem garantierten minimalen Schnitt gibt es keine Gewißheit, daß die beiden Gruppen nicht miteinander verwandt sind.

Ein weiteres Anwendungsgebiet ist die Compilierung paralleler Sprachen, bei der Kanten die Notwendigkeit der Prozessorkommunikation symbolisieren. Es ist also sinnvoll, minimale Schnitte zu finden, um nicht unnötig Ressourcen zu verschwenden. Beide Algorithmen scheinen für diese Aufgabe geeignet zu sein.

Für die exakte Lösung von *Traveling-Salesman*-Problemen wird derzeit auf die Technik der *cutting planes* (Schnittflächen) zurückgegriffen, bei der fortlaufend Teile eines Polytops im mehrdimensionalen Raum abgetrennt werden, sofern sie für die Lösung offensichtlich nicht mehr benötigt werden. Für diese Algorithmen müssen zahlreiche Min-Cut-Probleme gelöst werden (Lawler [8]). Da nach exakten Lösungen gefragt wird, scheint der sichere Algorithmus besser zu sein. Andererseits müssen gleichzeitig mehrere Schnitte gefunden werden, wofür sich der randomisierte Algorithmus anbietet.

Der Algorithmus MINIMUMCUT wurde von Queyranne [11] verallgemeinert und zur Minimierung submodularer Funktionen benutzt. Eine submodulare Funktion f auf einer endlichen Menge V hat die folgende Eigenschaft:

$$f(Y) + f(Z) \geq f(Y \cap Z) + f(Y \cup Z) \quad \text{für alle } Y, Z \subseteq V.$$

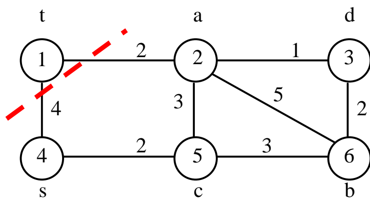
Die Gewichtsfunktion eines Schnittes ist eine submodulare Funktion. Dabei entspricht $f(U)$ ($U \subseteq V$) der gesamten Kapazität aller Kanten, welche U verlassen. Die Kapazität einer Kante ergibt sich durch die Funktion c: $E \rightarrow \square_{+}$. Wird die Funktion f minimiert, entspricht dies letztendlich der Suche nach dem minimalen Schnitt in einem Graphen $G = (V, E)$.

2 Der Algorithmus MINIMUMCUT

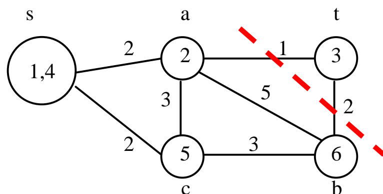
Dieser Teil des Seminarvortrags befaßt sich mit einem neuen Algorithmus zum Berechnen eines minimalen Schnittes in kantengewichteten Graphen, welcher von Mechthild Stoer und Frank Wagner in *Journal of the ACM*, Nr. 4, Juli 1997, Seiten 585-591 [A] vorgestellt wurde. Damals arbeitete Mechthild Stoer am *Televerkets Forskningsinstitutt* in Kjeller, Norwegen. Frank Wagner arbeitete am *Institut für Informatik* der Freien Universität Berlin. Der Algorithmus wurde bereits 1994 entwickelt und unabhängig davon von A. Frank erdacht. Er basiert auf einem 1992 von Nagamochi und Ibaraki [9, 10] vorgestellten Algorithmus, wobei dessen Komplexität deutlich verringert wurde. Dies führt zwar asymptotisch zu keiner besseren Laufzeit, läßt jedoch in der Praxis (durch kleinere Konstanten) eine wesentlich bessere Laufzeit erwarten. Außerdem ist die Korrektheit des vereinfachten Algorithmus leichter zu beweisen und benötigt keine Fallunterscheidung für ungewichtete Graphen, natürliche, rationale und reelle Kantengewichte.

2.1 *Beispiel*

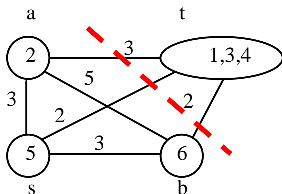
Zuerst wird anhand eines Beispiels der Algorithmus vorgestellt, ohne genauer auf die Ausformulierung des Verfahrens einzugehen.

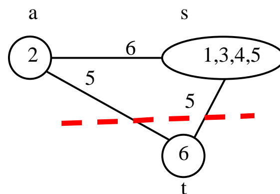


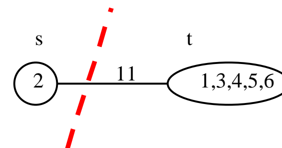
Es wird ein während des ganzen Verfahrens unveränderlicher Startknoten a=2 definiert. Danach werden die Knoten mit b, c, d usw. beschriftet (nach einem noch vorzustellenden Verfahren). Der vorletzte Knoten erhält die Markierung s, der letzte Knoten die Markierung t. Danach wird der Schnitt {t=1} als minimaler s-t-Schnitt dieser Phase (auch *cut_of_the_phase* genannt) ermittelt,. In diesem Fall ist der Wert 6 (markiert durch gestrichelte Linie). Danach werden die Knoten s und t zusammengefaßt und der aktuelle minimale Schnitt zwischengespeichert.



Der neue *cut_of_the_phase* ist {t=3} mit Wert 3 (durch Linie markiert). Wieder werden s und t zusammengefaßt und der jetzt kleinere minimale Schnitt zwischengespeichert.







Die weiteren minimalen s-t-Schnitte sind größer als der bisherige minimale Schnitt. Der Algorithmus wird mit Ausgabe des korrekten Schnittes beendet, wenn nur noch zwei Knoten übrig sind.

2.2 *Der Algorithmus*

Definition: Sei $G = (V, E)$ ein ungerichteter Graph mit der Knotenmenge V und der Kantenmenge E. Jede Kante $e \in E$ hat ein nichtnegatives Gewicht w(e). Sei a ein beliebiger Knoten aus V, der im folgenden als Startknoten bezeichnet wird. Die Menge A ist eine Untermenge von V und zu Beginn des Algorithmus leer.

Algorithmus

```
(* HAUPTPROGRAMM *)
1   PROCEDURE MINIMUMCUT (G, w, a)
2   BEGIN
3   WHILE |V| > 1 DO
4       BEGIN
5       MINIMUMCUTPHASE (G, w, a)
6       IF cut_of_the_phase < current_minimum_cut THEN current_minimum_cut =
        cut_of_the_phase
7       END
8   PRINT current_minimum_cut
9   END
(* ENDE DES HAUPTPROGRAMMS *)

(* UNTERPROGRAMM *)
PROCEDURE MINIMUMCUTPHASE (G, w, a)
A   BEGIN
B   A ← {a}
C   WHILE A ≠ V DO
D       BEGIN
E       ADD TO A THE MOST TIGHTLY CONNECTED VERTEX
F       END
G   STORE THE Cut_of_the_phase
H   SHRINK G BY MERGING THE TWO VERTICES ADDED LAST
I   END
(* ENDE DES UNTERPROGRAMMS *)
```

Anmerkungen zum Hauptprogramm: In **Zeile 1** des Programms werden der Graph (*G*), die Kantengewichtsfunktion (w) und der Startknoten (*a*) durch Variablenübergabe (Programmaufruf) definiert.

In **Zeile 3** beginnt die Schleife des Hauptprogramms, welche abbricht, sobald die Anzahl der Knoten in *V* eins erreicht.

In **Zeile 5** wird das Unterprogramm MINIMUMCUTPHASE mit den aktuellen Parametern aufgerufen.

In **Zeile 6** wird der durch das Unterprogramm ermittelte *cut_of_the_phase* (s. Zeile G) mit dem aktuellen minimalen Schnitt verglichen. Falls der *cut_of_the_phase* kleiner als der bisherige minimale Schnitt ist, wird der *cut_of_the_phase* zum neuen minimalen Schnitt.

In **Zeile 8** wird das Ergebnis ausgegeben.

Anmerkungen zum Unterprogramm: In **Zeile B** wird der Menge *A* der Startknoten *a* zugewiesen. Dies passiert bei jedem Aufruf des Unterprogramms, so daß die Menge *A* jedesmal neu initiiert wird.

In **Zeile C** beginnt eine Schleife, welche abbricht, sobald die Menge *A* der Knotenmenge *V* entspricht.

Zeile E bedarf einer ausführlicheren Erklärung: Es wird ein Knoten *v* aus $V \setminus A$ zu *A* hinzugefügt, der am engsten mit *A* verbunden ist („most tightly connected“). Dies bedeutet, daß die Summe der Gewichte der direkten Kanten von Knoten aus *A* nach *v* maximal sein muß.

Formal: Wähle ein $v \notin A$ so, daß $w(A, v) = \max\{w(A, y) \mid \forall y \notin A\}$

In **Zeile G** wird der *cut_of_the_phase* ermittelt. Dies ist der Schnitt, der aus dem mit *t* markierten, also dem letzten zu *A* hinzugefügten Knoten besteht. Dieser Knoten *t* kann eine Menge von in vorherigen Phasen verschmolzenen Knoten des ursprünglichen Graphen repräsentieren.

In **Zeile H** wird der Graph *G* um einen Knoten verkleinert, indem die beiden letzten zu *A* hinzugefügten Knoten (z.B. *s* und *t*) zusammengefaßt werden („merging“). Eine andere Bezeichnung für diesen Vorgang ist **Kontraktion**. Bei einer Kontraktion wird die Kante (*s*,*t*), falls vorhanden, entfernt und doppelte Kanten (Kanten, die von einem anderen Knoten sowohl zu *s* als auch *t* führen) werden mit der Summe der einzelnen Kantengewichte als neue Kante zusammengefaßt. Beispiel für eine Kontraktion (Knotenzusammenschluß) an der Kante (2,3):

Bemerkungen:

- Das Unterprogramm MINIMUMCUTPHASE enthält einen Algorithmus, welcher in der Literatur unter dem Namen *maximum adjacency (cardinality) search*, also „Maximumsnachbarsuche“, bekannt ist.
- Der Startknoten *a* könnte auch in jeder Phase zufällig gewählt werden, was aber keine Verbesserung des Verfahrens bedeutet, sondern die Implementierung eher aufwendiger machen dürfte.

2.3 Korrektheitsbeweis

Berechnet der vorgestellte Algorithmus tatsächlich einen minimalen Schnitt? Und wenn ja, warum? Um die Korrektheit des Algorithmus zu zeigen, werden zwei Beweise durchgeführt. Das erste zu beweisende Theorem bezieht sich auf das Hauptprogramm:

Theorem 1: Seien s und t zwei Knoten aus G. Sei $G / \{s,t\}$ der Graph, der entsteht, wenn s und t zusammengefaßt werden. Dann ist ein minimaler Schnitt von G das Minimum eines minimalen s-t-Schnittes von G und eines minimalen Schnittes von $G / \{s,t\}$.
Beweis: Wenn es einen minimalen Schnitt in G gibt, der s und t trennt (s und t liegen also in verschiedenen Partitionen), dann ist dies notwendigerweise auch ein minimaler s-t-Schnitt in G. Wenn dies nicht der Fall ist, muß sich der minimale Schnitt in $G / \{s,t\}$ befinden, da s und t in der selben Partition (bezüglich G) liegen und die Kante (s,t), welche als einzige Kante in $G / \{s,t\}$ verschwindet, somit nicht Kante eines minimalen Schnittes sein kann.

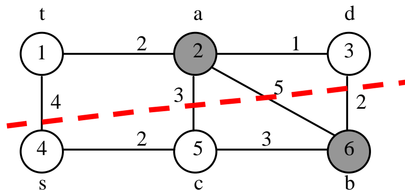
Daraus läßt sich direkt die rekursive Definition des Algorithmus herleiten: In jeder Phase wird ein minimaler s-t-Schnitt in G ermittelt und mit dem bisherigen minimalen Schnitt verglichen. Danach werden s und t zusammengefaßt und die nächste Phase mit $G = G / \{s,t\}$ gestartet. Der Algorithmus terminiert, wenn es nur noch einen Knoten in G gibt. In der letzten Phase (2 Knoten) muß auf jeden Fall der minimale Schnitt ermittelt werden, sofern dies nicht bereits vorher passiert ist.
Was jetzt noch fehlt, ist der Beweis für die Behauptung, daß ein minimaler s-t-Schnitt dem *cut_of_the_phase* entspricht, denn diesen liefert das Unterprogramm in jeder Phase.

Theorem 2: Seien s und t die beiden letzten zu A hinzugefügten Knoten. Dann ist jeder *cut_of_the_phase* ein minimaler s-t-Schnitt.

Beweis: Es genügt zu zeigen, daß jeder beliebige s-t-Schnitt C des aktuellen Graphen mindestens so groß wie der *cut_of_the_phase* ist.

Definitionen: Ein Knoten $v \neq a$ gilt als **aktiv** (zu C), wenn v und der Knoten v' (der *unmittelbar* vor v zu A hinzugefügt wurde) in unterschiedlichen Partitionen von C liegen. Sei w(C) das Gewicht des beliebig gewählten, aber festen Schnittes C. Sei A_v die Menge aller Knoten in A vor der Addition von v. Dann ist C_v der Schnitt, der sich aufgrund des Schnittes C auf der Menge $A_v \cup \{v\}$ ergibt und w(C_v) das Gewicht dieses Schnittes.

Beispiel:



Die gestrichelte Linie markiert den s-t-Schnitt C, welcher aus $\{a, d, t\}$ und $\{b, c, s\}$ besteht. Wähle Knoten v als b, da b ein aktiver Knoten ist. A_b besteht aus dem Knoten a ($A_b = \{a\}$). C_b ist der Schnitt, der sich ergibt, wenn der Schnitt C nur auf die Knotenmenge $A_b \cup \{b\}$ (grau markiert) angewendet wird, in diesem Fall also $\{a\}$ und $\{b\}$. Der Wert des Schnittes C_b ist 5.

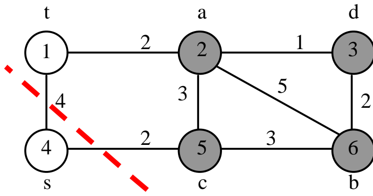
Es wird per Induktion für jeden aktiven Knoten v gezeigt, daß das Gesamtgewicht aller Kanten zwischen v und A_v nicht größer als das Gewicht des Schnittes C_v (für beliebiges C) ist. Formal: $w(A_v, v) \leq w(C_v)$.

Da t (der letzte zu A hinzugefügte Knoten) immer aktiv ist, weil s der vorletzte zu A hinzugefügte Knoten ist, folgt somit $w(A_t, t) \leq w(C_t)$. Weiterhin gilt, daß $w(A_t, t)$ nichts anderes als der *cut_of_the_phase* ist und, da C_t gleich C ist, w(C_t) dem Wert des Schnittes C entspricht. Daraus folgt, daß der *cut_of_the_phase* nicht größer als der Wert eines beliebigen s-t-Schnittes C ist.

Zurück zur Induktion und damit zunächst zur Verankerung:

Für den ersten aktiven Knoten x gilt sogar $w(A_x, x) = w(C_x)$. Da sich vor x alle Knoten in einer Partition versammeln, entspricht der Wert des Schnittes C_x genau den Kantenwerten zwischen x und allen vorher markierten Knoten.

Beispiel:



Die grauen Knoten $\{a, b, c, d\}$ gelten in Bezug auf den Schnitt C nicht als aktiv. Erst der Knoten s ist der erste aktive Knoten. Der Wert des Schnittes C_s entspricht dem Gewicht der Kante (s,c).

Induktionsschritt:
Sei die Induktionsannahme für alle aktiven Knoten bis zum aktiven Knoten v bewiesen. Sei nun der Knoten u der nächste aktive Knoten, der hinzugefügt wird. Dann gilt:

$$(1) \qquad w(A_u, u) = w(A_v, u) + w(A_u \setminus A_v, u)$$

Die Kanten zwischen A_u und u entsprechen den Kanten zwischen A_v und u und allen Kanten zwischen den Knoten aus $A_u \setminus A_v$ und u, da A_v eine echte Teilmenge von A_u ist. Damit sind natürlich auch die Gewichte identisch.
Es gilt weiterhin $w(A_v, u) \leq w(A_v, v)$, da v am engsten mit A_v verbunden ist (*most tightly connected*). Somit gilt:

$$(2) \qquad w(A_u, u) = w(A_v, u) + w(A_u \setminus A_v, u) \leq w(A_v, v) + w(A_u \setminus A_v, u)$$

Nach Induktionsannahme gilt jedoch $w(A_v, v) \leq w(C_v)$ und daraus folgt:

$$(3) \qquad w(A_u, u) \leq w(A_v, v) + w(A_u \setminus A_v, u) \leq w(C_v) + w(A_u \setminus A_v, u)$$

Da die Kanten zwischen $A_u \setminus A_v$ und u den Schnitt „schneiden“ (denn u ist der erste aktive Knoten nach v), leisten sie einen Beitrag zu $w(C_u)$, aber **nicht** zu $w(C_v)$. Andererseits ist $w(C_v)$ komplett in $w(C_u)$ „enthalten“, womit gilt: $w(C_v) + w(A_u \setminus A_v, u) = w(C_u)$. Daraus folgt letztendlich:

$$(4) \qquad \mathbf{w(A_u, u)} \leq w(C_v) + w(A_u \setminus A_v, u) = \mathbf{w(C_u)}$$

Damit ist der Induktionsschritt bewiesen und es folgt, daß der *cut_of_the_phase* nicht größer als der Wert eines beliebigen s-t-Schnittes C ist, mithin also ein minimaler s-t-Schnitt sein muß.

Bemerkung:

- Der eben bewiesene Satz gilt natürlich nicht für eine beliebige Wahl der Knoten *s* und *t*. Die Korrektheit läßt sich nur für den Fall beweisen, daß die Knoten in der vom Algorithmus beschriebenen Reihenfolge markiert werden. Diese Reihenfolge wird in Ungleichung (2) in den Beweis eingebracht (s.o.).

2.4 Laufzeitanalyse

Der Algorithmus MINIMUMCUT hat eine Laufzeit von $O(nm + n^2 \log n)$ für *n* Knoten und *m* Kanten. Da MINIMUMCUT im wesentlichen aus n-1 Aufrufen des Unterprogramms MINIMUMCUTPHASE besteht, genügt es demnach zu zeigen, daß MINIMUMCUTPHASE eine Laufzeit von $O(m + n \log n)$ hat.

Das Hauptproblem des Unterprogramms ist die Auswahl des nächsten Knotens, der zu *A* dazukommt (*most tightly connected vertex*). Dazu wird eine Prioritätswarteschlange aufgebaut, welche die Knoten aus $V \setminus A$ enthält. Die Warteschlange wird nach Schlüsseln sortiert, wobei der Schlüssel eines Knotens *v* die Summe der Kantengewichte von *v* nach *A* ist, also $w(A,v)$.
Jedesmal, wenn ein Knoten *v* in *A* eingefügt wird, muß die Warteschlange aktualisiert werden. Der Knoten *v* muß entfernt werden und der Schlüssel jedes Knotens *w* aus $V \setminus A$ muß um das Kantengewicht der Kante (v,w) erhöht werden, falls diese Kante existiert.

Die Operation EXTRACTMAX (den „nächsten" Knoten finden) muß *n*-mal durchgeführt werden. Die Operation INCREASEKEY (Schlüssel erhöhen) muß für jede Kante exakt einmal durchgeführt werden, wird insgesamt also *m*-mal benutzt.

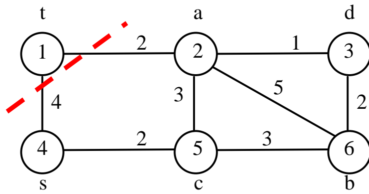
Eine INCREASEKEY-Operation läßt sich in konstanter Zeit durchführen, während eine EXTRACTMAX-Operation mittels Fibonacci-Heaps (Fredman und Tarjan [3]) in amortisierter Laufzeit von $O(\log n)$ realisierbar ist. Daraus ergibt sich also eine Gesamtlaufzeit von $O(m + n \log n)$ für einen Aufruf von MINIMUMCUTPHASE.

2.5 Abschließende Bemerkungen

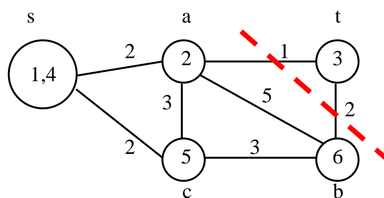
Der Algorithmus wurde inzwischen von Kurt Mehlhorn (Max-Planck-Institut Saarbrücken) implementiert und ist nun Bestandteil der Algorithmenbibliothek LEDA (Weiterführende Informationen unter http://www.mpi-sb.mpg.de/LEDA/index.html). Prinzipiell kann dies als Auszeichnung wegen des schnellen und einfachen Verfahrens verstanden werden. Es mag zwar asymptotisch schnellere Algorithmen geben, doch werden diese kaum eine derartig einfache Datenstruktur und Eleganz aufweisen können.

2.6 *Ausführliches Beispiel*

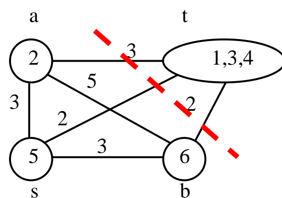
Abschließend folgt das bereits im Text kurz vorgestellte Beispiel in einer ausführlich kommentierten Version.



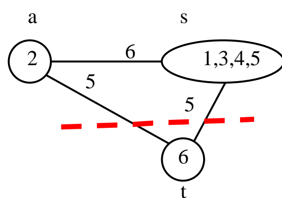
Der Startknoten a ist 2 und kommt in die Menge A. Knoten 6 hat mit einem Kantengewicht von 5 zu A den höchsten Wert und wird als b in A aufgenommen. Der Knoten 5 hat mit Kantengewicht 6 (3 zu a, 3 zu b) einen höheren Wert als Knoten 3 (Gewicht 3) oder Knoten 1 (Gewicht 2) und wird zu c. Knoten 3 wird zu d. Der vorletzte Knoten 4 erhält die Markierung s, der letzte Knoten 1 die Markierung t. Der Schnitt {t=1} ist der minimale s-t-Schnitt dieser Phase und hat den Wert 6. Anschließend werden die Knoten s und t zusammengefaßt und der aktuelle minimale Schnitt zwischengespeichert.



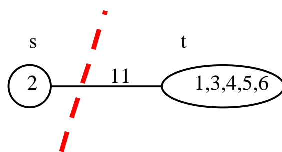
Knoten 2 als a ist erneut Startknoten und damit in A. Danach wird wieder Knoten 6 eingefügt, dann Knoten 5 und abschließend Knoten (1,4) als s und der Knoten 3 als t. Der neue minimale s-t-Schnitt dieser Phase ist {t=3} mit Wert 3. Wieder werden s und t zusammengefaßt und der jetzt kleinere minimale Schnitt zwischengespeichert. Diesmal kommt es übrigens zu dem Fall, daß s und t keine gemeinsame Kante besitzen.



Knoten 2 (a) liegt wieder in A. Danach wird wieder Knoten 6 eingefügt, dann Knoten 5 als s und abschließend Knoten (1,3,4) als t. Der neue minimale s-t-Schnitt dieser Phase ist {t=1,3,4} mit Wert 7. Wieder werden s und t zusammengefaßt. Der minimale Schnitt bleibt jedoch unverändert, da der neue Schnitt zu groß ist.



Knoten 2 ist erneut Startknoten. Danach wird der Knoten (1,3,4,5) eingefügt, da das Kantengewicht 6 am höchsten ist. Dieser Knoten ist s. Knoten 6 wird somit zu t. Der neue minimale s-t-Schnitt dieser Phase ist {t=6} mit Wert 10. s und t werden zusammengefaßt. Es gibt keine Änderung beim minimalen Schnitt.



Knoten 2 ist Startknoten und diesmal sofort s. Danach wird (1,3,4,5,6) als t in A eingefügt. Der minimale s-t-Schnitt ist trivialerweise 11 und der Algorithmus wird beendet (nachdem noch einmal alle Knoten zu einem “Superknoten” verschmolzen werden).

3 Ein stochastischer Ansatz zur Min-Cut-Berechnung

Motivation: Bisher wurde ein deterministischer Ansatz zur Min-Cut Berechnung vorgestellt. Es steht zu hoffen, daß die Berechnung minimaler Schnitte durch Hinzunahme des Zufallselements beschleunigt werden kann. Der folgende Abschnitt wird sich damit beschäftigen. Da es sich bei den benutzten Algorithmen ausnahmslos um Monte Carlo Algorithmen handelt, kann dabei nur versucht werden, die Erfolgswahrscheinlichkeit zu maximieren. Eine Erfolgsgarantie gibt es bei dieser Gattung von Algorithmen nicht.
Quelle: Karger, David R. und Stein, Clifford, A new approach to the Minimum Cut Problem [B].

3.1 Der Contract-*Algorithmus*

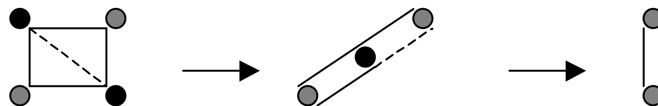
Der erste Ansatz für einen stochastischen Algorithmus in einem ungewichteten (Multi-) Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ mit n Knoten und m Kanten ist der folgende Algorithmus. In einem Multigraphen werden mehrere Kanten zwischen zwei Knoten zugelassen.

Algorithmus 1:
*Procedure Contract (**G**)*
*Solange **G** noch mehr als zwei Knoten enthält:*

> *Wähle zufällig eine Kante (**u**,**v**) aus **E**;*
> *Verschmelze die Knoten **u** und **v** zu einem neuen Knoten **u**;*
> *Alle Kanten der Form (**w**,**u**) oder (**w**,**v**) werden zu Kanten (**w**,**u**), für alle **w** aus **V**;*
> *Alle Kanten der Form (**u**,**v**) werden gelöscht;*

*Return **G**.*

Beispiel 1:



In dem Originalgraphen existieren zwei minimale Schnitte. Solange die beiden schwarzen Knoten nicht mit je einem der grauen Knoten verschmolzen werden, bleibt einer der beiden minimalen Schnitte übrig, ansonsten wird ein falscher Schnitt ausgegeben. Allerdings würde am Anfang ein grauer Knoten und ein schwarzen Knoten mit einer Wahrscheinlichkeit von 4/5 verschmolzen.

Contract wählt sich sukzessive eine Kante aus ***G*** aus, deren Randknoten er verschmilzt. Dabei wird der dabei entstandene Knoten in dem ersten der beiden Originalknoten gespeichert. Dies wiederholt der Algorithmus, solange noch mehr als zwei Knoten im Graphen vorhanden sind. Dazu benötigt der Algorithmus *n*-2 Iterationen. Dabei wird jede der beiden Mengen des Schnittes durch je einen Knoten repräsentiert. Der Schnitt, den dieser Algorithmus berechnet, besteht also aus den Kanten, die zwischen diesen beiden Knoten verlaufen. Nun stellen sich die folgenden Fragen:

1. Ist dieser Schnitt ein minimaler Schnitt?
2. Wie groß ist die Wahrscheinlichkeit, daß ein minimaler Schnitt die Kontraktionen auf zwei Knoten überlebt?

Lemma 1.1: Ein Schnitt (***A***, ***B***) wird vom *Contract*-Algorithmus nicht zerstört ⇔ Keine Kante des Schnittes wird kontrahiert.

Beweis: ⇒: Der Schnitt (***A***, ***B***) hat die Kontraktionen überstanden. Daraus folgt, daß alle Knoten der Menge ***A*** zu einem der Ausgabeknoten zusammengefaßt wurden und alle Knoten der Menge ***B*** zu dem anderen Knoten. Damit wurde während der Laufzeit des Algorithmus keine Schnittkante kontrahiert, da sonst ein Knoten der einen Menge in den *falschen* Ausgabeknoten eingefügt würde.

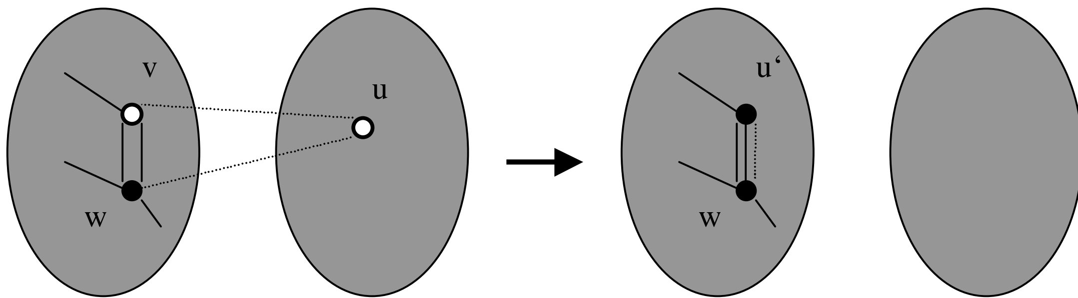
⇐: Keine Schnittkante wird kontrahiert. Z.Z.: Annahme: Dann verlaufen zu jeder Zeit zwischen zwei beliebigen Knoten entweder nur Schnittkanten oder nur Kanten, die nicht dem Schnitt angehören. Also können im Verlauf des Algorithmus nur Knoten verschmolzen werden, die sich in der gleichen Partition befinden.

Beweis durch Induktion

Induktionsverankerung: Die Annahme, daß zwischen zwei Knoten entweder nur Schnittkanten oder nur Kanten, die nicht dem Schnitt angehören, verlaufen, gilt für den Anfangsgraphen, also vor der ersten Kontraktion.

Induktionsschritt: Die Annahme gelte vor jeder Kontraktion, es ist z.Z., daß die Annahme nach jeder Kontraktion gilt. Dazu wird ein Beweis durch Widerspruch geführt.

Angenommen, es werden zwei Knoten *u* und *v* zu einem Superknoten *u*‘ verschmolzen, so daß ein Knoten *w* existiert, zu dem *u* Schnittkanten hat und *v* Kanten, die keine Schnittkanten sind. Damit würden zwischen *u*‘ und *w* parallele Kanten entstehen, von denen einige Schnittkanten und die anderen keine Schnittkanten sind. Dies würde die Annahme widerlegen.



Schnittkanten: ·········· Zu verschmelzende Knoten: ○

Da zwischen *v* und *w* keine Schnittkanten verlaufen, liegen *v* und *w* in der gleichen Partition. Da zwischen *u* und *w* Schnittkanten verlaufen, liegt *u* in der anderen Partition bezüglich *w* und *v*. Zwischen *u* und *v* verlaufen dann aber nur Schnittkanten, die bei der Verschmelzung kontrahiert werden. Das ist ein Widerspruch zur Voraussetzung, daß keine Schnittkanten kontrahiert werden.
Daraus folgt, daß die Annahme auch nach einer Kontraktion erfüllt ist. Damit ist die Behauptung gezeigt.

Theorem 1.2: Ein minimaler Schnitt in ***G*** wird durch *Contract* mit einer Wahrscheinlichkeit von $\Omega(n^{-2})$ nicht zerstört.
Beweis: Sei (***A***, ***B***) ein minimaler Schnitt mit ***c*** kreuzenden Kanten und damit einem Wert von ***c***. Im folgenden werden diese Kanten Schnittkanten genannt. Lemma 1.1 sagt aus, daß ein minimaler Schnitt die Kontraktion übersteht, wenn keine der Schnittkanten kontrahiert wird. Also müssen die verbleibenden Kanten des kontrahierten Graphen den minimalen Schnitt repräsentieren.
Nach jedem Kontraktionsschritt muß der minimale Schnitt immer noch $\geq$***c*** sein, da der Wert des minimalen Schnittes im ursprünglichen Graphen ***c*** ist. Wenn nun eine Kante (***u***,***v***) kontrahiert wird, darf die Kante nicht zwischen ***A*** und ***B*** verlaufen und der minimale Schnitt des Graphen ***G***/(***u***,***v***) hat damit immer noch den Wert ***c***.
Nun wird jedoch mit jeder Kontraktion die Anzahl der Knoten des Graphen reduziert. Betrachten wir nun den Zeitpunkt, an dem nur noch *r* Knoten übrig sind. Der Wert des minimalen Schnittes im reduzierten Graph ist immer noch $\geq$***c.*** Damit muß jeder Knoten einen Minimalgrad von $\geq$***c*** haben. Da noch *r* Knoten vorhanden sind, ist die minimale Anzahl von Kanten $\geq$***rc/2***, von denen ***c*** zum minimalen Schnitt gehören. Wenn nun eine zu kontra-

hierende Kante ausgewählt wird, ist die Wahrscheinlichkeit, daß diese in dem minimalen Schnitt liegt, $\le \boldsymbol{c}/(\boldsymbol{rc}/2) = 2/\boldsymbol{r}$. Wird nun eine Kante kontrahiert, so ist die Wahrscheinlichkeit, daß diese nicht im minimalen Schnitt liegt, durch $1\text{-}2/\boldsymbol{r}$ gegeben. Insgesamt müssen n-2 Knoten verschmolzen werden. Damit ist die Wahrscheinlichkeit, daß während all dieser Kontraktionen keine Kante des minimalen Schnittes angetastet wird durch

$$\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{3}\right)=\frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{2}{4}\frac{1}{3}=\frac{2}{n(n-1)}=\binom{n}{2}^{-1}=\Omega\left(n^{-2}\right)$$

gegeben.

Wird ein Graph der Form eines Ringes aus n Knoten betrachtet, so existieren in diesem Ring $(n^2\text{-}n)/2$ minimale Schnitte, die alle mit gleicher Wahrscheinlichkeit gefunden werden. Da die Wahrscheinlichkeit, in einem solchen Graphen mittels obigem Algorithmus einen minimaler Schnitt zu finden, gleich eins ist, ist demnach die Wahrscheinlichkeit, in diesem Graphen einen bestimmten minimalen Schnitt zu finden, $\Omega(n^{-2})$. Das war aber auch die Aussage von Theorem 1.2. Da ein solche Ringgraph die Maximalzahl möglicher minimaler Schnitte enthält, liefert Theorem 1.2 damit auch eine recht genaue Abschätzung für die Anzahl minimaler Schnitte in einem Graphen.
Allerdings ist die berechnete Wahrscheinlichkeit, daß ein minimaler Schnitt in einem normalen Graphen gefunden wird, erschreckend klein. Diese Wahrscheinlichkeit kann deutlich verbessert werden, indem der Algorithmus mehrfach hintereinander auf demselben Graphen ausgeführt wird. Dazu später mehr.

Hält der *Contract*-Algorithmus mit k verbleibenden Knoten, d.h. die äußere Schleife terminiert nicht erst, wenn nur noch zwei Knoten übrig sind, sondern bei k Knoten, so ergibt sich folgende Erweiterung zu Theorem 1.2:

Korollar 1.3: Ein minimaler Schnitt $(\boldsymbol{A}, \boldsymbol{B})$ übersteht die Kontraktion auf k Knoten mit einer Wahrscheinlichkeit von

$$\binom{k}{2}\Big/\binom{n}{2}=\frac{(k-1)k}{(n-1)n}=\Omega\left((k/n)^2\right),\quad k\ge 2$$

3.1.1 Gewichtete Graphen

Eine Erweiterung des ursprünglichen Contract-Algorithmus auf (ganzzahlig) gewichtete Graphen führt zu folgendem Algorithmus:

Algorithmus 2:

*Procedure Contract‘ (***G***)*

Solange **G** *aus mehr als zwei Knoten besteht*

    *Wähle eine Kante (***u,v***) mit einer Wahrscheinlichkeit gemäß ihres Gewichtes (*)*

    Verschmelze **u** *und* **v**

Return **G**

(*) Die Gewichte aller Kanten werden aufsummiert. Daraus ergibt sich eine Wahrscheinlichkeit, daß eine Kante gewählt wird, durch Kantengewicht/Gesamtgewicht. Damit hat eine Kante mit hohem Kantengewicht auch eine hohe Wahrscheinlichkeit, ausgewählt zu werden.

Der Algorithmus *Contract‘* entsteht aus dem ursprünglichen Algorithmus *Contract* durch die folgende Betrachtung: Eine Kante mit Gewicht ***w*** im gewichteten Graphen ***G*** kann in ***w*** parallele Kanten in einem ungewichteten Graphen ***G‘*** mit den gleichen Knoten abgebildet werden. Dadurch wird der minimale Schnitt nicht verändert. Aus diesem Grund könnte der ursprüngliche Algorithmus auf ***G‘*** angewendet werden. In diesem Fall ist die Wahrscheinlichkeit, daß wir eine Kante zwischen ***u*** und ***v*** zur Kontraktion auswählen, proportional zur Anzahl der parallel verlaufenden Kanten in ***G‘*** und damit zum Gewicht der abgebildeten Kante im ursprünglichen Graphen ***G***. Im weiteren wird nur noch der gewichtete *Contract‘*-Algorithmus betrachtet und der Einfachheit halber mit *Contract* bezeichnet. Die Analyse dieses Algorithmus gleicht der im ungewichteten Fall. Damit folgt:

Korollar 1.4: Der Algorithmus *Contract* findet einen minimalen Schnitten in **G** mit einer Wahrscheinlichkeit von $\Omega(n^{-2})$.

3.2 *Implementierung des* Contract-*Algorithmus*

Im folgenden werden mehrere Implementierungen des Contract-Algorithmus vorgestellt. Die erste Implementierung hat einen Platzbedarf von $O(n^2)$, eine später betrachtete Version benötigt nur $O(m)$, was bei dünn besetzten Graphen besser wäre. Allerdings ist die erste

Version leichter zu analysieren und schlägt sich mit der Zeit, ihre Datenstruktur zu aktualisieren, nicht als dominanter Faktor in der Laufzeitanalyse des Gesamtalgorithmus nieder.

In dieser ersten Version wird eine $n \times n$ Adjazenz-Matrix benutzt, die im folgenden mit $\boldsymbol{W}$ bezeichnet wird. Hierzu werden die Knoten durchnumeriert. $\boldsymbol{W}_{u,v}$ enthält das Gewicht der Kante $(\boldsymbol{u},\boldsymbol{v})$ oder äquivalent dazu die Anzahl der parallel verlaufenden Kanten im Multigraphen. Gibt es keine Kante $(\boldsymbol{u},\boldsymbol{v})$, so ist der Wert von $\boldsymbol{W}$ an dieser Stelle 0. Zusätzlich führen wir die Liste $\boldsymbol{D}(\boldsymbol{u})$ für alle Knoten $\boldsymbol{u}$ mit

$$D(u) = \sum_{v} W_{u,v}$$

Dieses Array enthält die Summe der Gewichte der Kanten, die von u abgehen. Im folgenden wird eine Iteration von *Contract* in zwei Teilschritte zerlegt: Zufällige Wahl einer Kante und Verschmelzung der begrenzenden Knoten.

3.2.1 Auswahl einer Kante

Eine der Basisoperationen ist die Auswahl einer Kante mit Wahrscheinlichkeit gemäß ihres Gewichtes. Ein erster Ansatz läuft wie folgt: Das aufsummierte Gewicht der ersten k Kanten $\boldsymbol{e}_1,...,\boldsymbol{e}_k$ mit Gewichten $\boldsymbol{w}_1,...,\boldsymbol{w}_k$ wird berechnet mit

$$W_k = \sum_{i=1}^{k} w_i$$

$\mathbf{W}_m$ ist damit das Gesamtgewicht aller vorhandenen Kanten. Danach wird zufällig eine natürliche Zahl r gewählt mit $0 \leq r \leq \mathbf{W}_m$. Als letztes wird binäre Suche angewendet, um die Kante $\mathbf{e}_i$ zu suchen mit $\mathbf{W}_{i-1} \leq r < \mathbf{W}_i$. Das ist in $O(\log \mathbf{W}_m)$ Zeit machbar. Hierbei wird vorläufig ignoriert, daß die Gewichte der Kante polynomiell in n sein müssen, um eine polynomielle Laufzeit zu gewährleisten. Von nun an wird angenommen, daß eine Black-Box *Random-Select* zur Verfügung steht. *Random-Select* erhält als Eingabe ein Gewichtsarray der Länge m und liefert in $O(\log m)$ Zeit eine natürliche Zahl zwischen 0 und m, wobei die Wahrscheinlichkeit, daß i zurückgeliefert wird, proportional zum Gewicht der Kante $\mathbf{e}_i$ ist. Später wird noch eine Implementierung von Random-Select vorgestellt, die in jedem Fall polynomiellen Zeitbedarf hat. Nun wird *Random-Select* benutzt, um eine zu kontrahierende Kante zu bestimmen. Dazu wird zuerst der Endpunkt $\boldsymbol{u}$ mit einer Wahrscheinlichkeit proportional zu $\boldsymbol{D}(\boldsymbol{u})$ gewählt und dann der zweite Endpunkt $\boldsymbol{v}$ mit einer Wahrscheinlichkeit proportional zu $\boldsymbol{W}_{u,v}$. Jede dieser Auswahlen erfordert $O(n)$ Zeit plus $O(\log n)$ Zeit für den Aufruf von Random-Select, also $O(n)$ Zeit insgesamt. Damit wäre

diese Art der Auswahl asymptotisch besser als die direkte Wahl einer Kante. Im folgenden Lemma wird die Korrektheit dieser Vorgehensweise bewiesen.

Lemma 2.1: Wird eine Kante, wie oben beschrieben, gewählt, dann ist die Wahrscheinlichkeit, daß diese Kante gewählt wurde proportional zu ihrem Gewicht $\boldsymbol{W}_{\mathrm{u,v}}$.

Beweis: Sei

$$\sigma = \sum_v D(v)$$

Damit ist $\sigma = 2\boldsymbol{W}_{\mathrm{m}}$. Dann ist

$$\begin{aligned} \mathrm{P}[(\boldsymbol{u},\boldsymbol{v}) \text{ wurde gewählt}] = {} & \mathrm{P}[\boldsymbol{u} \text{ wurde gewählt}]\mathrm{P}[(\boldsymbol{u},\boldsymbol{v}) \text{ wurde gewählt}|\ \boldsymbol{u} \text{ wurde gewählt}] \\ & +\mathrm{P}[\boldsymbol{v} \text{ wurde gewählt}]\mathrm{P}[(\boldsymbol{u},\boldsymbol{v}) \text{ wurde gewählt}|\ \boldsymbol{v} \text{ wurde gewählt}] \\ & = \frac{D(u)}{\sigma}\frac{W_{u,v}}{D(u)} + \frac{D(v)}{\sigma}\frac{W_{u,v}}{D(v)} = \frac{2W_{u,v}}{\sigma} = \alpha W_{u,v} \end{aligned}$$

Mit $\alpha = 1/\boldsymbol{W}_{\mathrm{m}}$. Damit ist die Wahrscheinlichkeit, eine Kante auf diese Art zu wählen, gleich der Wahrscheinlichkeit, eine Kante direkt auszuwählen.

3.2.2 Kontraktion einer Kante

Da nun eine Kante gewählt werden kann, muß die Kontraktion implementiert werden. Gegeben ist die Matrix ***W*** und das Array ***D***, die den Graphen ***G*** repräsentieren. ***W*** und ***D*** müssen folgendermaßen bei der Kontraktion einer Kante geändert werden, um den resultierenden Graphen ***G'*** zu repräsentieren: Sei die Kante (***u***,***v***) ausgewählt worden. Der aus der Verschmelzung von ***u*** und ***v*** resultierende Knoten soll in ***u*** gespeichert werden. Dazu muß die Zeile *u* mit der Summe der Zeilen ***u*** und ***v*** besetzt werden, und die Spalte ***u*** mit der Summe der Spalten ***u*** und ***v***. Danach wird die Zeile und die Spalte *v* auf null gesetzt. ***D*** wird analog auf den neuen Stand gebracht. Damit ergibt sich der folgende Algorithmus:

Algorithmus 3:

*Procedure Edge-Contract (***u***,***v***);*

*D(***u***) := D(***u***) + D(***v***) – 2*$W_{\mathbf{u,v}}$*;*

*D(***v***) := 0;* $\boldsymbol{W}_{\mathbf{u,v}} := \boldsymbol{W}_{\mathbf{v,u}} := 0$;

For alle Knoten **w** *außer* **u** *und* **v** *do*

$W_{\mathbf{u,w}} := W_{\mathbf{u,w}} + W_{\mathbf{v,w}}$;

$W_{\mathbf{w,u}} := W_{\mathbf{w,u}} + W_{\mathbf{w,v}}$;

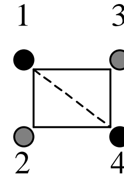
$W_{\mathbf{v,w}} := W_{\mathbf{w,v}} := 0$

***Beispiel 2**: Der Graph aus Beispiel 1 wird durch die Matrix W und das Array D(u)*

	1	*2*	*3*	*4*
1	*0*	*1*	*1*	*1*
2	*1*	*0*	*0*	*1*
3	*1*	*0*	*0*	*1*
4	*1*	*1*	*1*	*0*

u	*1*	*2*	*3*	*4*
D(u)	*3*	*2*	*2*	*3*

Knoten	*1*	*2*	*3*	*4*
Verschmolzen	*1*	*2*	*3*	*4*

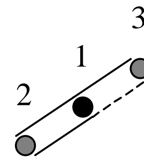


repräsentiert. Wird nun der erste Kontraktionsschritt durchgeführt, also der Knoten 1 und der Knoten 4 verschmolzen, wird diese Matrix überführt in die folgende Darstellung: Alle Kanten des Knoten 4 werden auf den Knoten 1 umgeleitet. Die Kante zwischen 1 und 4 wird gelöscht und die 4. Zeile und Spalte der Matrix wird auf null gesetzt. Zusätzlich wird das Array D(u) angepaßt.

	1	*2*	*3*	*4*
1	*0*	*2*	*2*	*0*
2	*2*	*0*	*0*	*0*
3	*2*	*0*	*0*	*0*
4	*0*	*0*	*0*	*0*

u	*1*	*2*	*3*	*4*
D(u)	*4*	*2*	*2*	*0*

Knoten	*1*	*2*	*3*	*4*
Verschmolzen	*1*	*2*	*3*	*1*

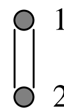


Die zweite Verschmelzung von 1 und 3 wird analog durchgeführt.

	1	*2*	*3*	*4*
1	*0*	*2*	*0*	*0*
2	*2*	*0*	*0*	*0*
3	*0*	*0*	*0*	*0*
4	*0*	*0*	*0*	*0*

u	*1*	*2*	*3*	*4*
D(u)	*2*	*2*	*0*	*0*

Knoten	*1*	*2*	*3*	*4*
Verschmolzen	*1*	*2*	*1*	*1*



Zu diesem Zeitpunkt kann der Wert des berechneten Schnittes durch Auslesen des Arrays D(u) bestimmt werden.

Der Algorithmus 3 läuft in O(n) Zeit. Dabei ist zu jeder Zeit das Gewicht einer Kante zwischen den Knoten $\boldsymbol{u}$ und $\boldsymbol{v}$ (bzw. die Anzahl der parallelen Kanten im ungewichteten Fall) in $\boldsymbol{W}_{u,v}$ gespeichert und damit in O(1) abrufbar. Liegt der Graph in dieser Darstellung vor, kann nach Terminierung des *Contract*-Algorithmus das Ergebnis, also der Wert des berechneten Schnittes in O(n), an den letzten beiden von null verschiedenen Stellen des Arrays ***D(u)*** ausgelesen werden. Um auf diese Weise den genauen Schnitt zu erhalten, muß

etwas mehr Aufwand getrieben werden. Dazu bietet es sich an, die Knoten durchzunumerieren und ein Array Verschmolzen(u) in der Größe n anzulegen. In diesem Array wird an der Position x gespeichert, mit welchem Knoten der Knoten $\boldsymbol{x}$ verschmolzen wurde. Initialisiert werden die Zellen des Arrays mit den Nummern der Knoten. Nach einem Kontraktionsschritt muß nun das Array wieder auf den neuesten Stand gebracht werden. Dies wird realisiert, indem innerhalb des Arrays die Nummer des zu löschenden Knotens gesucht wird und durch die Nummer des Knotens ersetzt wird, der diese beiden Knoten nach dem Schritt repräsentiert. Das ist in O(n) Zeit möglich. Damit bleibt die Zeit eines *Edge-Contract*-Durchlaufes auch mit detaillierter Knotenliste linear. Die Knotenliste wurde im Beispiel 2 schon berücksichtigt.
Sind die Knoten des Graphen am Ende zu zwei Knoten verschmolzen worden, so können die Schnittkanten leicht bestimmt werden, indem man zuerst die Knoten in die Gruppen $\boldsymbol{A}$ und $\boldsymbol{B}$ aufteilt. Im Array Verschmolzen(u) sind alle Knoten mit zwei Ausgabeknoten assoziiert worden. Diese beiden Ausgabeknoten können durch einen Durchlauf durch das Array $\boldsymbol{D}(\boldsymbol{u})$ ermittelt werden. Danach werden die Kanten identifiziert, die kreuzend zwischen den beiden Mengen laufen. Das ist in O(n^2) Zeit mit Hilfe der Matrix möglich. Bei Multigraphen ist ein Wert $\boldsymbol{W}_{u,v}$ der Matrix als die Zahl der parallelen Kanten zu interpretieren.
Das führt zu dem folgenden Korollar:

Korollar 2.2: Der *Contract'*-Algorithmus kann in O(n^2) Zeit ausgeführt werden.

Wenn der ursprüngliche Graph auf zwei Knoten $\boldsymbol{u}$ und $\boldsymbol{v}$ reduziert wurde, ist der Wert des Schnittes in $\boldsymbol{W}_{u,v}$ gespeichert. Falls der Graph auf k Knoten reduziert werden soll, so kann dies mit n-k Kanten-Kontraktionen erreicht werden. Im weiteren soll diese Kontraktion auf k Knoten durch *Contract*($\boldsymbol{G}$, k), also durch den ursprünglichen *Contract*-Algorithmus, dessen äußere Schleife nach der Reduktion des Eingabegraphen auf k Knoten terminiert, bewerkstelligt werden. Dieser Algorithmus liefert einen Graphen $\boldsymbol{G}'$ mit k Knoten in O(n^2) Zeit. Dabei ist die Wahrscheinlichkeit, daß ein bestimmter minimaler Schnitt die Kontraktion übersteht nach Korollar 1.3 durch

$$\binom{k}{2} \Big/ \binom{n}{2}$$

gegeben.

Der Algorithmus kommt auch mit einem Platzbedarf von O(m) aus, wenn statt der Adjazenz-Matrix die Adjazenzlisten-Darstellung gewählt wird. Dabei werden alle Kanten, die in einen Knoten $\boldsymbol{v}$ hineinführen, in einer solchen Liste organisiert. Zusätzlich werden die zwei Kopien einer Kante ($\boldsymbol{u}$,$\boldsymbol{v}$) und ($\boldsymbol{v}$,$\boldsymbol{u}$) mit Zeigern markiert. Wenn nun zwei Knoten $\boldsymbol{u}$ und $\boldsymbol{v}$ verschmolzen werden, muß die Liste von $\boldsymbol{u}$ und $\boldsymbol{v}$ auf den neuen Stand gebracht werden (nach den gleichen Regeln wie oben). Die Ergebnisliste ist zwar unsortiert, das kann allerdings mit Hilfe eines Bucket-Sort in O(n) Zeit behoben werden. Damit bleibt die Ausführungszeit des Gesamtalgorithmus O(n^2).

3.3 Der Rekursive Contract'-Algorithmus

Der Kontraktions-Algorithmus kann benutzt werden, um einen minimalen Schnitt zu finden, allerdings ist die Wahrscheinlichkeit, daß der Algorithmus wirklich den minimalen Schnitt findet, $\Omega(n^{-2})$. Um die Erfolgswahrscheinlichkeit zu maximieren, kann dieser Algorithmus ($dn^2 \ln n$)-mal ausgeführt werden und den kleinsten gefundenen Schnitt ausgeben, der bei allen Durchläufen gefunden wird. Die einzige Chance, daß diese Art der Suche keinen minimalen Schnitt findet, besteht darin, daß in allen Durchläufen kein minimaler Schnitt gefunden wird. Die Wahrscheinlichkeit dafür kann mit

$$\left(1-\frac{1}{n^2}\right)^{dn^2 \ln n} \leq e^{-d \ln n} \leq n^{-d}$$

veranschlagt werden. Die Laufzeit des Algorithmus ist dann allerdings O($n^4 \ln n$) und damit höher als die eines deterministischen Algorithmus. Aus diesem Grund wird im folgenden der vorgestellte *Contract'*-Algorithmus in einen rekursiven *Contract* Algorithmus verpackt. Die Idee dahinter ist, daß der neue Algorithmus die Problemgröße der O($n^2 \log n$) Aufrufe des Kontraktions-Algorithmus reduziert.

Dazu jetzt ein paar Grundgedanken: Die erste Kontraktion zweier Kanten hat mit $2/n$ eine kleine Wahrscheinlichkeit, eine Kante eines minimalen Schnittes zu kontrahieren. Auf der anderen Seite hat die letzte Kontraktion eine Wahrscheinlichkeit von 2/3 eine solche Kante zu kontrahieren. Aus diesem Grunde arbeitet der ursprüngliche Algorithmus am Anfang deutlich besser als gegen Ende seiner Laufzeit. Diese Chancen könnten aufgebessert werden, wenn anfangs ein schneller probabilistischer Algorithmus verwendet wird und gegen Ende auf einen langsameren Algorithmus mit besseren Erfolgsaussichten umgesattelt wird.

Eine Möglichkeit hierzu besteht darin, daß ein deterministischer Min-Cut Algorithmus für diese Zwecke genutzt wird. Dies bringt auch bessere Ergebnisse. Allerdings stellt sich her-

aus, daß es besser ist, statt einen Durchlauf des Kontraktions-Algorithmus zwei solche Durchläufe durchzuführen. Das führt zu folgendem Algorithmus:

Algorithmus 4:

*Recursive-Contract(***G***,* n*)*

Eingabe: Ein Graph **G** *mit* n *Knoten*

Falls **G** *weniger als 7 Knoten hat*

Dann

  G'*:= Contract(***G***, 2)*

  Ausgabe: Das Gewicht der Kanten zwischen den beiden resultierenden Knoten

Sonst Wiederhole zweimal

  G' *:= Contract(***G***,* $\lceil n/(\sqrt{2}) + 1 \rceil$*)*

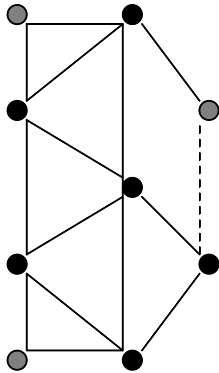
  *Recursive-Contract(***G'***,* $\lceil n/(\sqrt{2}) + 1 \rceil$*)*

  Ausgabe: Das Minimum der beiden Resultate

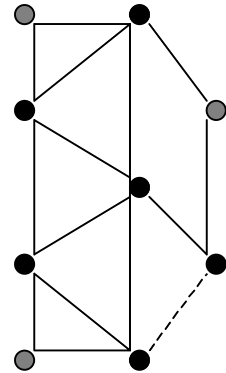
Dieser Algorithmus führt zwei unabhängige Aufrufe durch. In jedem dieser beiden Aufrufe wird der Graph partiell so kontrahiert, daß die Wahrscheinlichkeit, eine Kante des minimalen Schnittes zu kontrahieren, relativ klein bleibt. Mit der Reduktion des Graphen auf $\lceil n/(\sqrt{2}) + 1 \rceil$ Knoten ist die Wahrscheinlichkeit, daß der minimale Schnitt nicht angetastet wird, größer als 50%, so daß erwartet werden kann, daß einer der Aufrufe den minimalen Schnitt nicht zerstört. Dieser Algorithmus liefert nur den Wert eines minimalen Schnittes. Werden die Knoten allerdings, wie oben beschrieben, zusätzlich in einem Array verwaltet, so läßt sich der Schnitt in Linearzeit ausgeben. Allerdings steigt damit der Platzbedarf des Algorithmus deutlich.

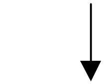
Beispiel 3:

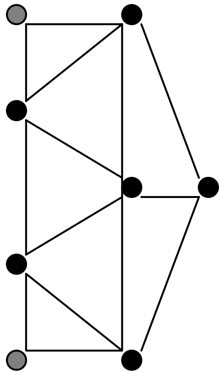
Es gilt, den minimalen Schnitt des folgenden Graphen zu berechnen:



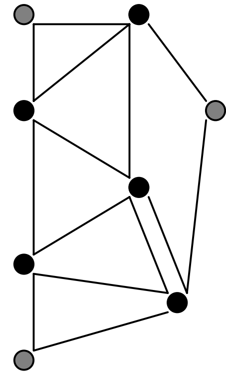
Im ersten Schritt des Algorithmus wird dieser Graph zweimal zu je einem Graphen mit 8 Knoten reduziert. Die Knoten, von denen einer abgespalten werden müßte, um einen minimalen Schnitt zu realisieren, sind grau markiert. Die Kante, die Contract ausgewählt hat, wurden gestrichelt dargestellt.







Für jeden der beiden auf diese Weise entstandenen Graphen würde nun ein rekursiver Aufruf des Algorithmus ausgeführt, um zu versuchen, den minimalen Schnitt zu finden. Da der Contract zu diesem Zeitpunkt den Graphen immer nur um einen Knoten reduziert, wird das Beispiel am Ende des Textes weitergeführt.



Lemma 3.1: Der Algorithmus *Recursive-Contract* läuft in $O(n^2 \log n)$ Zeit.

Beweis: Eine Rekursion besteht aus zwei unabhängigen Kontraktionen des Graphen auf $\lceil n/\sqrt{2} + 1 \rceil$ Knoten, gefolgt von einem rekursiven Aufruf des Algorithmus. Die Kontraktionen können jeweils in $O(n^2)$ Zeit durchgeführt werden (s.o.). Damit ergibt sich folgende Rekursionsgleichung für die Laufzeit:

$$T(n) = 2\left(n^2 + T\left(\left\lceil \frac{n}{\sqrt{2}} + 1 \right\rceil\right)\right)$$

Diese Lösung dieser Rekursionsgleichung kann laut Skript *Theoretische Informatik I* von Professor Dr. Schnitger, Satz 1.1 [12] mit $T(n) = O(n^2 \log n)$ abgeschätzt werden.

Lemma 3.2: Der Algorithmus Recursive-Contract benötigt $O(n^2)$ oder $O(m \log(n^2/m))$ Platz, abhängig von der Implementierung.

Beweis: Der Graph muß auf jeder Ebene der Rekursion gespeichert werden. Dabei ist die Größe des Graphen auf der *k*-ten Ebene der Rekursion durch die Rekursionsgleichungen $n_1 = n$, $n_{k+1} = \lceil n_k/\sqrt{2} + 1 \rceil$ gegeben. Wird die $O(n^2)$-Platz Implementierung benutzt, dann ist der benötigte Platz durch

$$O\left(\sum_k n_k^2\right) = O\left(n^2\right)$$

definiert. Um dieses Ergebnis zu verbessern, kann die Variante von *Contract* benutzt werden, die linearen Platzbedarf hat, die hier aber nicht vorgestellt wird. Da der Graph auf keiner Ebene mehr als $\min(m, n_k^2)$ Kanten hat, kann er pro Ebene in $O(\min(m, n_k^2))$ Platz gespeichert werden. Das führt zu $\Sigma_k\, O(\min(m, n_k^2)) = O(m \log(n^2/m))$ Platzbedarf.

Diese Analyse zeigt, daß die Laufzeit des *Recursive-Contract* Algorithmus akzeptabel ist. Allerdings ist die einzige Schranke für die Anzahl an Kanten, die angenommen werden kann, $O(n^2)$ und damit die benötigte Zeit, den Graphen auf $\lceil n/(\sqrt{2}) + 1 \rceil$ Knoten zu reduzieren auch durch $O(n^2)$ gegeben. Des weiteren ist durch die maximale Anzahl von n^2 Blättern im Rekursionsbaum eine Laufzeit von $\Omega(n^2)$ für *Recursive-Contract* gegeben, unabhängig von der Laufzeit von *Contract*. Deshalb wird die Linearzeit-Version des *Contract*-Algorithmus, die jedoch hier nicht vorgestellt wird, auch keinen globalen Geschwindigkeitsvorteil bringen.

Als nächstes muß die Wahrscheinlichkeit analysiert werden, mit der obiger Algorithmus einen bestimmten minimalen Schnitt findet. Es kann gesagt werden, daß der Algorithmus einen bestimmten minimalen Schnitt findet, wenn dieser mit einem Blatt im Rekursionsbaum korrespondiert.

Lemma 3.3: Der *Recursive-Contraction* Algorithmus findet einen bestimmten minimalen Schnitt mit einer Wahrscheinlichkeit von $\Omega(1/\log n)$.

Beweis: Angenommen, ein bestimmter minimaler Schnitt wurde bis zu einem bestimmten Knoten im Rekursionsbaum nicht zerstört. Dieser Schnitt wird es bis zu einem Blatt unter dem Knoten schaffen, wenn zwei Bedingungen erfüllt sind:

1. Er muß durch eine der beiden Aufrufe von *Contract* in diesem Knoten intakt belassen werden und
2. Er muß von dem auf (1.) folgenden rekursiven Aufrufe gefunden werden.

Jeder der zwei Aufrufe hat eine Erfolgswahrscheinlichkeit, einen minimalen Schnitt zu finden, gleich dem Produkt der Wahrscheinlichkeiten, daß der Schnitt die Kontraktion übersteht, und daß er von einem rekursiven Aufruf gefunden wird. Die Wahrscheinlichkeit, daß der Schnitt die Kontraktion überlebt, ist laut Korollar 1.3 mindestens

$$\left(\frac{\left(\left\lceil n/\sqrt{2}+1\right\rceil\right)\left(\left\lceil n/\sqrt{2}+1\right\rceil-1\right)}{n(n-1)}\right)\geq\frac{1}{2}$$

In den Aufrufen von *Contract* ist die Wahrscheinlichkeit mindestens 1/15 (mit 6 oder weniger Knoten). Das führt zu der folgenden Rekursionsgleichung P(n) für einen Graphen mit n Knoten:

$$P(n)\geq\begin{cases}1-\left(1-\frac{1}{2}P\left(\left\lceil\frac{n}{\sqrt{2}}+1\right\rceil\right)\right)^2 & \text{falls } n\geq 7\\ \frac{1}{15} & \text{sonst}\end{cases}$$

Da es ausreichend ist, eine untere Schranke für die Wahrscheinlichkeit zu berechnen, kann an dieser Stelle die Gleichheit angenommen werden. Diese Rekursionsgleichung kann mit einer Variablenersetzung gelöst werden. Sei p_k die Erfolgswahrscheinlichkeit auf der k-ten Ebene der Rekursion, wobei ein Blatt auf Ebene 0 liegt. Damit kann die Gleichung wie folgt umgeschrieben werden:

$$\begin{aligned}p_0 &= \frac{1}{15}\\ p_{k+1} &= 1-\left(1-\frac{1}{2}p_k\right)^2 = p_k-\frac{1}{4}p_k^2\end{aligned}$$

Sei nun $z_k = 4/p_k - 1$, also $p_k = 4/(z_k + 1)$. Wird dieses in das obige System eingesetzt, so ergibt sich:

$$\begin{aligned}z_0 &= 59\\ z_{k+1} &= z_k+1+\frac{1}{z_k}\end{aligned}$$

Da $z_k \geq 1$ ist, ergibt sich per Induktion, daß $k < z_k < 59 + 2k$. Damit ist $z_k = \Theta(k)$, woraus folgt, daß $p_k = \Theta(1/k)$ ist. Daraus folgt:

$$P(n)\geq p_{2\log_2 n+O(1)}=\Theta\left(\frac{1}{\log n}\right)$$

Damit ist gezeigt, daß die Wahrscheinlichkeit, einen bestimmten minimalen Schnitt zu finden, bei $\Omega(1/\log n)$ liegt.

Diese Wahrscheinlichkeit ist deutlich höher, als die des ursprünglichen *Contract*-Algorithmus. Allerdings ist sie auch zu niedrig, um minimale Schnitte gesichert finden zu können. Aus diesem Grund wird der *Recursive-Contract*-Algorithmus auch mehrfach auf einem Graphen wiederholt, um ein besseres Ergebnis erzielen zu können.

Theorem 3.4: Alle minimalen Schnitte in einem willkürlich gewichteten Graphen mit n Knoten und m Kanten können mit hoher Wahrscheinlichkeit in $O(n^2 \log^3 n)$ Zeit und $O(m \log (n^2/m))$ Platz gefunden werden.

Beweis: Es existieren maximal $(n^2-n)/2$ minimale Schnitte in einem Graphen. Wird *Recursive-Contract* $O(\log^2 n)$-mal wiederholt, dann ist die Wahrscheinlichkeit durch $O(1/n^4)$ beschränkt, einen bestimmten minimalen Schnitt nicht zu finden. Also ist die Wahrscheinlichkeit, einen der maximal $(n^2-n)/2$ minimalen Schnitte zu verpassen, durch $O((n^2-n)/n^4) = O(1/n^2)$ nach oben beschränkt.

Algorithmus 4 benötigt keine besonderen Datenstrukturen und ist damit leicht zu implementieren. Der gesamte Durchlauf kann als Berechnungsbaum aufgefaßt werden, wobei jede Kante des Baumes einer Kontraktion um ca. den Faktor $\sqrt{2}$ entspricht. Ein Blatt entspricht der Kontraktion des Graphen zu zwei Knoten und definiert damit einen potentiellen minimalen Schnitt, aber auf jeden Fall einen Schnitt. Die Tiefe dieses Baumes ist dabei $2 \log n + O(1)$ und er hat $O(n^2)$ Blätter. Damit ist der Algorithmus schneller als n^2 Durchläufe des ursprünglichen *Contract*-Algorithmus. Für die detaillierte Ausgabe eines minimalen Schnittes als Knotenmenge wird allerdings pro Schnitt $O(n)$ Zeit benötigt. Bei maximal $O(n^2)$ Schnitten kann damit alleine für die Ausgabe ein Zeitbedarf von $O(n^3)$ entstehen. Allerdings treten einzelne Schnitte mehrfach auf, so daß diese Zeit eingeschränkt werden kann, indem solche Mehrfachnennungen unterdrückt werden. Das jedoch kann mit einer Hashing-Technik, die von Applegate et al.[1] vorgestellt wurde, erreicht werden: Jedem Knoten wird ein $\log n$-Bit Schlüssel zugeordnet. Werden zwei Knoten verschmolzen, so werden deren Schlüssel mit einer exclusive-or Verknüpfung kombiniert. Mit einer hohen Wahrscheinlichkeit entstehen dabei keine zwei Schnitte mit identischen Knoten-Schlüsseln. Wenn nun alle gefundenen Schnitte ausgegeben werden sollen, so müssen nur die Schlüssel dieser Schnitte verglichen werden, um Doppelnennungen auszuschließen.

3.4 Streng polynomielle zufällige Auswahl

Im folgenden wird gezeigt, daß der Algorithmus *Recursive-Contract* in der behaupteten Zeit von $O(n^2 \log^3 n)$ läuft, indem eine geeignete *Random-Select*-Funktion zur Verfügung gestellt wird. Die Eingabe dieser Funktion ist ein Array $\boldsymbol{W}$ der Länge m. Dieses Array wird kumulativ aus den Gewichten $\boldsymbol{w}_i$ erzeugt mit $\boldsymbol{W}_k = \Sigma_{i \leq k}\, \boldsymbol{w}_i$. Random-Select soll einen Index i wählen mit einer Wahrscheinlichkeit proportional zum Gewicht $\boldsymbol{w}_i$. Es ist zu zeigen, daß die Funktion *Random-Select* in der gewünschten Zeitspanne läuft.

Sei $\boldsymbol{M} = \boldsymbol{W}_m$ die Summe aller Gewichte. Wenn die Kantengewichte $\boldsymbol{w}_i$ und damit auch $\boldsymbol{M}$ polynomiell in m sind, wird die Standardmethode benutzt, um *Random-Select* mit einem Zeitbedarf von $O(\log m)$ zu implementieren: Generiere eine uniforme ($\log \boldsymbol{M}$)-Bit Zufallszahl k im Bereich $[0, \boldsymbol{M}]$ und gebe den Wert i zurück mit $\boldsymbol{W}_{i-1} \leq k < \boldsymbol{W}_i$.

Falls sich die Gewichte der Kanten zu $\boldsymbol{M}$ addieren und $\boldsymbol{M}$ nicht mehr polynomiell in m ist, wird für den Algorithmus ein Zeitbedarf von $\Omega(\log \boldsymbol{M})$ benötigt. Im folgenden wird gezeigt, daß sich diese Auswahl mit einem vernachlässigbaren Fehler auch in $O(\log m)$ bewerkstelligen läßt.

Angenommen es sei bekannt, daß nur t Aufrufe von *Random-Select* in der Laufzeit des Algorithmus aufträten. Um ein Kante auszuwählen, auch wenn die Summe der Gewichte superpolynomiell in m ist, wird $\boldsymbol{N} = tm^4$ gesetzt und eine uniforme Zufallszahl s aus $[0, \boldsymbol{N}]$ berechnet. Nun wird die Kante i ausgewählt mit $\boldsymbol{W}_{i-1} \leq \boldsymbol{W}_m s/\boldsymbol{N} < \boldsymbol{W}_i$. Die auf diese Weise gewählte Kante wird sich von der mit der ursprünglichen Methode gewählten Kante nur dann unterscheiden, wenn $\boldsymbol{W}_n s/\boldsymbol{N}$ und $\boldsymbol{W}_n(s+1)/\boldsymbol{N}$ verschiedene Kanten referieren. Das kann höchstens m-mal passieren, so daß die Wahrscheinlichkeit dafür bei $m/\boldsymbol{N} = O(1/tm^3)$ liegt. Da dieser Ansatz schon einen Fehler enthält, kann auch ein Zufallsgenerator benutzt werden. Ein dabei auftretender Fehler wird dann vernachlässigt. Leider wird dieser Ansatz auch im Fall, daß nur noch zwei Möglichkeiten zur Auswahl stehen, eine Zufallszahl der Länge $\Omega(\log t)$ Bits erzeugen.

Im folgenden wird ein geschickterer Ansatz vorgeschlagen:

Angenommen, es werden $\log \boldsymbol{M}$ Zufallsbits erzeugt, um eine uniforme Zufallszahl im Bereich $[0, \boldsymbol{M}]$ zu generieren, aber es wird gehalten, sobald alle weiteren Bits keinen Unterschied mehr ausmachen. Bei einer Eingabe der Länge m wird der Bereich $[0, \boldsymbol{M}]$ in $2m$ gleich große Intervalle der Länge $\boldsymbol{M}/2m$ eingeteilt. Es werden $1+\log m$ Zufallsbits benötigt, um eines dieser Intervalle uniform zufällig zu wählen, was $O(\log m)$ Zeit für eine binäre

Suche in den kumulierten Kantengewichten erfordert. Falls das besuchte Intervall keinen der Werte ***W***i enthält (was mit einer Wahrscheinlichkeit von etwa ½ passiert, da *m* der 2*m* Intervalle nur einen Gewichtsbereich enthalten), dann ist ein eindeutiger Index gefunden worden und eine Vertiefung der Suche wäre unnötig, da dieser Bereich nicht wieder verlassen würde. Enthält das Intervall mehr als einen Indexwert, so wird das Intervall seinerseits in 2*m* gleich große Subintervalle aufgeteilt und es werden wieder 1+log *m* Zufallsbits benötigt um ein Subintervall auszuwählen. Diese Vorgehensweise wird solange wiederholt, bis ein eindeutiger Index ausgewählt wurde. Jede solche Aufteilung benötigt O(log *m*) Zeit und hat mit einer Wahrscheinlichkeit von ½ Erfolg.

Lemma 4.1: Bei einer Eingabelänge *m* ist die erwartete Laufzeit von *Random-Select* O(log *m*). Die Wahrscheinlichkeit, daß *Random-Select* mehr als (*t* log *m*) Zeit benötigt, ist $O(2^{-t})$.
Beweis: Jede binäre Suche in einem Subintervall benötigt O(log *m*) Zeit. Eine Intervallsuche heißt erfolgreich, wenn mit ihr ein eindeutiger Index bestimmt werden kann. Diese Wahrscheinlichkeit ist mindestens ½. Die totale Zeit der Suche, bis ein Erfolg eintritt, wird also von den erfolglosen Intervallsuchen vorher bestimmt. Da in jeder Intervallsuche die Fehlerwahrscheinlichkeit bei nahezu ½ liegt, ist die Wahrscheinlichkeit, daß bei einem Aufruf *t* erfolglose Intervallsuchen durchgeführt werden, $O(2^{-t})$ und die erwartete Anzahl von Fehlern vor dem ersten Erfolg ist 2.

Bemerkung: *Random-Select* kann also implementiert werden, in dem einzelne Zufallsbits erzeugt werden und angehalten wird, sobald das Ergebnis eindeutig ist.

Lemma 4.2: Wenn *Random-Select* *t*-mal mit Eingabelänge *m* aufgerufen wird, dann ist mit einer Wahrscheinlichkeit von $1 - e^{-\Omega(t)}$ der amortisierte Zeitbedarf jedes Aufrufes O(log *m*).
Beweis: Jede Intervallsuche benötigt O(log *m*) Zeit. Zu zeigen: Die amortisierte Anzahl der Intervallsuchen pro Aufruf ist O(1), die Gesamtzahl der Intervallsuchen ist also O(*t*). Die Anzahl der Erfolge in *t* Aufrufen von *Random-Select* ist *t*. Die Gesamtzahl der dafür notwendigen Intervallsuchen wird durch die Anzahl der Intervallsuchen vor dem *t*-ten Erfolg bestimmt. Diese Zahl ist aber die negative Binomialverteilung für den *t*-ten Erfolg mit einer Wahrscheinlichkeit von ½. Da die Wahrscheinlichkeiten gleich sind, kann die gleiche Anzahl von Fehlschlägen und Erfolgen angenommen werden, nämlich *t* in 2*t* Versuchen.

Theorem 4.3: Falls m Aufrufe von *Random-Select* durchgeführt wurden und jede Eingabe die Länge $m^{O(1)}$ hat, dann sind mit hoher Wahrscheinlichkeit die amortisierten Kosten von *Random-Select* O(log s) bei einer Eingabelänge von s.

Beweis: Die i-te Eingabe habe Länge m_i und sei $t_i = \lceil \log m_i \rceil$. Aus Lemma 4.2 ist bekannt, daß die erwartete Laufzeit von *Random-Select* bei Eingabe i O(t_i) beträgt.

Zu zeigen: Die totale Laufzeit von *Random-Select* für die komplette Eingabe ist mit hoher Wahrscheinlichkeit $O(\Sigma_i t_i)$. Dabei ist der längste Wert von t_i in O(log m).

Der i-te Aufruf von *Random-Select* heißt typisch, wenn mehr als 5 log m weitere Aufrufe mit dem Wert von t_i vorhanden sind, sonst atypisch. Da t_i durch O(log m) beschränkt ist, kann es höchstens $O(\log^2 m)$ atypische Aufrufe geben. Aus Lemma 4.1 und der Tatsache, daß $t_i = O(\log m)$ gilt, folgt, daß die Zeit für den i-ten Aufruf mit hoher Wahrscheinlichkeit in $O(\log^2 m)$ ist. Daher ist die Zeit für alle atypischen Aufrufe mit hoher Wahrscheinlichkeit $O(\log^4 m)$. Mit Lemma 4.2 folgt, falls der i-te Aufruf typisch ist, daß die amortisierten Kosten des i-ten Aufrufs mit hoher Wahrscheinlichkeit O(t_i) betragen. Deshalb kann die Gesamtlaufzeit aller Aufrufe mit

$O(\log^4 n + \Sigma_i t_i) = O(n+\Sigma_i t_i)$ angegeben werden. Da es m Aufrufe gibt, sind die amortisierten Kosten pro Aufruf $O(1 + t_i) = O(\log m_i)$.

Damit ist gezeigt, daß die amortisierten Kosten eines Aufrufs von Random-Select bei einer Eingabelänge von m O(log m) betragen. Mit diesem Ergebnis ist die berechnete Laufzeit des *Recursive-Select*-Algorithmus gesichert.

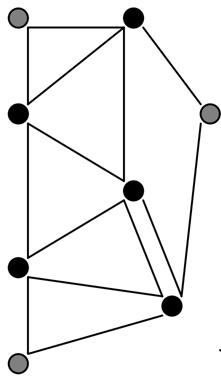
3.5 Ausblick

Durch Parallelisierung kann eine weitere Beschleunigung der Berechnung erreicht werden, wobei dafür zuerst gezeigt werden müßte, daß die vorgestellten Algorithmen das zulassen. Der Hauptaugenmerk liegt dabei auf der parallelen Kontraktion von Kanten.

Wie sich zeigt, kann mit Hilfe einer Zufallspermutation der Kanten eine zufällige Auswahl der Kanten simuliert werden. Danach wird überprüft, in wie viele Zusammenhangskomponenten der Graph zerfällt, wenn diese Kanten entfernt werden und abhängig von dieser Anzahl verfahren. Ziel dabei ist, zwei Zusammenhangskomponenten zu erzeugen und die kreuzenden Kanten zu bestimmen. Diese werden dann als angenommener minimaler Schnitt ausgegeben. Da dieser Algorithmus die gleichen Schranken wie der ursprüngliche

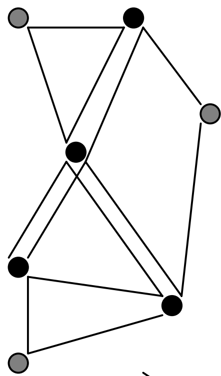
Contract-Algorithmus aufweist, muß auch dieser mehrfach ausgeführt werden. Allerdings kann durch die Parallelisierung einiges an Zeitkosten eingespart werden.
Eine weitere Anwendung ergibt sich direkt aus der vorigen Betrachtung. Dabei handelt es sich um die Suche nach Mehrfachschnitten, also die Schnittkanten zwischen mehreren Zusammenhangskomponenten. Mit den sequentiellen Algorithmen ergibt sich ein Zeitbedarf für die Suche von $O(n^{2(r-1)}\log^2 n)$. Allerdings zeigt sich auch hier, daß die parallelen Versionen der Algorithmen gute Dienste leisten können.
Die Algorithmen können auch dazu benutzt werden, alle fast-minimalen Schnitte zu finden. Diese Schnitte liegen in einer α-Umgebung um den minimalen Schnitt, d.h. Schnitte deren Wert im Intervall $[\boldsymbol{c}, \alpha\boldsymbol{c}]$ liegen. Dabei liegt die dafür benötigte Rechenzeit in $O(n^{2\alpha}\log^2 n)$.

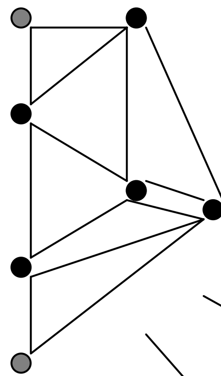
3.6 Ausführliches Beispiel von Recursiv-Contract



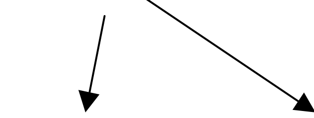
Es wird der linke Teilbaum von Beispiel 3 weiter betrachtet.

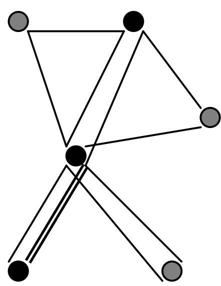
Der Graph wird im ersten Schritt kopiert und jede der Kopien wird auf $\lceil n/\sqrt{2} +1 \rceil$ Knoten reduziert. Da die ursprüngliche Anzahl der Knoten 8 war, haben die reduzierten Graphen also noch 7 Knoten.

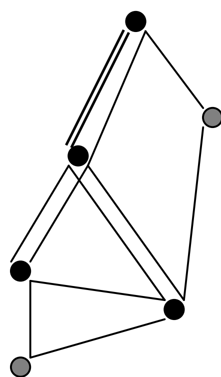


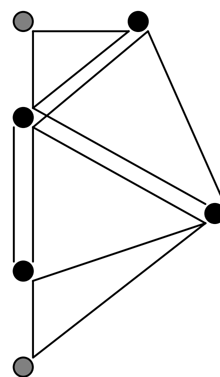


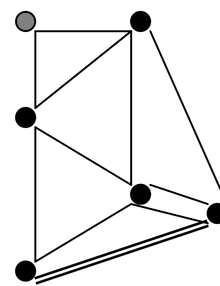
In den entstandenen Graphen sind immer noch mehr als sechs Knoten vorhanden. Deshalb wird auch er wieder kopiert und die Kopien kontrahiert.











Jeder der entstandenen Graphen hat jetzt sechs Knoten (die Minimalzahl, die bei einem Start mit mehr als 6 Knoten und dieser Aufteilung möglich sind). Auf jeden der Graphen wird nun *Contract*(G,2) angewendet, um einen Schnitt zu bestimmen. Das Minimum aus einem Ast des Rekursionsbaumes wird nach oben weitergegeben.

4 Literaturverweise

Behandelte Papers:

[A] STOER, MECHTHILD und WAGNER, FRANK 1997. **A simple Min-Cut Algorithm**. *Journal of the ACM*, Nr. 4, Juli 1997, Seiten 585-591.

[B] KARGER, D. R. und STEIN, C. 1996. **A New Approach to the Minimum Cut Problem**. *Journal of the ACM*, Nr. 4, Juli 1996, Seiten 601-640.

Sonstige Literatur:

[1] APPLEGATE, D. und BIXBY, D. und CHVÁTAL, V. und COOK, W. 1995, **Finding cuts in the tsp**. *Tech. Rep. 95-05*, DIMACS, Rutgers University, New Brunswick, NJ.

[2] FORD, L. R. und FULKERSON, D. R. 1956. **Maximal flow through a network**. *Can. J. Math. 8*, Seiten 399-404.

[3] FREDMAN, M. L. und TARJAN, R. E. 1987. **Fibonacci heaps and their use in improved network optimization algorithms**. *Journal of the ACM*, Nr. 3, Juli 1987, Seiten 596-615.

[4] GAREY, M. R. und JOHNSON, D. S. 1979**. Computers and Intractability: A Guide to the Theory of NP-Completeness**. *W. H. Freeman and Company*, San Francisco, Calif.

[5] GOMORY, R. E. und HU, T. C. 1961. **Multi-terminal network flows**. *J. Soc. Indust. Appl. Math. 9, Nr. 4, Dezember 1961*, Seiten 551-570.

[6] HAGERUP, T. 1999. **Flüsse in Netzwerken**. Mitschrift zur Vorlesung im Wintersemester 99/00 an der Johann Wolfgang Goethe-Universität Frankfurt am Main, Fachbereich Informatik.

[7] HAO, J. und ORLIN, J. B. 1994. **A faster algorithm for finding the minimum cut in a directed graph**. *J. Algorithms 17, 3*, Seiten 424-446.

[8] LAWLER, E. L. und LENSTRA, J. K. und RINOOY KAN, A. H. und SHMOYS, D. B. 1985. **The Traveling Salesman Problem**. *Wiley*, New York.

[9] NAGAMOCHI, H. und IBARAKI, T. 1992. **Computing edge-connectivity in multigraphs and capacitated graphs**. *SIAM J. Disc. Math. 5*, Seiten 54-66.

[10] NAGAMOCHI, H. und IBARAKI, T. 1992. **Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph**. *Algorithmica 7*, Seiten 583-596.

[11] QUEYRANNE, M. 1995. **A combinatorial algorithm for minimizing symmetric submodular functions**. *Proceedings of the 6th ACM-SIAM Symposium on Discrete Mathematics* ACM, New York, Seiten 98-101.

[12] SCHNITGER, G., 1998. **Theoretische Informatik I**. Skript zur Vorlesung an der Johann Wolfgang Goethe-Universität Frankfurt am Main, Fachbereich Informatik.