HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Frau
Antje Ahrens

Entwicklung von Algorithmen für Zuverlässigkeitsprobleme auf ungerichteten Graphen

2012

Fakultät **Mathematik/Naturwissenschaften/Informatik**

BACHELORARBEIT

Entwicklung von Algorithmen für Zuverlässigkeitsprobleme auf ungerichteten Graphen

Autorin:
Antje Ahrens

Studiengang:
Angewandte Mathematik

Seminargruppe:
MA08w1-B

Erstprüfer:
Herr Prof. Dr. Peter Tittmann

Zweitprüfer:
Herr Prof. Dr. Klaus Dohmen

Mittweida, 2012

Bibliografische Angaben

Ahrens, Antje: Entwicklung von Algorithmen für Zuverlässigkeitsprobleme auf ungerichteten Graphen, 71 Seiten, Hochschule Mittweida (FH), Fakultät Mathematik/Naturwissenschaften/Informatik

Bachelorarbeit, 2012

Referat

Die Zuverlässigkeitstheorie ist ein praxisnahes Forschungsgebiet. In dieser Arbeit wird ein erster Einblick in dieses Themengebiet gegeben und Hilfsmittel zur effektiven Berechnung der K-Zuverlässigkeit vorgestellt. Die vorgestellten Möglichkeiten wurden algorithmisch erfasst, implementiert und anhand der Implementierung einige Test hinsichtlich der K-Zuverlässigkeit durchgeführt.

I. Inhaltsverzeichnis

II. Tabellenverzeichnis

1 Motivation und Zielstellung

Die heutige, immer moderner werdende Gesellschaft ist von Netzwerken verschiedenster Art abhängig - z.B. vom Strom-, Wasser-, Funk- oder auch Verkehrsnetz. Deshalb stellt es sich als überaus wichtig dar, vorherzusagen, wie sicher ein Netzwerk funktioniert.
In den Anwendungsgebieten stellt sich häufig das Problem dar, dass Kanten ausfallen können. Um dieser Thematik gerecht zu werden, muss das Modell eines Graphen dahingehend erweitert werden, dass jeder Kante eine Existenz- oder Ausfallwahrscheinlichkeit zugeordnet wird. Die **Zuverlässigkeitstheorie** beschäftigt sich daher mit der Untersuchung von Graphen, bei denen Kanten nur mit einer gewissen Wahrscheinlichkeit vorhanden bzw. intakt sind, und wie sich das auf verschiedene Grapheneigenschaften auswirkt. Graphen, die mit einer solchen zusätzlichen Information ausgestattet sind, werden auch Netzwerke genannt.
Eines der Ziele auf diesem Gebiet ist die effektive Berechnung der Wahrscheinlichkeit, dass die gesamte oder eine Teilmenge der Knotenmenge eines Graphen miteinander verbunden ist. Für die folgenden Erkenntnisse stellt es sich dabei als bedeutsam dar, dass die Kanten mit einer bestimmten Wahrscheinlichkeit unabhängig voneinander ausfallen. Die errechnete Wahrscheinlichkeit für das Funktionieren eines Netzwerkes wird als **Zuverlässigkeit** des Netzwerkes bezeichnet.
Innerhalb dieser Arbeit werden daher die theoretischen Grundlagen der Zuverlässigkeit dargelegt, wichtige Werkzeuge zur Verkürzung der Laufzeit vorgestellt und diese dann algorithmisch erfasst werden. Der Nachweis der Funktionstüchtigkeit dieser Algorithmen wird im Anschluss durch eine zur Arbeit gehörende Implementierung in Python aufgezeigt.

2 Theorie der Netzwerkzuverlässigkeit

2.1 Graphentheoretische Grundlagen

Je nach Untersuchungsgebiet wird der Begriff des Graphen unterschiedlich definiert. Für die Problemstellung der Arbeit erscheint die folgende Definition sinnvoll:

Definition 2.1
Ein **(ungerichteter) Graph** $G = (V, E, \varphi)$ besteht aus der endlichen Knotenmenge V, der Kantenmenge E und der Inzidenzfunktion $\varphi : E \to V^{(2)}$, wobei $V^{(2)}$ die Menge aller zweielementigen Teilmengen von V sei. Es gilt $\varphi(e) = \{u, v\}$, wenn die Kante e die Knoten u und v verbindet. Man schreibt verkürzt $G = (V, E)$ und $e = \{u, v\}$.
Knoten, die durch eine Kante verbunden sind, werden als **adjazent** bezeichnet, Knoten und Kanten heißen zueinander **inzident**. Falls mehrere Kanten zwischen zwei Knoten verlaufen, spricht man von **parallelen Kanten**.

Zusätzlich zur Definition eines Graphen werden weitere Begriffe benötigt, um im Verlauf alle Argumentationen nachvollziehen zu können.

Definition 2.2
Eine **Kantenfolge** M in einem Graphen $G = (V, E)$ ist eine Folge von Knoten und Kanten $v_1 e_1 v_2 e_2 \ldots v_{n-1} e_{n-1} v_n$, mit $v_i \in V, \forall i = 1, \ldots, n$ und $e_j \in E, \forall j = 1, \ldots, n-1$, sodass $\forall i \in \{1, \ldots, n-1\}$ die Kante e_i die Endknoten v_i und v_{i+1} besitzt.

Definition 2.3
In einem Graphen $G = (V, E)$ bezeichnet man eine Kantenfolge als **Weg**, wenn sie keinen Knoten mehrfach enthält.
Die Länge eines Weges entspricht der Anzahl der in ihm liegenden Kanten.

Definition 2.4
Ein Graph $H = (W, F)$ heißt **Untergraph** eines Graphen $G = (V, E)$, wenn $W \subseteq V$ und $F \subseteq E$ gilt.

Definition 2.5
In einem Graphen $G = (V, E)$ bezeichnet man mit der **offenen Nachbarschaft** eines Knoten $v \in V$ die Menge aller Knoten, die zu v adjazent sind, geschrieben als:

$$N(v) = \{u | u \in V \backslash \{v\} \wedge \{u, v\} \in E\}$$

Die **geschlossene Nachbarschaft** $N[v]$ enthält zusätzlich noch den Knoten v selbst, d.h. $N[v] = N(v) \cup \{v\}$.

Definition 2.6
Der **Grad** $deg(v)$ eines Knoten $v \in V$ in einem Graphen $G = (V,E)$ ist die Anzahl der zu ihm inzidenten Kanten.

Definition 2.7
Ein Graph $G = (V,E)$ heißt **zusammenhängend**, wenn zwischen allen Paaren von Knoten aus V ein Weg in G existiert.
Eine Teilmenge $T \subseteq V$ ist dementsprechend in G zusammenhängend, wenn man für alle Paare aus T einen Weg in G findet.

Definition 2.8
Ein maximal zusammenhängender Untergraph $U = (W,F)$ eines Graphen $G = (V,E)$ heißt **Komponente** von G.
Maximal bedeutet, dass es zu keinem Knoten aus W adjazente Knoten bzw. inzidente Kanten gibt, die nicht in U liegen.

Definition 2.9
Der **Produktgraph** $G \times H$ zweier Graphen $G = (V,E)$ und $H = (W,F)$ besitzt die Knotenmenge $V \times W = \{(v,w) | v \in V, w \in W\}$. Knoten (v_1,w_1) und (v_2,w_2) sind genau dann adjazent, wenn $v_1 = v_2$ gilt und w_1 zu w_2 in H benachbart sind oder $w_1 = w_2$ gilt und v_1 zu v_2 in G adjazent ist.

Definition 2.10
Eine **Kette** κ in einem Graphen $G = (V,E)$ ist eine Folge von unterschiedlichen Knoten und Kanten $v_1\{v_1,v_2\}v_2\{v_2,v_3\}..v_{n-1}\{v_{n-1},v_n\}v_n$. Der Grad der Knoten v_2 bis v_{n-1} beträgt zwei, sie werden innere Knoten genannt. Die Endknoten v_1 und v_n weisen einen Grad echt größer zwei auf.
Eine Kette muss nicht notwendigerweise innere Knoten enthalten, somit bilden bereits zwei Knoten mit einer, die beiden verbindenden, Kante eine Kette.
Ketten, die innere Knoten besitzen, werden als echte Ketten bezeichnet.

Definition 2.11
Seien κ_1 und κ_2 zwei Ketten mit denselben Endknoten $u,v \in V$ in einem Graphen $G = (V,E)$, so nennt man $\chi = \kappa_1 \cup \kappa_2$ **Polygon**.

Definition 2.12
Ein Graph $G = (V,E)$ heißt **serienparallel** oder **SP-Graph**, wenn er durch folgende Operationen auf eine Kante reduziert werden kann:

1. Ersetzen von parallelen Kanten durch eine Kante
2. Entfernen eines Knoten von Grad 2 und Einfügen einer neuer Kante zwischen dessen Nachbarknoten

Im weiteren Verlauf wird ersichtlich, dass zur Berechnung der Zuverlässigkeit Graphen-

operationen benötigt werden, die nachfolgend definiert werden.

Definition 2.13
Aus einem Graph $G = (V,E)$ entsteht durch das **Entfernen einer Kante** $e \in E$ ein neuer Graph $G - e = (V, E \backslash \{e\})$.
Durch das **Entfernen eines Knoten** wird ebenso ein neuer Graph erzeugt, als $G - v$ bezeichnet, dabei werden neben dem Knoten v aus der Knotenmenge auch alle Kanten aus E entfernt, die zu v inzident sind, d.h. $G - v = (V \backslash \{v\}, E \backslash \{\{u,v\} \in E, u \in V\})$.
Man kann beide Operation auf Teilmengen $U \subseteq V$ bzw. $F \subseteq E$ erweitern, in dem alle Elemente dieser Mengen einzeln entfernt werden, wobei die Reihenfolge der Entfernung keine Rolle spielt.

Definition 2.14
Es sei $G = (V,E)$ ein Graph. Bei der **Knotenfusion** zweier Knoten $u,v \in V$ werden diese zu einem neuen Knoten $\overline{uv}$ verschmolzen. In den zu u oder v inzidenten Kanten werden die Knoten durch den neuen Knoten $\overline{uv}$ ersetzt. Im Speziellen entstehen Kanten $\{\overline{uv}, \overline{uv}\}$, falls u und v adjazent waren, welche entfernt werden. Die Bezeichnung des neuen Graphen lautet G_{uv}.

Definition 2.15
In einem Graphen $G = (V,E)$ findet bei der **Kantenkontraktion** von $e = \{u,v\} \in E$ zunächst das Entfernen von e und danach die Fusion von u und v statt. Der neu entstandene Graph wird mit G/e bezeichnet.

Satz 2.16
Ein Graph ist genau dann zusammenhängend, wenn nach Kontraktion aller Kanten die Knotenmenge nur noch aus einem Knoten besteht.

Beweis: Wenn ein Graph auf einen Knoten kontrahierbar ist, so existiert für jedes Knotenpaar eine Kantenfolge, über die die beiden verbunden sind. Dies ist aber gleichbedeutend mit der Tatsache, dass alle Paare verbunden sind und der Graph somit zusammenhängend ist. Die Umkehrung der Schlussfolgerung gilt. □

Aus diesem Satz kann gefolgert werden, dass ein Graph genau k Komponenten besitzt, wenn nach Kontraktion aller Kanten noch genau k isolierte Knoten vorhanden sind.

2.2 Zuverlässigkeitstheorie

Der in der Motivationsstellung benutzte Begriff der Zuverlässigkeit ist genau genommen nicht ganz korrekt, denn man muss dabei stets unterscheiden, worauf der Zusammenhang bezogen ist. Es ist präziser, den folgenden Begriff zu verwenden:

Definition 2.17
Unter der **K-Zuverlässigkeit** $Rel(G,K)$ eines Graphen $G=(V,E)$ versteht man die Wahrscheinlichkeit, dass die Knoten in einer Menge $K \subseteq V$ - die sogenannten Terminalknoten - alle untereinander verbunden sind, wenn die Kanten von G mit gegebener Wahrscheinlichkeit stochastisch unabhängig voneinander ausfallen.
Von besonderer Bedeutung sind zwei Spezialfälle. Der erste ist die allgemeine Zusammenhangswahrscheinlichkeit, in welcher als Terminalmenge die gesamte Knotenmenge V betrachtet wird. Dieser Fall entspricht der Frage, ob der Graph insgesamt zusammenhängt. Weiter interessiert häufig die st-Zuverlässigkeit. Hierbei handelt es sich um die K-Zuverlässigkeit mit den Terminalknoten $s,t \in V$. Die resultierende Frage ist, mit welcher Wahrscheinlichkeit ein Weg zwischen s und t in G existiert.

Im Folgenden sei vereinbart, dass zu einem Graphen $G=(V,E)$ auch stets eine Funktion $p : E \to [0,1]$ gegeben ist. Der Wert $p(e) := p_e$ soll für jede Kante $e \in E$ die Wahrscheinlichkeit sein, dass die Kante intakt ist.

Satz 2.18 (Dekomposition)
Sei $G=(V,E)$ ein Graph, $K \subseteq V$ die Menge der Terminalknoten und $e=\{u,v\} \in E$ eine beliebige Kante. Dann gilt:

$$Rel(G,K) = \begin{cases} 1, \textit{ wenn |K|=1 gilt} \\ 0, \textit{ wenn Knoten aus K in versch. Komponenten von G} \\ p_e \cdot Rel(G \backslash e, \tilde{K}) + (1-p_e) \cdot Rel(G-e,K), \textit{ sonst} \end{cases} \tag{2.1}$$

$\tilde{K}$ ist die Terminalmenge K ohne die Endknoten u,v der Kante e. Falls mindestens dieser Knoten ein Terminal war, so enthält $\tilde{K}$ zusätzlich noch den durch die Kontraktion entstandenen Knoten.

Beweis:
1. Fall: $|K|=1$
Wenn K die Mächtigkeit 1 besitzt, besteht K nur aus einem Knoten. Ein Knoten ist trivialerweise zusammenhängend, somit ist die K-Zuverlässigkeit 1.
2. Fall: Knoten aus K in verschiedenen Komponenten
Da Komponenten maximal zusammenhängende Untergraphen sind, existiert zwischen ihnen keine Verbindung. Somit kann aber zwischen Terminalknoten, die in unterschiedlichen Komponenten liegen, ebenso keine Verbindung existieren. Daher ist die K-Zuverlässigkeit 0.

3. Fall: $|K| > 1$, alle Knoten aus K in einer Komponente
Die frei wählbare Kante e hat genau zwei Zustände - intakt oder ausgefallen. Somit kann man die K-Zuverlässigkeit des Graphen G nach dem Prinzip der totalen Wahrscheinlichkeit zerlegen. Es gilt also:

$$\begin{aligned} Rel(G,K) =& p_e \cdot P(K \text{ in } G \text{ zusammenhängend} \mid e \text{ intakt}) \\ &+ (1-p_e) \cdot P(K \text{ in } G \text{ zusammenhängend} \mid e \text{ nicht intakt}) \end{aligned}$$

Funktioniert Kante e, so sind die Endknoten u und v stets verbunden. Durch das Kontrahieren der Kante würden die beiden zu einem neuen Knoten, für die folgende Betrachtung $\overline{uv}$ genannt, verschmelzen, wodurch die Verbundenheit der Knoten gewährleistet wäre. Allerdings ist es notwendig, sich dabei Gedanken über die Zugehörigkeit von u, v und $\overline{uv}$ zur Terminalmenge K zu machen.
Gehören sowohl u als auch v nicht zu K, hat die Operation keinen Einfluss auf K und muss somit nicht verändert werden. Falls jedoch mindestens einer der beiden ein Terminal ist, so repräsentiert $\overline{uv}$ einen Terminalknoten und dementsprechend sollte K um diesen Knoten erweitert werden. Die aus dem Graphen entfernten Knoten u und v sollten damit gleichzeitig auch aus K entfernt werden, da sie im Graphen nicht mehr existieren.
Es folgt, dass $P(K \text{ in } G \text{ zusammenhängend} \mid e \text{ intakt}) = Rel(G \backslash e, \tilde{K})$.
Ist e nicht intakt, so leistet die Kante keinen Beitrag zum Zusammenhang, kann also auch weggelassen werden. Die Terminalzugehörigkeit wird dabei in keiner Weise beeinflusst, da keine Knoten verändert werden.
Somit gilt, dass $P(K \text{ in } G \text{ zusammenhängend} \mid e \text{ nicht intakt}) = Rel(G-e, K)$. □

Mithilfe dieses Satzes kann man die K-Zuverlässigkeit rekursiv berechnen. Auch ohne genauere Analyse ist ersichtlich, dass sich ein exponentieller Aufwand in Abhängigkeit von der Kantenanzahl ergibt. Für eine praktische Anwendung müssen daher Methoden gefunden werden, die Anzahl der Kanten oder Knoten im Graphen zu verringern. Dieser Überlegung zufolge wurden die zuverlässigkeitserhaltenden Reduktionen entwickelt:

Definition 2.19
Bei einer **Reduktion** wird in einem Graphen $G = (V,E)$ mit seinen Terminalknoten $K \subseteq V$ eine bestimmte Knoten- und Kantenmenge durch andere ersetzt, so dass ein neuer Graph G' mit einer Menge K' vorliegt. Sie heißt **zuverlässigkeitserhaltend**, wenn gilt:

$$Rel(G,K) = \theta \cdot Rel(G',K') \tag{2.2}$$

Der Faktor θ ist für jede konkrete zuverlässigkeitserhaltende Reduktion für einen Graphen G spezifisch und muss eindeutig aus G berechenbar sein, d.h. es gilt $\theta = \theta(G)$.

Der neue Graph G' sollte dabei im Sinne der Berechenbarkeit seiner K-Zuverlässigkeit bessere Eigenschaften haben als der ursprüngliche Graph G, in diesem Falle also zumindest weniger Kanten, im Idealfall auch weniger Knoten enthalten. Es gibt eine Vielzahl von inzwischen bekannten Reduktionen, von denen die folgenden betrachtet wer-

den. In allen Grafiken sind Terminalknoten schwarz, Nichtterminals weiß ausgefüllt.

Definition 2.20
Es sei $G = (V,E)$ ein Graph, $K \subseteq V$ die Menge der Terminalknoten, sowie $v \in V$ ein Knoten mit $deg(v) = 1$ und dem Nachbarknoten $u \in V$. Dann entsteht durch die **Grad-1-Reduktion** für v ein Graph G' mit K', indem v aus G entfernt wird. K' ist definiert durch:

$$K' = \begin{cases} K \setminus \{v\} \cup u, \text{ wenn} v \in K \\ K, \text{ sonst} \end{cases}$$

Satz 2.21 (Grad-1-Reduktion)
Unter den Voraussetzungen der Definition ist die Grad-1-Reduktion entsprechend der aufgeführten Unterscheidung eine zuverlässigkeitserhaltende Reduktion.

$$\theta = \begin{cases} p_e, \textit{ wenn } v \in K, \textit{ } e \textit{ die zu } v \textit{ inzidente Kante} \\ 1, \textit{ wenn } v \in V \setminus K \end{cases} \tag{2.3}$$

Beweis:
Fall 1: Wenn v in K liegt, so muss die zu ihm inzidente Kante e funktionieren, da ansonsten der Knoten nicht mit den anderen Terminalknoten verbunden sein kann. Betrachtet man die Dekomposition in (2.1), so ergibt sich der hintere Summand zu 0. Die K-Zuverlässigkeit von G' mit K' hat den selben Wert wie die Wahrscheinlichkeit, dass G/e zusammenhängend ist, da es sich um isomorphe Graphen handelt.
Fall 2: Wenn v nicht in K liegt, ist es irrelevant, ob die inzidente Kante funktioniert oder nicht, folglich kann der Knoten v ohne Konsequenzen entfernt werden. □

Definition 2.22
Es sei $G = (V,E)$ ein Graph, K die Menge der Terminals und $e, f \in E$ zwei parallele Kanten zwischen den Knoten $u, v \in V$. Die **Parallelreduktion** erstellt G' aus G durch das Ersetzen der Kanten e und f durch eine neue Kante g mit $p_g = 1 - (1 - p_e) \cdot (1 - p_f)$. Die zu G' gehörige Terminalmenge K' ist gleich K.

Satz 2.23 (Parallelreduktion)
Die Parallelreduktion mit den Voraussetzungen der Definition ist eine zuverlässigkeitserhaltende Reduktion mit $\theta = 1$.

Beweis: Zwischen den Knoten u und v existiert genau dann keine Kante, wenn sowohl e als auch f nicht funktionieren, die Wahrscheinlichkeit für dieses Ereignis ist $(1 - p_e) \cdot (1 - p_f)$. Somit sind die beiden Knoten mit der Wahrscheinlichkeit mit $1 - (1 - p_e) \cdot (1 - p_f) = p_g$ adjazent. Es ergibt sich, dass man die beiden parallelen Kanten durch eine neue Kante ersetzen kann, wenn man deren Wahrscheinlichkeit auf p_g setzt. □

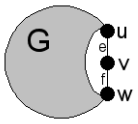
Satz 2.24 (Komponentenreduktion)
Wenn ein Graph $G = (V,E)$ aus zwei Komponenten besteht und eine dieser keine Terminalknoten enthält, so kann diese gelöscht werden, ohne dass sich dadurch die K-Zuverlässigkeit ändert. Es handelt sich um eine zuverlässigkeitserhaltende Reduktion mit $\theta = 1$.

Beweis: Wenn in einer Komponente keine Terminals vorkommen, so ist es unerheblich, wie dort der Zusammenhang gestaltet ist. □

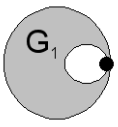
Definition 2.25
Es sei $G = (V,E)$ ein Graph mit der Terminalmenge $K \subseteq V$, sowie den Knoten $u,v,w \in V$ mit $deg(v) = 2$ und Kanten $e = \{u,v\}, f = \{v,w\} \in E$.
Die **Serienreduktion** kann auf v angewendet werden, wenn v nicht in K liegt. Aus dem Graphen G entsteht der neue Graph $\tilde{G}$ mit den zugehörigen Terminalknoten $\tilde{K} := K$. In $\tilde{G}$ sind die Kanten e und f durch eine neue Kante $g = \{u,w\}$ mit der Wahrscheinlichkeit $p_g = p_e \cdot p_f$ ersetzt und v wurde entfernt.
Wenn alle 3 Knoten Terminals sind, so kann eine **Grad-2-Reduktion** von v durchgeführt werden. Aus dem Graph G mit K bildet man einen neuen Graph G' ebenso durch Ersetzen von e und f durch eine neue Kante $g = \{u,w\}$ mit $p_g = \frac{p_e \cdot p_f}{(1-(1-p_e)\cdot(1-p_f))}$ und dem Entfernen v aus Knotenmenge. Die zugehörige Terminalmenge ist $K' := K \backslash \{v\}$.

Satz 2.26 (Grad-2-Reduktion)
Die Grad-2-Reduktion ist unter allen Voraussetzungen der Definition mit $\theta = p_e + p_f - p_e p_f = 1-(1-p_e)(1-p_f)$ eine zuverlässigkeitserhaltende Reduktion.

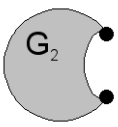
Beweis:
Die K-Zuverlässigkeit von G kann mithilfe des Prinzip der totalen Wahrscheinlichkeit zerlegt werden. Dabei unterscheidet man, welche Zustände die Kanten e, f annehmen können und wie sich dies auf den Graphen auswirkt.



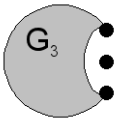
Sind beide Kanten intakt so können sie kontrahiert werden, wodurch alle drei Knoten identifiziert werden.
Wkt. dieses Ereignisses: $p_e p_f$



Wenn nur eine intakt ist, die andere jedoch ausfällt, besteht zwischen u und w keine Verbindung über den betrachteten Weg. Die existierende Kante kann ebenso kontrahiert werden, dabei wird v entweder mit u oder mit w verschmolzen.
Wkt. dieses Ereignisses: $p_e + p_f - 2p_e p_f$:



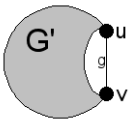
Fallen beide Kanten aus, so entsteht ein Graph mit isoliertem Knoten:
Wkt. dieses Ereignisses: $(1-p_e)(1-p_f)$



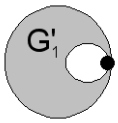
Da G_3 einen isolierten Terminalknoten enthält, ist K im Graphen G nicht mehr zusammenhängend und die K-Zuverlässigkeit ist gleich 0. Die anderen beiden resultierenden Graphen können weiterhin zusammenhängend sein, es ergibt sich:

$$Rel(G,K) = p_e p_f \cdot Rel(G_1,K_1) + (p_e + p_f - 2p_e p_f) \cdot Rel(G_2,K_2)$$

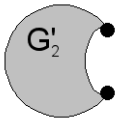
Analoge Überlegungen für den Ersatzgraphen G' ergeben:



Beide Knoten sind verbunden, wenn die Kante g intakt ist:
Wkt. dieses Ereignisses: p_g



Fällt die Kante aus, so sind die Knoten getrennt:
Wkt dieses Ereignisses: $1 - p_g$



Woraus sich die Gleichung ergibt:

$$Rel(G',K') = p_g \cdot Rel(G'_1,K'_1) + (1-p_g) \cdot Rel(G'_2,K'_2)$$

Da die Reduktion zuverlässigkeitserhaltend sein soll, gilt die folgende Beziehung:

$$\begin{aligned} &\theta Rel(G',K') = Rel(G,K) \\ &\theta \cdot (p_g \cdot Rel(G'_1,K'_1) + (1-p_g) \cdot Rel(G'_2,K'_2)) \\ =&p_e p_f \cdot Rel(G_1,K_1) + (p_e + p_f - 2p_e p_f) \cdot Rel(G_2,K_2) \end{aligned}$$

Die neu enstanden Graphen G'_1 und G'_2 sind gleich den Graphen G_1 und G_2, somit ist deren K-Zuverlässigkeit gleich. Man kann einen Koeffizientenvergleich durchführen und erhält:

$$\begin{aligned} \theta p_g &= p_e p_f \\ \theta(1-p_g) = \theta - \theta p_g &= p_e + p_f - 2p_e p_f \end{aligned}$$

$$\begin{aligned} \theta - p_e p_f &= p_e + p_f - 2p_e p_f \\ \theta &= p_e + p_f - p_e p_f \end{aligned}$$

$$p_g = \frac{p_e p_f}{\theta}$$

□

Satz 2.27
Die Serienreduktion ist eine zuverlässigkeitserhaltende Reduktion mit $\theta = 1$.

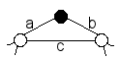
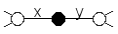
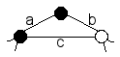
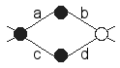
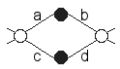
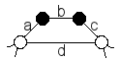
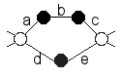
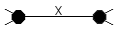
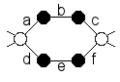
Beweis: Wenn der Knoten v kein Terminalknoten ist, so muss dieser nicht mit den anderen verbunden sein, damit der Graph zusammenhängt. Allerdings könnte über diesen Knoten die Verbindungsstrecke zwischen anderen Knoten verlaufen, was durch eine Kante zwischen seinen Nachbarknoten ebenso realisiert werden kann. Damit eine Verbindung über diesen Knoten existiert, müssen die zu ihm inzidenten Kanten funktionieren. Somit sollte die Wahrscheinlichkeit für die neu eingesetzte Kante $p_e p_f$ betragen. □

Eine weitere, komplexere Reduktion wurde von Wood und Satyanarayana eingeführt. Deren Grundlage bildet ein Satz von Wood, für dessen Verständnis zunächst folgende Begriffe eingeführt werden müssen:

Definition 2.28
Ein Graph G mit einer Terminalmenge K heißt **SP-reduzierbar**, wenn er mithilfe von Parallel- und Serienreduktionen auf eine Kante reduzierbar ist.

Oberflächlich betrachtet scheinen die Begriffe SP-Graph und SP-reduzierbar dieselbe Bedeutung zu haben. Jedoch ist festzustellen, dass ein SP-reduzierbarer Graph stets ein SP-Graph ist, die Umkehrung jedoch gilt nicht. Durch eine ungünstige Verteilung der Terminalknoten kann es passieren, dass ein SP-Graph nicht SP-reduzierbar ist, in solch einem Fall spricht man von **SP-komplexen Graphen**.
Wood gelang es zu zeigen, dass in einem SP-komplexen Graphen stets mindestens eines von 7 Polygonen auftaucht und entwickelte mit Satyanarayana die folgende Reduktion.

Definition 2.29
Gegeben sei ein Graph $G = (V, E)$ mit den Terminalknoten K und dem Polygon χ vom Typ i (s. Tab. 2.1). Mithilfe der **Polygon-Kette-Reduktion** entsteht aus G ein neuer G' mit der Terminalmenge K'. Dies geschieht durch das Ersetzen des Polygons durch die entsprechende Kette κ vom Typ i und der Umrechnung der Wahrscheinlichkeiten entsprechend der Tabelle 2.1.

Polygon	Kette	Umrechnung	Wahrscheinlichkeit
 Typ 1		$\alpha = q_a p_b q_c$ $\beta = p_a q_b q_c$ $\gamma = p_a p_b p_c (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c})$	$p_x = \frac{\gamma}{\alpha+\gamma}$ $p_y = \frac{\gamma}{\beta+\gamma}$ $\theta = \frac{(\alpha+\gamma)(\beta+\gamma)}{\gamma}$
 Typ 2		$\alpha = q_a p_b q_c$ $\beta = p_a q_b q_c$ $\gamma = p_a p_b p_c (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c})$	
 Typ 3		$\alpha = p_a q_b q_c p_d + q_a p_b p_c q_d + q_a p_b q_c p_d$ $\beta = p_a q_b p_c q_d$ $\gamma = p_a p_b p_c p_d (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d})$	
 Typ 4		$\alpha = q_a p_b q_c p_d$ $\beta = p_a q_b q_c p_d + q_a p_b p_c q_d$ $\gamma = p_a p_b q_c q_d$ $\delta = p_a p_b p_c p_d (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d})$	$p_x = \frac{\delta}{\alpha+\delta}$ $p_y = \frac{\delta}{\beta+\delta}$ $p_z = \frac{\delta}{\gamma+\delta}$ $\theta = \frac{(\alpha+\delta)(\beta+\delta)(\gamma+\delta)}{\delta^2}$
 Typ 5		$\alpha = q_a p_b p_c q_d$ $\beta = p_a q_b p_c q_d$ $\gamma = p_a p_b q_c q_d$ $\delta = p_a p_b p_c p_d (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d})$	
 Typ 6		$\alpha = q_a p_b p_c q_d p_e$ $\beta = p_a q_b p_c (p_d q_e + q_d p_e)$ $+ p_b (q_a p_c p_d q_e + p_a q_c q_d p_e)$ $\gamma = p_a p_b q_c p_d q_e$ $\delta = p_a p_b p_c p_d p_e$ $\cdot (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} + \frac{q_e}{p_e})$	Für Typ 5 mit $\lvert K\rvert = 2$ ist die Ersatzstruktur  mit Wkt. $p_x = \frac{(p_b + p_a q_a p_c p_d)}{\theta}$ und Faktor $\theta = p_b + p_a q_b p_c$
 Typ 7		$\alpha = q_a p_b p_c q_d p_e p_f$ $\beta = p_a q_b p_c (q_d p_e p_f + p_d q_e p_f p_d p_e q_f)$ $+ p_a p_b q_c p_f (p_d q_e + q_d p_e)$ $+ q_a p_b p_c p_d (q_e p_f + p_e q_f)$ $\gamma = p_a p_b q_c p_d p_e q_f$ $\delta = p_a p_b p_c p_d p_e p_f$ $\cdot (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} + \frac{q_e}{p_e} + \frac{q_f}{p_f})$	

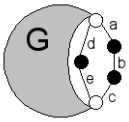
Tab. 2.1: Polygon-Kette-Reduktionen

Bem.: Für eine Kante $e \in E$ berechnet sich $q_e = 1 - p_e$

Satz 2.30 (Polygon-Kette-Reduktion)
Es sei $G = (V, E)$ mit der Terminalmenge K. Dann ist die Polygon-Kette-Reduktion für ein Polygon χ in G entsprechend des Typs von χ eine zuverlässigkeitserhaltende Reduktion mit θ laut Tabelle 2.1.

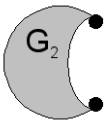
Beweis: Für jeden Typ der Polygon-Kette-Reduktion kann man einen konstruktiven Be-

weis anführen, stellvertretend wird Typ 6 vorgestellt. Für die anderen Typs lässt sich ein analoges Vorgehen entwickeln. Erneut ist die Grundidee das Prinzip der totalen Wahrscheinlichkeit.
Aus dem Originalgraph können 4 verschiedene neue Graphenstrukturen entstehen, die noch zusammenhängend sein können (Strukturen, in denen Terminalknoten vom Restgraphen getrennt sind, werden ignoriert, da deren K-Zuverlässigkeitswert 0 beträgt). Drei Varianten entstehen, wenn keine der beiden Ketten vollständig intakt ist. Je nach der Verteilung der intakten Kanten können entweder zu beiden oder nur zu einem der Endknoten Verbindungen existieren. In der vierten Struktur sind die Endknoten verbunden. Alle existieren Verbindungen werden dargestellt, in dem die zugehören Kanten kontrahiert werden.



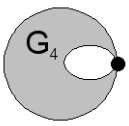
Dieser Restgraph entsteht, wenn die Kanten a und d ausfallen. Für den Zusammenhang ist dann wichtig, dass die restlichen Kanten intakt sind, so dass sie kontrahiert werden. Da der neue Superknoten auch für Terminalknoten steht, wird er in die Terminalmenge aufgenommen.
Wkt.: $\alpha = q_a p_b p_c q_d p_e$

Bei diesem Fall gibt es verschiedene Varianten. Damit keine Kette vollständig ist, kann entweder d oder e intakt sein. Wenn d ausfällt, muss a intakt sein, bei Ausfall von e muss c funktionieren, damit beide Außenknoten mit mindestens einem Terminal verbunden sind. Von den übrigen beiden Kanten der rechten Kette darf genau eine intakt sein, die andere muss ausfallen.
Wkt.: $\beta = (p_a q_b + q_a p_b) p_c p_d q_e + p_a (p_b q_c + q_b p_c) q_d p_e$



Analog zum ersten Fall sollten hier c und e ausfallen, die restlichen Kanten intakt sein.
Wkt.: $\gamma = p_a p_b q_c p_d q_e$

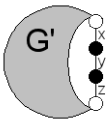
Bei diesem Fall muss mindestens eine Kette komplett funktionieren, und die Terminals der anderen sollten zumindest mit einem der Außenknoten verbunden sein. Dies kann auf verschiedene Weise geschehen.
Wkt.: $\delta = p_a p_b p_c (p_d q_e + q_d p_e + p_d p_e)) + (q_a p_b p_c + p_a q_b p_c + p_a p_b q_c) p_d p_e = p_a p_b p_c p_d p_e (1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} + \frac{q_e}{p_e})$



Man kann also schließen, dass für die K-Zuverlässigkeit des Graphen G gilt:

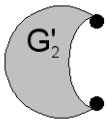
$$\begin{aligned} Rel(G,K) =& \alpha \cdot Rel(G_1,K_1) + \beta \cdot Rel(G_2,K_2) \\ &+ \gamma \cdot Rel(G_3,K_3) + \delta \cdot Rel(G_4,K_4) \end{aligned}$$

Eine ebensolche Zerlegung lässt sich für die Ersatzstruktur bilden:



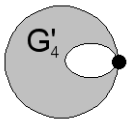
Es sollte keine Verbindung zum oberen Knoten existieren, somit muss x ausfallen, die restlichen Kanten intakt sein.
Wkt.: $(1-p_x)p_yp_z$

Für diesen Fall darf die mittlere Kante nicht funktionieren, dafür aber die beiden Äußeren.
Wkt.: $p_x(1-p_y)p_z$



Analog zu dem ersten Fall.
Wkt.: $p_xp_y(1-p_z)$

Die gesamte Kette muss funktionieren.
Wkt.: $p_xp_yp_z$



Es lässt sich also auch deren K-Zuverlässigkeit als Summe darstellen:

$$\begin{aligned} Rel(G',K') =& [(1-p_x)p_yp_z] \cdot Rel(G'_1,K'_1) + [p_x(1-p_y)p_z] \cdot Rel(G'_2,K'_2) \\ &+ [p_xp_y(1-p_z)] \cdot Rel(G'_3,K'_3) + [p_xp_yp_z] \cdot Rel(G'_4,K'_4) \end{aligned}$$

Die beiden Graphen G und G' unterscheiden sich lediglich in den aufgezeigten Bereichen. Betrachtet man jedoch die Ersatzstrukturen, so besitzen beide dieselben Fälle und mithilfe der Beziehung $Rel(G,K) = \theta Rel(G',K')$ kann man einen Koeffizientenvergleich durchführen. Es ergeben sich die folgenden vier Gleichungen:

$$\alpha = \theta(1-p_x)p_yp_z \quad (1)$$
$$\beta = \theta p_x(1-p_y)p_z \quad (2)$$
$$\gamma = \theta p_xp_y(1-p_z) \quad (3)$$
$$\delta = \theta p_xp_yp_z \quad (4)$$

Die einzusetzenden Werte für p_x, p_y und p_z lassen sie sich wie folgt daraus berechnen:

aus (1) und (4) folgt:
$$\frac{1-p_x}{p_x} = \frac{1}{p_x} - 1 = \frac{\alpha}{\delta}$$
$$\frac{1}{p_x} = \frac{\alpha+\delta}{\delta}$$
$$p_x = \frac{\delta}{\alpha+\delta}$$

aus (2) und (4) folgt:
$$\frac{1-p_y}{p_y} = \frac{1}{p_y} - 1 = \frac{\beta}{\delta}$$
$$\frac{1}{p_y} = \frac{\beta+\delta}{\delta}$$
$$p_y = \frac{\delta}{\beta+\delta}$$

aus (3) und (4) folgt:
$$\frac{1-p_z}{p_z} = \frac{1}{p_z} - 1 = \frac{\gamma}{\delta}$$
$$\frac{1}{p_z} = \frac{\gamma+\delta}{\delta}$$
$$p_z = \frac{\delta}{\gamma+\delta}$$

Zur Bestimmung von θ kann man an dieser Stelle jede der vier Gleichungen des Koeffizientenvergleichs nutzen, doch eignet sich Nummer (4) am Besten und es ergibt sich:

$$\theta = \frac{\delta}{p_x p_y p_z}$$
$$= \frac{\delta}{\frac{\delta}{\alpha+\delta}\frac{\delta}{\beta+\delta}\frac{\delta}{\gamma+\delta}}$$
$$= \frac{(\alpha+\delta)(\beta+\delta)(\gamma+\delta)}{\delta^2}$$

□

3 Algorithmen

Zum praktischen Nutzen der theoretischen Erkenntnisse ist eine algorithmische Erfassung unablässig. Die erstellten Algorithmen werden in diesem Kapitel vorgestellt.
Zur Optimierung der Laufzeit stellte es sich als sinnvoll heraus, innerhalb einiger Abläufe bereits andere vorzubereiten, z.B. durch Speicherung von Daten. An den entsprechenden Stellen sind in den Algorithmen daher Bemerkungen als Platzhalter eingefügt, die an späterer Stelle erklärt werden. Für die Untersuchungen des Aufwands gelte bei einem Graphen $G = (V,E)$, dass $|V| = n$, $|E| = m$ und K die Terminalmenge sei.

3.1 Voraussetzungen

Grundlage der Programmierung bildet die von Herrn Prof. Dr. Peter Tittmann (HS Mittweida) entwickelte Pythonklasse 'MultiGraph' zur Speicherung und Darstellung von Graphen mit parallelen Kanten. Innerhalb dieser existiert ein Zusammenhangstest für den Gesamtgraphen. Er beruht auf dem Prinzip der Tiefensuche, indem ausgehend von einem beliebigen Startknoten die Nachbarschaft zu durchsuchen ist, alle adjazenten Knoten gespeichert werden und dieses Vorgehen so lang wiederholt wird, bis keine Neuen mehr gefunden werden. Durch einen Vergleich mit der Knotenmenge wird ersichtlich, ob der Graph zusammenhängend ist oder nicht.

Algorithmus 1 Zusammenhangstest für die gesamte Knotenmenge V aus 'MultiGraph'

Eingabe: Graph G
Rückgabe: 1 für G zusammenhängend, sonst 0

Wähle einen Knotens u aus V
Initialisiere $X = \{u\}$, $Y = N(u)$, $k = 1$
while $Y \backslash X \neq \emptyset$ **do**
  Addiere zu k die Anzahl der Knoten in $Y \backslash X$
  Setze $Z = Y \backslash X$
  Erweitere X um Y
  Füge zu Y alle Nachbarn von Knoten aus Z hinzu
end while
if $k =$ Anzahl der Knoten im Graph **then**
  return 1
else
  return 0
end if

In Analogie zu diesem Vorgehen wurde ein Algorithmus zum Test des Zusammenhangs einer Knotenteilmenge entwickelt. Der ursprüngliche Test kann dafür nicht genutzt werden, da in einem insgesamt nicht zusammenhängenden Graphen ein Teil der Knoten

durchaus verbunden sein kann, wenn sie alle in einer Komponente vorkommen. Daher ergeben sich einige Veränderungen bezüglich der Umsetzung.
Unterschiede existieren in der Wahl des Anfangsknoten - dieser muss zwingend aus der Terminalmenge gewählt werden - und in der Endprüfung, da nicht alle Knoten in der gefunden Menge liegen müssen, sondern nur die Knoten der Terminalmenge K.

Algorithmus 2 Zusammenhangstest für Terminalmenge $K \subseteq V$

Eingabe: Graph G, Terminalmenge K
Rückgabe: 1 für K in G zusammenhängend, sonst 0

Wähle eines Knoten u aus K, setzen $X = \{u\}$
$Y = N(u)$
while $Y \backslash X \neq \emptyset$ **do**
  Setze $Z = Y \backslash X$
  Erweitere X um Y
  Füge zu Y alle Nachbarn von Knoten aus Z hinzu
end while
if $K \subseteq X$ **then**
  Bem. 1
  return 1
else
  return 0
end if

Bem. 1: An dieser Stelle kann auf eine besonders triviale Weise die Komponentenreduktion vorgenommen werden. In X ist die Komponente gespeichert, in der sich alle Knoten aus K befinden, somit sind die Knoten in $V \backslash X$ in anderen Komponenten und können gelöscht werden.

3.2 Rekursion

Zur Umsetzung der Dekompositionsformel als Rekursion muss man die Abbruchbedingungen beachten. Dabei würde man theoretisch jedes Mal erneut den Zusammenhang prüfen, was vor allem bei großen Graphen einen enormen Aufwand bedeutet. Mit der Überlegung, dass durch eine Kantenkontraktion der Zusammenhang niemals zerstört wird, folgt, dass immer in diesem Teil der Verzweigung die Prüfung entfallen kann. Umsetzung der Überlegung erfolgt durch eine Variable - hier i genannt -, die im Falle der Kontraktion den Wert 1 annimmt, für die Entfernung den Wert 0 aufweist. Mit Hilfe der Abfrage 'if $i = 1$ oderK in Graph zusammenhängend' kann die Eigenschaft der 'ODER'-Verknüpfung genutzt werden, dass sie den Wert 'WAHR' liefert, wenn eine der Aussagen wahr ist. Bei der Verarbeitung der Abfrage erfolgt eine Abarbeitung von links nach rechts, sodass der Wert 'WAHR' angenommen wird, sobald bei der Abfrage von links nach rechts eine Bedingung 'WAHR' liefert. Somit wird nur für $i = 0$ der Zusammenhang überprüft.

Während die Entfernung einer Kante nur eine Änderung in der Kantenmenge bewirkt, können durch die Kontraktion parallele Kanten entstehen, so dass durch eine zugehörige Parallelreduktion die Terminalmenge und die Wahrscheinlichkeitswerte verändert werden. Daher ist es sinnvoll, für diese Operation einen neuen Graphen zu erstellen und die Beeinflussten ebenfalls angepasst neu zu speichern.

Algorithmus 3 Rekursion für K-Zuverlässigkeit

Name: $K\_connect(G,K,p,i)$
Eingabe: Graph G, Terminals K, Wahrscheinlichkeitsvektor p, Variable i
Rückgabe: K-Zuverlässigkeit von K in G

if $i = 1$ oder K in Graph zusammenhängend **then**
  Setze $\theta = 1$
  Führe die gewünschten Reduktionen aus (dabei wird θ verändert)
  if $|K| = 1$ **then**
    return θ
  else
    Wähle eine Kante e aus E
    Bilde $H = G \backslash e$ und dazu passende Terminalmenge K' und Wkt.-vektor p'
    return $\theta \cdot (p_e \cdot K\_connect(H,K',p',1) + (1-p_e) \cdot K\_connect(G-e,K,p,0))$
  end if
else
  return 0
end if

Zunächst sollte man den Aufwand für diese Rekursion ohne zusätzliche Reduktionen betrachten. Eine Vorstellung für den Ablauf einer Rekursion ist immer ein Wurzelbaum. Wurzel dieses konkreten Baumes ist der ursprüngliche Graph $G = (V,E)$, aus diesem entspringen zwei Abzweigungen, jeweils eine für Kontraktion und für Entfernen einer Kante. Im nächsten Schritt schließen sich an diese beiden für eine nächste Kante ebenso zwei Fälle an, es entsteht ein binärer Baum. Im schlimmsten Falle muss man in allen Verzweigungen alle Kanten betrachten, somit hätte der Baum eine Tiefe von $m = |E|$ und die Anzahl von Blättern beliefe sich auf 2^m.
Es folgt, dass die Beantwortung der Frage nach dem Wert der K-Zuverlässigkeit für einen gegeben Graphen $G = (V,E)$ mit ebenso gegebenem Ausfallwahrscheinlichkeiten p_e $\forall e \in E$ einen exponentiellen Aufwand von $O(2^m)$ verursacht. Daraus ist außerdem offensichtlich, dass jede Entfernung von Knoten oder einzelnen Kanten zu einer Verkürzung der Laufzeit führt.

3.3 Reduktionen

Es ist von besonderem Interesse, eine Abschätzung zu geben, wie lang die einzelnen Algorithmen laufen, um eine Aussage über die praktische Nutzung treffen zu können. Die Effektivität der Reduktionen lässt sich nicht im Vorhinein angeben, da dieser Wert

stark von der Struktur des betrachteten Graphen abhängt. Folglich ist es nur möglich, eine Abschätzung über den Mehraufwand einer jeden Reduktion anzugeben und aufgrund der theoretischen Aspekte anzunehmen, dass dieser Mehraufwand gerechtfertigt ist. Gerechtfertigt bedeutet in diesem Zusammenhang, dass die Gesamtlaufzeit verringert wird.
Es ist offensichtlich, dass mit jeder ausgeführten Reduktion mindestens eine Kante entfällt. In der Analyse der Rekursion wurde ein binärer Baum beschrieben, um die exponentielle Laufzeit herzuleiten. Betrachtet man einen Teilbaum, dessen Wurzel ein Graph ist, in dem eine Reduktion durchgeführt werden kann. Ohne diese habe der Teilbaum eine Tiefe von l, trägt also zur Gesamtlaufzeit einen Teil von $O(2^l)$ bei. Mit der Reduktion hat er eine Tiefe von maximal $l-1$, somit ist sein Beitrag nur noch in $O(2^{l-1})$, was besonders bei großen l eine beachtliche Verkürzung darstellt. Aus dieser Überlegung heraus erhalten die Reduktionen ihre Berechtigung.

3.3.1 Grad-1-Reduktion

Die in Def. 2.20 vorgestellte Grad-1-Reduktion, kann wie folgt implementiert werden:

Algorithmus 4 Grad-1-Reduktion

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ
Rückgabe: durch Grad-1-Red. veränderte Werte von Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

for v in Knotenmenge von G **do**
  if $deg(v) = 1$ **then**
    if $v \in K$ **then**
      Suche die zu v adjazente Kante $e = \{u, v\}$
      $\theta = \theta \cdot p_e$
      Lösche v aus K und füge u hinzu
    end if
    Lösche v aus G
  end if
end for

In diesem Algorithmus wird jeder Knoten des Graphen auf seinen Grad hin untersucht. Die Berechnung des Grades für jeden Knoten erfolgt innerhalb der genutzten Klasse MultiGraph. Darin wird als eine Eigenschaft eines jeden Knoten die Menge der zu ihm inzidenter Knoten gespeichert, somit muss man für den Grad lediglich die Mächtigkeit der Menge angeben, was für den Aufwand zu vernachlässigen ist. Da zum Startzeitpunkt des Programms der Graph bereits erstellt ist, zählt der Aufwand für diese Speicherung nicht mit hinein.
Ist ein Knoten v vom Grad 1 gefunden, wird die zu ihm inzidente Kante betrachtet. Diese aufzufinden ist durch oben genannte Speicherung nicht sehr aufwendig, ebenso wenig

wie das Entfernen des Knoten. Da in dieser Reduktion nur ein Knoten und eine Kante entfernt wird, ist der Aufwand, egal wie der Graph ansonsten gestaltet ist, konstant. Zusätzlich ist zu testen, ob sich der Knoten v in der Terminalmenge befindet. Falls dies gilt, so wird der Knoten auch daraus entfernt und sein Nachbar stattdessen aufgenommen, außerdem gibt es eine Veränderung des Reduktionsfaktors. Der Aufwand für die Entscheidung, ob sich der Knoten in der Menge befindet, liegt in $O(|K|)$, die restlichen Operation sind konstant.
Die Untersuchung, ob ein Knoten den Grad 1 besitzt, muss für jeden Knoten durchgeführt werden. Es ergibt sich also ein Gesamtaufwand von $O(n \cdot |K|)$.

3.3.2 Parallelreduktion

Im Gegensatz zur Grad-1-Reduktion steht die Parallelreduktion in Abhängigkeit zur Kantenanzahl m. Diese kann gegenüber n beliebig groß werden, wenn parallele Kanten erlaubt sind (für zusammenhängenden Graphen brauch es mind. $n-1$ Kanten). Triviale Algorithmen liefern eine Laufzeit in $O(m^2)$ bzw. $O(n^2 m)$, diese Werte sind jedoch nicht optimal.
Die hier vorgestellte Variante hat dafür einen erhöhten Speicheraufwand, der es allerdings ermöglicht, in $O(m)$ zu bleiben. Der resultierende Aufwand für die Parallelreduktion ist daher $O(2m) \in O(m)$.

Algorithmus 5 Parallelreduktion für einen Graphen

Eingabe: Graph G, Wkt.-vektor p
Rückgabe: durch Parallelreduktion veränderte Werte für Graph G, Wkt.-vektor p

Zuordnungsvorschrift q - $q(u,v)$ ist Wkt., dass alle Kanten zwischen Knoten u und v ausfallen
Menge H - Speicherung aller Knotenpaare, zwischen denen Kanten existieren
Zuordnungsvorschrift Z - $Z(u,v)$ sind die Nummern von Kanten zwischen Knoten u und v
for e in Kantenmenge von G **do**
  Sei $e = (u,v)$
  Füge zu $Z(u,v)$ Kante e hinzu
  Setze $q(u,v) = q(u,v) \cdot (1 - p_e)$
  Füge (u,v) zur Menge H hinzu
end for
for (u,v) in H **do**
  Lösche alle Kanten bis auf eine (e genannt), die sich in $Z(u,v)$ befinden, aus G
  Setze $p_e = 1 - q(u,v)$
end for

In der ersten FOR-Schleife über die Kanten wird eine Zuordnung erstellt, die jedem Knotenpaar alle dazwischen verlaufenden Kanten zuordnet. Außerdem erfolgt die Berechnung der Wahrscheinlichkeit, dass alle Kante zwischen zwei Knoten ausfallen und die Speicherung der Knotenpaare, zwischen denen Kanten existieren. All diese Opera-

tionen sind nicht von der Problemgröße abhängig, somit hat diese erste Schleife einen Gesamtaufwand von $O(m)$.
Die zweite FOR-Schleife läuft zwar nur über alle Knotenpaare, doch innerhalb werden, falls wirklich parallele Kanten auftreten, alle dazwischenliegen Kanten betrachtet. Zusammengenommen wird in dieser 2. Schleife also ebenso die gesamte Kantenmenge betrachtet. Parallele Kante werden gelöscht und die Wahrscheinlichkeiten überschrieben. Durch die in der ersten Schleife berechnete Wahrscheinlichkeit erfolgt dies auf eine besonders triviale Weise. Somit steht diese Schleife ebenfalls in $O(m)$.
Am Anfang der Berechnung muss man natürlicherweise den Gesamtgraph hinsichtlich paralleler Kanten untersuchen, im weiteren Verlauf kann man jedoch genau bestimmen, zwischen welchen Knotenpaaren gesucht werden muss. Somit erfolgt eine Optimierung bezüglich dieses ersten Algorithmus, wenn dieser Gedanke umgesetzt wird. Bei der Kontraktion innerhalb der Rekursion sowie bei Grad-2- und Serien-Reduktionen können neue parallele Kanten entstehen, maximal ein Paar pro Knotenpaar. Im Falle der Anwendung nach der Kontraktion ist zu beachten, dass dies mehrere Nachbarn des neu entstandenen Knoten betreffen kann. Somit sollte, statt einer Gesamtuntersuchung in jedem Schritt, an den erwähnten Stellen gezielt untersucht werden, ob parallele Kanten entstanden sind, wozu der folgende Algorithmus dient:

Algorithmus 6 Parallelreduktion zwischen zwei bestimmten Knoten

Eingabe: Graph G, Knoten u, v, Wkt.-vektor p
Rückgabe: durch Parallelreduktion veränderte Werte für Graph G, Wkt.-vektor p

if es existieren zwei Kanten e, f zwischen u und v **then**
  Lösche diese Kanten aus G
  Füge neue Kante g zwischen u und v ein
  Setze $p_g = 1 - (1 - p_e) \cdot (1 - p_f)$
end if

Auch hier ergibt sich der Schwerpunkt der Laufzeit durch das Auffinden der Kanten, jedoch ist dies weniger aufwendig. Das Suchen der Kanten kann umgesetzt werden, in dem die inzidente Kantenmenge des einen Knoten durchsucht wird, ob der andere Knoten in zwei Kanten aus dieser Menge vorkommt. Liegt ein vollständiger Graph vor, so ist die Anzahl dieser Kanten allerdings $n - 1 \in O(n)$, was somit der Laufzeit im worst case entspricht, da alle weiteren Operation nicht in Abhängigkeit zur Problemgröße stehen.

3.3.3 Grad-2- und Serienreduktion

Ein Anwendungsfall der gerade vorgestellten Parallelreduktion zwischen zwei konkreten Knoten sind die Grad-2- und die Serienreduktion. Diese sollten, genauso wie man sie theoretisch zusammen einführen kann, gemeinsam durchgeführt werden, um die Laufzeit zu optimieren.

Algorithmus 7 Grad-2-/Serienreduktion

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ
Rückgabe: durch Grad-2-/Serienred. veränderte Werte von Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

for v in der Knotenmenge von G **do**
  if $deg(v) = 2$ **then**
    if v in K **then**
      if $N(v)$ in K **then**
        Suche und speichere die zu v inzidenten Kanten e, f
        Suche und speichere die zu v adjazenten Knoten u, w
        Setze $\theta = \theta \cdot (1 - (1 - p_e) \cdot (1 - p_f))$
        Lösche v aus G (dadurch automatisch auch Kanten e und f) und K
        Füge eine neue Kante $g = (u, w)$ in G ein
        Setze $p_g = (p_e \cdot p_f)/(1 - (1 - p_e) \cdot (1 - p_f))$
        Führe Parallelreduktion zwischen u und w durch
      else
        Bem. 2
      end if
    else
      Suche und speichere die zu v inzidenten Kanten e, f
      Suche und speichere die zu v adjazenten Knoten u, w
      Lösche v aus G (dadurch automatisch auch Kanten e und f)
      Füge eine neue Kante $g = (u, w)$ in G ein
      Setze $p_g = (p_e \cdot p_f)$
      Führe Parallelreduktion zwischen u und w durch
    end if
  end if
end for

Bem. 2: Knoten, die diesen Pfad aufrufen, können in einer Kette liegen und somit für die Polygon-Kette-Reduktion in Frage kommen. Werden sie gespeichert, entfällt eine erneute Suche, was den Aufwand reduziert - die Menge, in der gespeichert wird, sei PK.
Die grundlegende Vorgehensweise für diese Reduktionen stimmt mit der der Grad-1-Reduktionen überein. Besonderheit dieses Algorithmus ist die Verknüpfung zweier Reduktionen und die Vorbereitung einer Dritten. Durch diese Zusammenfassung wird die Laufzeit verringert, denn die eigentlichen Operationen der Reduktionen haben einen von der Größe des Problems unabhängigen Aufwand, während das Auswählen der Re-

duktionen in Abhängigkeit steht.
Zu Beginn das Algorithmus wird bestimmt, ob ein konkreter Knoten den Grad 2 besitzt. Danach erfolgt eine Untersuchung der Verteilung von Terminal- und Nicht-Terminalknoten um den Knoten. Zur Erinnerung, eine Grad-2-Red. ist möglich, wenn ein Terminalknoten vom Grad zwei adjazent zu zwei weiteren Terminalknoten ist, eine Serienreduktionen kann für alle Knoten vom Grad zwei, die keine Terminalknoten sind, angewendet werden und für die Polygon-Ketten-Red. benötigt es Terminalknoten vom Grad zwei, die zu maximal einem Terminal adjazent sind.
Nach dem Auffinden eines Knoten vom Grad zwei findet daher die Abfrage statt, ob er in der Terminalmenge enthalten ist, wie aus der vorhergehenden Analyse bekannt steht der Aufwand in Abhängigkeit zur Größe des Problems, also $O(|K|)$. Falls diese Abfrage positiv ausfällt, muss die Nachbarschaft des Knoten auf Zugehörigkeit zu den Terminals untersucht werden, es kommt erneut ein Aufwand von $O(|K|)$ hinzu.
Innerhalb der Reduktionen müssen die inzidenten Kanten und adjanzenten Knoten gefunden, Berechnungen durchgeführt, der betreffende Knoten entfernt und eine neue Kante eingefügt werden. All diese Operationen sind jedoch unabhängig von der Problemgröße, da der Grad des Knoten gleich zwei ist. Allerdings muss in beiden Fällen untersucht werden, ob eine Parallelreduktion möglich ist, welche wie bereits bekannt in Abhängigkeit von der Problemgröße steht und einen Mehraufwand von $O(n)$ bringt.
Es ergibt sich daher mit der FOR-Schleife eine Gesamtaufwand von:
$O(n) \cdot (O(|K|) + O(n)) \in O(n^2)$.

3.3.4 Polygon-Kette-Reduktion

Aus der vorangegangenen Algorithmus mitgenommen sei die Menge PK, die als Vorbereitung für die Polygon-Kette-Reduktionen (im Folgenden auch kurz mit PKR bezeichnet) gilt. Der folgende Ablaufplan zeigt auf, wie man möglichst effektiv Ketten bzw. Polygone herausfiltern und den Reduktionstyp bestimmen kann. Die Reduktionen selbst sind eigenständig aufgeführt in den Algorithmen 9 bis 14. Aus der Theorie ist bekannt, dass jede Reduktion spezifische Koeffizienten $\alpha, \beta, \gamma, \delta$ aufweist, die Berechnung der neuen Wahrscheinlichkeitswerte und des Reduktionsfaktors jedoch durch die gleiche Berechnungsvorschrift hervorgehen (bzw. es gibt 2 verschiedene Vorschriften). Daher ist es sinnvoll, diese Berechnungen auszulagern und die entsprechende Funktionen mit den spezifischen Koeffizienten aufzurufen. Die Algorithmen hierzu sind 15 bzw. 16.

Algorithmus 8 Polygon-Kette-Reduktion

Eingabe: Graph G, Terminals K, Menge PK, Wkt.-vektor p, Reduktionsfaktor θ
Rückgabe: durch Polygon-Kette-Red. veränderte Werte von Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ, Menge der wieder zu untersuchenden Knoten U

while $PK \neq \emptyset$ **do**
  Wähle Knoten v aus PK
  Suche und speichere die Kette κ, in der v liegt
  Entferne alle inneren Knoten von κ aus PK
end while
Initialisiere $U = \emptyset$
for Knotenpaare k, l zwischen denen Ketten gefunden wurden **do**
  while es gibt mind. zwei echte Ketten κ_1, κ_2 **do**
    if Grad von k und Grad von l größer 2 **then**
      m sei Anzahl der Knoten in κ_1, n die Anzahl in κ_2
      if $min(m,n) = 3$ **then**
        if $max(m,n) = 3$ **then**
          if k oder l in K **then**
            Polygon-Kette-Reduktion nach Typ 3
          else
            Polygon-Kette-Reduktion nach Typ 4
          end if
        else
          Polygon-Kette-Reduktion nach Typ 6
        end if
      else
        Polygon-Kette-Reduktion nach Typ 7
      end if
      Entferne κ_1 und κ_2 aus dem Speicher der zu untersuchenden Ketten
      Füge die neu erzeugte Kette dem Speicher der zu untersuchenden Ketten hinzu
    end if
  end while
  if Grad von k und Grad von l größer 2 **then**
    if es gibt eine echte Kette κ und eine Kante **then**
      n sei Anzahl der Knoten in κ_1
      if $n = 3$ **then**
        Polygon-Kette-Reduktion nach Typ 1 oder Typ 2
      else
        Polygon-Kette-Reduktion nach Typ 5
      end if
    end if
  end if
end for

Erster Schritt der Reduktion ist es, aus den in Alg. 7 gespeicherten Knoten die existierenden Ketten zu erzeugen. Die Menge PK habe $pk \leq n$ Knoten. Das Auffinden der Kette zu einem konkreten Knoten $v \in PK$ verlangt verschiedene Fallunterscheidungen. Es muss grundsätzlich geprüft werden, ob eine 3-er oder eine 4-er Kette vorliegt, d.h. ob der Grad eines Nachbarn von v ebenso 2 ist. In diesem Fall könnte es passieren, dass gar keine Kette vorliegt. Dies geschieht, wenn Anfangs- und Endknoten der Kette gleich wären, somit wäre es ein Kreis und keine Kette. Dennoch können diese Unterscheidungen in konstanter Zeit getroffen werden, somit ergibt sich ein Gesamtaufwand für die erste WHILE-Schleife von $O(pk)$. Zu beachten ist, dass die gefundenen Ketten alle echt sind.
Im zweiten Schritt werden alle Knotenpaare betrachtet, zwischen denen im ersten Schritt echte Ketten gefunden wurden. Die Untersuchung auf mögliche PKR unterteilt sich wiederum in zwei Schritte.
Der erste Teilschritt beinhaltet die Untersuchung, ob zwei oder mehr echte Ketten zwischen dem Paar liegen. Wenn dies der Fall ist, sind Polygon-Kette-Reduktionen durchführbar. Es werden zwei Ketten ausgewählt, für das daraus resultierende Polygon der Typ bestimmt und der Graph entsprechend reduziert. Dadurch verringert sich die Zahl der Kette zwischen dem Knotenpaar um 1, da aus zwei Ketten eine wird. Im schlimmsten Falle gibt es zu Beginn $n-2$ zu untersuchende Ketten, wenn alle Ketten Länge 3 haben und zwischen einem Knotenpaar verlaufen. Daraus folgt, dass es bis zu $n-3 \in O(n)$ Iterationsdurchläufe in diesem ersten Teilschritt geben kann.
Für zwei Ketten ist zu dem noch herauszufinden, welcher Typ - es könnte 3, 4, 6 oder 7 sein - vorliegt.
Um herauszufinden, welcher genau anzuwenden ist, erfolgt ein Vergleich der Längen. Der Aufwand hierfür ist konstant, allerdings zerstört der Typ 3 die Konstanz der Gesamtuntersuchung. In Typ 3 ist einer der Endknoten ein Terminal, folglich muss geprüft werden, ob einer der Knoten diese Eigenschaft besitzt. Daher kann es für jedes zu untersuchende Knotenpaar zu einer Abfrage kommen, die eine Gesamtaufwendung von $O(|K|)$ liefert. Die zusätzliche Bedingung, dass die beiden Ketten aus dem Speicher gelöscht und die neue eingefügt wird, ist unabhängig von der Problemgröße.
Nach dem 1. Teilschritt befindet sich für das betrachtete Knotenpaar eine Kette. Wenn des weiteren eine direkte Kante zwischen den beiden existiert, kann eine weitere Reduktion ausgeführt werden. Das Auffinden dieser Kante steht in $O(n)$, die restlichen Operationen sind konstant.
Somit hat der insgesamte Ablauf ohne die konkreten Reduktionen einen Aufwand von $O(n^2)$.

Bei der Speicherung der Ketten kommt es zu einer Festlegung der Reihenfolge, d.h. einer der Endknoten ist der Erste/Startknoten und einer der Letzte/Zielknoten. Daher ist es sinnvoll, bei der Speicherung eine Normierung festzulegen, damit eine Verbesserung der Erkennung übereinstimmender Endknoten erfolgen kann. Als günstig hat sich erwiesen, die Ketten so zu ordnen, dass der Startknoten kleiner als der Zielknoten ist. Für die folgenden Betrachtung sei diese Sortierung genutzt und für die Knoten sei $k < l$, d.h. k ist Start und l ist Ziel.
Des weiteren ist der letzte Schritt im Algorithmus das Berechnen der neuen Wahrscheinlichkeiten für die Ersatzstrukturen. Es ist angegeben, wie die Ersatzstruktur aussieht, d.h. (x,y) steht für die Kette mit einem inneren Knoten und den Kanten x und y. In den Berechnungsklassen werden die Kanten in der Reihenfolge abgearbeitet, wie sie aufgelistet sind.

Algorithmus 9 Polygon-Kette-Reduktion nach Typ 1 oder 2

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ, Paar k,l, Kette κ, Kante e
Rückgabe: durch PKR Typ 1/2 veränderte Werte für Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

Finde die Kanten f_1 und f_2 der Kette, f_1 sei die zu k adjazente Kante
Berechne $\alpha = (1-p(f_1)) \cdot p(f_2) \cdot (1-p(e))$
Berechne $\beta = p(f_1) \cdot (1-p(f_2)) \cdot (1-p(e))$
Berechne $\gamma = p(f_1) \cdot p(f_2) \cdot p(e) \cdot (1+(1-p(f_1))/p(f_1)+(1-p(f_2))/p(f_2)+(1-p(e))/p(e))$
Lösche e aus G
if $l \in K$ **then**
  Wkt. der Kette (f_1, f_2) und θ neu nach Berechnungsklasse I(β, α, γ)
else
  Wkt. der Kette (f_1, f_2) und θ neu nach Berechnungsklasse I(α, β, γ)
end if

Besonderheit bei dieser Verknüpfung von Typ 1 und Typ 2 ist, dass die Berechnungen und Veränderungen identisch sind. Achten muss man dabei darauf, das bei Typ 2 für die Berechnung wichtig ist, auf welcher Seite sich der Terminalknoten befindet. Innerhalb des vorgestellten Algorithmus werden die Koeffizienten so berechnet, dass k ein Terminalknoten sein könnte, aber l dürfte es nicht sein. Betrachtet man aber den Fall, dass l ein Terminal ist, so tauschen gerade α und β die Rollen. Da in jedem Fall einmal die Terminalmenge durchsucht werden muss, ergibt sich ein zusätzlicher Aufwand von $O(|K|)$.

Algorithmus 10 Polygon-Kette-Reduktion nach Typ 3

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ, Paar k,l, Ketten κ_1, κ_2
Rückgabe: durch PKR Typ 3 veränderte Werte für Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

if $k \in K$ **then**
  Finde die Kanten e_1, e_2 von κ_1, e_1 sei zu k adjazent
  Finde die Kanten f_1, f_2 von κ_2, f_1 sei zu k adjazent
else
  Finde die Kanten e_1, e_2 von κ_1, e_1 sei zu l adjazent
  Finde die Kanten f_1, f_2 von κ_2, f_1 sei zu l adjazent
end if
Berechne $\alpha = p(e_1)\cdot(1-p(e_2))\cdot(1-p(f_1))\cdot p(f_2)+(1-p(e_1))\cdot p(e_2)\cdot p(f_1)\cdot(1-p(f_2))+(1-p(e_1))\cdot p(e_2)\cdot(1-p(f_1))\cdot p(f_2)$
Berechne $\beta = p(e_1)\cdot(1-p(e_2))\cdot p(f_1)\cdot(1-p(f_2))$
Berechne $\gamma = p(e_1)\cdot p(e_2)\cdot p(f_1)\cdot p(f_2)\cdot(1+(1-p(e_1))/p(e_1)+(1-p(e_2))/p(e_2)+(1-p(f_1)])/p(f_1)+(1-p(f_2))/p(f_2))$
Lösche den inneren Knoten von κ_2 aus G und K
Wkt. der Kette (e_1, e_2) und θ neu nach Berechnungsklasse I (α, β, γ)

Die Abfrage bei Typ 3 nach Terminalzugehörigkeit kann verschieden gestaltet werden, so dass sie immer in $O(|K|)$ bleibt. Über die hier gewählte Weise müssen die Berechnungen nicht getrennt verlaufen, da die Rollen der Kanten getauscht werden.

Algorithmus 11 Polygon-Kette-Reduktion nach Typ 4

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ, Paar k,l, Ketten κ_1, κ_2
Rückgabe: durch PKR Typ 4 veränderte Werte für Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

Finde die Kanten e_1, e_2 von κ_1, e_1 sei zu k adjazent
Finde die Kanten f_1, f_2 von κ_2, f_1 sei zu k adjazent
Berechne $\alpha = (1-p(e_1)) \cdot p(e_2) \cdot (1-p(f_1)) \cdot p(f_2)$
Berechne $\beta = p(e_1) \cdot (1-p(e_2)) \cdot (1-p(f_1)) \cdot p(f_2) + (1-p(e_1)) \cdot p(e_2) \cdot p(f_1) \cdot (1-p(f_2))$
Berechne $\gamma = p(e_1) \cdot p(e_2) \cdot (1-p(f_1)) \cdot (1-p(f_2))$
Berechne $\delta = p(e_1) \cdot p(e_2) \cdot p(f_1) \cdot p(f_2) \cdot (1 + (1-p(e_1))/p(e_1) + (1-p(e_2))/p(e_2) + (1-p(f_1))/p(f_1) + (1-p(f_2))/p(f_2))$
Lösche e_2 und f_1 aus G
Füge Kante h zwischen den inneren Knoten der beiden Ketten ein
Wkt. der Kette (e_1, h, f_2) und θ neu nach Berechnungsklasse II (α, β, γ, δ)

In Typ 4 sind beide Endknoten des Polygons Nichtterminals. Somit muss nicht wie in den Typen zuvor eine Abfrage laufen, die eine problemgrößenabhängige Laufzeit erzeugt. Da die Knoten innerhalb einer Kette den Grad 2 aufweisen, ist es möglich, in konstanter Zeit die Kanten zu finden, sowie einfach Berechnungen stets instantsunabhängig sind.

Algorithmus 12 Polygon-Kette-Reduktion nach Typ 5

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ, Paar k,l, Kette κ, Kante e
Rückgabe: durch PKR Typ 5 veränderte Werte für Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

Finde die Kanten f_1, f_2 und f_3 der Kette, f_1 sei die zu k adjazente Kante, f_3 die zu l
if $|K| > 2$ **then**
  Berechne $\alpha = (1-p(f_1)) \cdot p(f_2) \cdot p(f_3) \cdot (1-p(e))$
  Berechne $\beta = p(f_1) \cdot (1-p(f_2)) \cdot p(f_3) \cdot (1-p(e))$
  Berechne $\gamma = p(f_1) \cdot p(f_2) \cdot (1-p(f_3)) \cdot (1-p(e))$
  Berechne $\delta = p(f_1) \cdot p(f_2) \cdot p(f_3) \cdot p(e) \cdot (1 + (1-p(f_1))/p(f_1) + (1-p(f_2))/p(f_2) + \cdot(1-p(f_3))/p(f_3) + (1-p(e))/p(e))$
  Lösche e aus G
  Wkt. der Kette (f_1, f_2, f_3) und θ neu nach Berechnungsklasse II(α, β, γ, δ)
else
  $\tau = (p(f_2) + p(f_1) \cdot (1-p(f_2)) \cdot p(f_3)$
  $p(e) = (p(f_2) + p(f_1) * (1-p(f_2)) * p(f_e) * p(e))/\tau$
  $\theta = \theta * \tau$
  Es seien a_1 und a_2 die inneren Knoten von κ
  Lösche a_1 und a_2 aus G und K
  Füge k und l zu K hinzu
end if

Eine Besonderheit weist Typ 5 auf, da für $|K| = 2$ die eigentliche Reduktion nicht funktioniert. Grund dafür ist die Tatsache, dass die beiden inneren Knoten der echten kette die beiden einzigen Terminalknoten sind. Da dieser Fall so aus Reihe fällt, ist die zugehörige Berechnung so speziell, dass sie keiner Klasse zugeordnet werden kann und somit innerhalb des Typs vollzogen wird.

Algorithmus 13 Polygon-Kette-Reduktion nach Typ 6

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ, Paar k,l, Ketten κ_1, κ_2
Rückgabe: durch PKR Typ 6 veränderte Werte für Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

Es sei κ_1 die längere Kette (Fallunterscheidung)
Finde die Kanten e_1, e_2 und e_3 von κ_1, e_1 sei die zu k adjazente Kante, e_3 die zu l
Finde die Kanten f_1 und f_2 von κ_2, f_1 sei die zu k adjazente Kante
Berechne $\alpha = (1-p(e_1)) \cdot p(e_2) \cdot p(e_3) \cdot (1-p(f_1)) \cdot p(f_2)$
Berechne $\beta = p(e_1) \cdot (1-p(e_2)) \cdot p(e_3) \cdot (p(f_1) \cdot (1-p(f_2)) + (1-p(f_1)) \cdot p(f_2)) + p(e_2) \cdot ((1-p(e_1)) \cdot p(e_3) \cdot p(f_1) \cdot (1-p(f_2)) + p(e_1) \cdot (1-p(e_3)) \cdot (1-p(f_1)) \cdot p(f_2))$
Berechne $\gamma = p(e_1) \cdot p(e_2) \cdot (1-p(e_3)) \cdot p(f_1) \cdot (1-p(f_2))$
Berechne $\delta = p(e_1) \cdot p(e_2) \cdot p(e_3) * p(f_1) \cdot p(f_2) \cdot (1 + (1-p(e_1))/p(e_1) + (1-p(e_2))/p(e_2) + (1-p(e_3))/p(e_3) + (1-p(f_1))/p(f_1) + (1-p(f_2))/p(f_2))$
Lösche den inneren Knoten von κ_2 aus G und K
Wkt. der Kette (e_1, e_2, e_3) und θ neu nach Berechnungsklasse II (α, β, γ, δ)

Bei Typ 6 wird die kürzere Kette gelöscht. Man muss daher bei der Umsetzung des Algorithmus darauf achten, eine Fallunterscheidung durchzuführen, welche Kette kürzer und welche länger ist. Es ist dabei unnötig, die Berechnung anzupassen, durch einen einfachen Umtausch entsteht eine elegantere Lösung. In der zur Arbeit gehörenden Implementierung fand der Umtausch durch die Speicherung der Kanten statt, das Löschen des Knoten muss dann während der Fallunterscheidung mit geschehen.

Algorithmus 14 Polygon-Kette-Reduktion nach Typ 7

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ, Paar k,l, Ketten κ_1, κ_2
Rückgabe: durch PKR Typ 7 veränderte Werte für Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

Finde die Kanten e_1, e_2 und e_3 von κ_1, e_1 sei die zu k adjazente Kante, e_3 die zu l
Finde die Kanten f_1, f_2 und f_3 von κ_2, f_1 sei die zu k adjazente Kante,
Berechne $\alpha = (1-p(e_1)) \cdot p(e_2) \cdot p(e_3) \cdot (1-p(f_1)) \cdot p(f_2) \cdot p(f_3)$

Berechne $\beta = p(e_1) \cdot (1-p(e_2)) \cdot p(e_3) \cdot ((1-p(f_1)) \cdot p(f_2) \cdot p(f_3) + p(f_1) \cdot (1-p(f_2)) \cdot p(f_3) + p(f_1) \cdot p(f_2) \cdot (1-p(f_3))) + p(e_1) \cdot p(e_2) \cdot (1-p(e_3)) \cdot p(f_3) \cdot (p(f_1) \cdot (1-p(f_2)) + (1-p(f_1)) \cdot p(f_2)) + (1-p(e_1)) \cdot p(e_2) \cdot p(e_3) \cdot p(f_1) \cdot ((1-p(f_2)) * p(f_3) + (f_2) \cdot (1-p(f_3)))$

Berechne $\gamma = p(e_1) \cdot p(e_2) \cdot (1-p(e_3)) \cdot p(f_1) \cdot p(f_2) \cdot (1-p(f_3))$

Berechne $\delta = p(e_1) \cdot p(e_2) \cdot p(e_3) \cdot p(f_1) \cdot p(f_2) \cdot p(f_3) \cdot (1 + (1-p(e_1))/p(e_1) + (1-p(e_2))/p(e_2) + (1-p(e_3))/p(e_3) + (1-p(f_1))/p(f_1) + (1-p(f_2))/p(f_2) + (1-p(f_3))/p(f_3))$
Lösche die inneren Knoten von κ_2 aus G und K
Wkt. der Kette (e_1, e_2, e_3) und θ neu nach Berechnungsklasse II (α, β, γ, δ)

Vergleicht man die Reduktionen miteinander, so ist festzustellen, dass die PKR höheren Typs, die eine größere Struktur verlangen, in ihrer Umsetzung weniger komplex sind. Grund dafür ist, dass keine Unterscheidung hinsichtlich Terminals als Außenknoten durchgeführt werden muss. Was bedeutet, es macht keinen Unterschied, ob man von Start zu Ziel oder umgedreht berechnet, während es in Fall 2 und 3 durchaus eine Rolle spielt.

Algorithmus 15 Berechnungsklasse I

Eingabe: Kette (a,b), α, β, γ, θ, Wkt.-vektor p
Rückgabe: der Klasse entsprechend veränderte Werte von $p(a), p(b)$ und θ

$p(a) = \gamma/(\alpha+\gamma)$
$p(b) = \gamma/(\beta+\gamma)$
$\theta = \theta \cdot (\alpha+\gamma) \cdot (\beta+\gamma)/\gamma$

Algorithmus 16 Berechnungsklasse II

Eingabe: Kette (a,b,c), α, β, γ, δ, θ, Wkt.-vektor p
Rückgabe: der Klasse entsprechend veränderte Werte von $p(a), p(b), p(c)$ und θ

$p(a) = \delta/(\alpha+\delta)$
$p(b) = \delta/(\beta+\delta)$
$p(c) = \delta/(\gamma+\delta)$
$\theta = \theta \cdot (\alpha+\delta) \cdot (\beta+\delta) \cdot (\gamma+\delta)/\gamma^2$

3.3.5 optimierter Durchführung der Reduktionen

Ein Beispiel zeigt, dass die Hintereinanderdurchführung der einzelnen Reduktion keine optimale Lösung darstellt. Dabei werden zwei benachbarte Knoten v,u mit der Eigenschaft, das der Grad von v drei und der Grad von u eins beträgt, betrachtet. Wenn zuerst v, danach u in der FOR-Schleife erreicht wird, gäbe es keine Möglichkeit, danach noch einmal v zu überprüfen, ob für diesen Knoten eine Reduktion möglich wäre. Somit müssen zwei Veränderung im Vorgehen stattfinden. Zum einen dürfen die Grad-1-, die Grad-2- und die Serienreduktion nicht getrennt voneinander durchgeführt werden. Stattdessen sollte für jeden noch zu untersuchenden Knoten geprüft werden, ob eine der drei anwendbar ist. Zum anderen kann mit einer FOR-Schleife nur eine feste, nicht veränderbare Menge durchsucht werden. Wie an diesem einfachen Beispiel gesehen, ist es aber unumgänglich, Knoten wieder in die zu untersuchende Menge aufzunehmen. Lösung bietet eine Suche mithilfe einer WHILE-Schleife, wie im folgenden Ablaufplan gezeigt:

Algorithmus 17 Durchführung

Eingabe: Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ
Rückgabe: durch Reduktionen veränderte Werte von Graph G, Terminals K, Wkt.-vektor p, Reduktionsfaktor θ

Initialisiere der Menge $U = V$ zur Speicherung der noch zu untersuchenden Knoten
while $U \neq \emptyset$ **do**
  while $U \neq \emptyset$ **do**
    Wähle v aus U
    if $deg(v) = 1$ **then**
      es sei u der Nachbarknoten von v
      Setze $U = U \cup \{u\} \cup \{w | w \in N(u), w \in PK\}$, $PK = PK \setminus \{u\}$
      Grad-1-Reduktion (ohne FOR-Schleife)
    end if
    if $deg(v) = 2$ **then**
      Grad-2-/Serienreduktion (ohne FOR-Schleife)
      Falls eine dieser durchgeführt, $U = U \cup N(v)$, $PK = PK\ \{v\}$
    end if
    Entferne v aus U
  end while
  Polygon-Kette-Reduktion (mit Veränderung von U)
  Wenn der Grad eines Endknoten eines Polygons auf 2 sinkt, diese zu U hinzufügen
end while

Es erscheint überflüssig, zweimal dieselbe Abfrage zu durchlaufen, doch der Sinn folgt aus den Polygon-Kette-Reduktionen. In dem Ablauf werden diese getrennt von den anderen ausgeführt, was nicht zwingend ist. Würde man sie aber mit den anderen gemeinsam ausführen, müsste nach jeder anderen Reduktion eine Prüfung erfolgen, ob dadurch Knoten einer bereits gefundenen Kette verändert wurden. Deshalb ist es sinnvoller, zunächst alle in Frage kommenden Knoten zu sammeln und dann getrennt zu untersuchen. Falls eine Polygon-Kette-Reduktion ausgeführt wird, werden wieder Grade von Knoten verändert, sie müssen also zu U erneut hinzugefügt werden. Die äußere WHILE-Schleife wird daher für die Rückkehr nach der Polygon-Kette-Reduktion, die innere für Grad-1 und Grad-2-/Serienreduktion benötigt.
Durch diese Zusammenfassung entsteht eine Veränderung in der Laufzeit. Innerhalb der WHILE-Schleifen können Elemente wieder neu hinzugefügt werden, d.h. die Menge der zu untersuchenden Knoten wird größer. In der inneren WHILE-Schleife gibt es zwei Stellen, an denen Knoten hinzugenommen werden. Wenn eine Grad-2- oder Serienreduktion möglich ist, müssen anschließend die beiden Endknoten wieder untersucht werden. Anzahl der Mehruntersuchungen hierdurch sind demzufolge maximal doppelt so hoch wie die Anzahl dieser Reduktionen. Bei Ausführung der Grad-1-Reduktion muss der Nachbarknoten des zu reduzierenden Knoten wieder aufgenommen werden, da sein Grad verändert wurde. Zusätzlich müssen alle Knoten, die zum Nachbarn adjazent sind, und bereits als Terminalknoten vom Grad 2 erkannten wurden, untersucht werden. Wenn ein Wechsel des Nachbarknotens von Nichterminal zu Terminal stattgefun-

den hat, könnte für diese eine Grad-2-Reduktion möglich geworden sein. Somit wäre es noch besser, dieses Hinzufügen innerhalb der Grad-1-Reduktion zu vollziehen., Darin wird unterschieden, ob der zu reduzierende Knoten ein Terminal war, was die Grundvoraussetzung für den Tausch darstellt. Zu Beginn der Polygon-Ketten-Reduktionen beträgt der Grad eines jeden Endknoten eines jeden Polygons mehr als zwei. Im Laufe der Reduktion sinkt er, so dass Grad 2 erreicht werden kann. Dies bedeutet die Rückkehr zu den vorhergehenden Reduktionen, für welche lediglich die betreffenden Knoten in U wieder aufgenommen werden müssen.
Dennoch kann festgestellt werden, dass es niemals n^2 viele Untersuchungen geben wird, da immer, wenn Knoten neu untersucht werden, vorher ebenso welche entfernt werden mussten. Somit herrscht ein Gleichgewicht, welches bewirkt, dass es lediglich eine Konstante c gibt, so dass es $O(c \cdot n) \in O(n)$ viele Untersuchungen gibt. Es ist auch festzustellen, dass nur innerhalb der inneren WHILE-Schleife ein Durchlauf der Menge U stattfindet. Somit kann man für die Laufzeitentwicklung schließen, dass dieser Durchlaufplan inklusive der darin enthalten Reduktionen den folgenden Aufwand produziert: $O(n) \cdot (O(n \cdot |K|) + O(n^2)) + n^2(|K| + |K|)) \in O(n^3)$

4 Auswertung

4.1 Gittergraph

Definition 4.1
Ein Gittergraph $G = (V, E)$ vom Typ $n \times m$ ist ein Produktgraph zweier Wege, einer der Länge $n-1$ und einer der Länge $m-1$.

In diesem Test wurden Gittergraphen vom Typ $3 \times n$ für n von 1 bis 15 gebildet, und für diese die K-Zuverlässigkeit berechnet. Die Startterminalmenge K baute sich wie folgt auf:

- Ein Knoten vom Grad 2 wird Terminalknoten
- Jeder Terminalknoten hat nur Nichterminals als Nachbarn
- Jeder Nichtterminalknoten ist nur zu Terminalknoten adjazent

Für diese Konstellation wurde untersucht, welche Reduktionen wie häufig vorkamen, deren Auftreten ist aus Tabelle 4.1 und 4.2 zu entnehmen. Bei jeder Gittervergrößerung wird G um 3 Knoten und 5 Kanten erweitert. Existierten vorher bereits k Kanten im Graphen, so ergibt sich nach der Erweiterung im schlimmsten Falle ein Aufwand von $O(2^{k+5})$, da an jedem Blatt im Verzweigungsbaum der Dekomposition ein kleinerer Baum für diese 5 Kanten angefügt werden kann. Demzufolge ist das rasante Ansteigen der Reduktionsanzahlen eine logische Konsequenz. In der Tabelle werden die Polygone von Typ 1 und 2 gemeinschaftlich erfasst. Dies entsteht durch die besondere Berechnung, in der es nicht möglich ist, nachvollziehen, welcher Typ ausgeführt wurde. In der Auswertung wird daher von Typ $1/2$ gesprochen. Eine weitere Vorbemerkung sei noch, dass innerhalb dieses Tests die Kanten zufällig nummeriert und immer die Kante mit der kleinsten Nummer zur Rekursion ausgewählt wurden. Somit kann man von einer zufälliger Auswahl der Kanten sprechen.

Zu beobachten ist, dass die Anzahl der Grad-2-Reduktionen sowohl im Gegensatz zu Grad-1-, wie auch zu den Serienreduktionen schneller anwächst. Eine Ursache für dieses Verhalten stellt die Tatsache dar, dass die Anforderungen für eine Serienreduktion geringer sind als die der Grad-2-Reduktion, da keine Bedingung an die Nachbarn gestellt werden. Somit können Serienreduktionen bereits in einem früheren Reduktionsschritt durchgeführt werden, während Grad-2-Reduktionen erst zu einem späteren Zeitpunkt möglich werden. Da sich mit jedem Schritt die Anzahl der zu untersuchenden Graphen verdoppelt, gibt es insgesamt also mehr Möglichkeiten, eine Grad-2-Reduktion durchzuführen. Dieses Verhalten wird ebenso sichtbar, wenn man sich für jeden Schritt die Knotenmenge und die Terminalmenge ansieht. Es ist zu bemerken, dass die Anzahl der Terminalknoten stets langsamer sinkt als die der Nichtterminals. Sicherlich spielt es

dabei auch eine Rolle, dass während der Kontraktion für die Rekursion die Terminals eine dominierende Rolle einnehmen, da bei der Fusion von Nichtterminal mit Terminal ein neuer Terminalknoten entsteht. Dennoch ist dies nicht der alleinige Grund, da bei gleicher Anzahl von Grad-2- und Serienreduktionen das Verhältnis ein anderes wäre.
Am häufigsten wird jedoch die Parallelreduktion ausgeführt. Beim Vergleich der Häufigkeit muss man jedoch bedenken, dass die Parallelreduktion unter anderem durch die Grad-2- oder Serienreduktion möglich wird und somit eine Abhängigkeit der Zahlen besteht. Dennoch zeigt diese hohe Anzahl an Durchführungen, dass sie eine wichtige Rolle bei der Laufzeitverkürzung darstellt. Würde sie nicht durchgeführt, könnten parallele Kanten nur durch Rekursionsschritte entfernt werden, zudem wären andere Reduktionen geblockt. Das 3×10 Gitter weist eine Anomalie auf. Die Reduktionszahlen sind sehr ähnlich denen des 3×9 Gitters, obwohl in jedem anderen Übergang die Zahlen deutlich ansteigen. Auch die Anzahl der durchgeführten Rekursionsschritte waren in allen Testläufen ähnlich. Den Grund dieser Anomalie herauszufiltern ist nicht möglich, da dafür alle Schritte nachverfolgt werden müssten (Größenordnung 15000), denn es liegt nach logischen Argumenten kein Grund dafür vor. Es ist zu vermuten, dass in diesem Falle die Nummerierung der Kanten günstiger war als in anderen Fällen (Vgl. dazu auch nächstes Kap.).
Das Verhalten der Polygon-Ketten-Reduktionen ist sicherlich noch von besonderer Wichtigkeit. Diese sind die aufwendigsten der hier vorgestellten Reduktionen, sowohl was deren Berechnung als auch deren Herleitung betrifft, dennoch können durch sie bis zu 2 Knoten bzw. 4 Kanten auf einmal entfernt werden (Typ 7). Betrachtet man jedoch die Tabelle 4.2, so folgt eine unausweichliche logische Schlussfolgerung - je komplizierter das Polygon geformt ist, umso seltener tritt es auf. Gleichzeitig bewirkt ein kompliziertes Polygon auch immer einen größeren Effekt.
Am häufigsten sind Polygone vom Typ 1/2 oder 3 entstanden. Das betrachtete Gitter hatte eine hohe Terminalknotendichte, d.h. etwa die Hälfte alle Knoten waren Terminals. Somit entstehen häufiger Polygone, in denen möglichst viele Terminals auftauchen. Gleichzeitig sind aufgrund der Gitterstruktur Ketten der Länge 3 öfter zu erwarten als Ketten der Länge 4. Interessant ist die Tatsache, dass niemals eine Typ 5 Polygon gefunden wurde. Dieses Phänomen kann man theoretisch begründen, da jedes Polygon durch eine Kombination aus Kontraktion und Entfernung von Knoten entstehen muss (Grad-2 und Serienred. werden in diesem Aspekt als Kontraktion angesehen). Dabei gilt, dass bei der Kontraktion eines Terminal- mit einem Nichterminalknoten ein Terminalknoten entsteht. Für den Typ 5 - und auch für Typ 1 - benötigt man eine Kante zwischen zwei Nichtterminals. Im Ausgangsgraphen existiert solch eine Kante nicht, und durch die Bedingung an die Kontraktion kann sie insbesondere auch nicht entstehen.
Bei größeren Gitterstrukturen ist festzustellen, dass auch Typ 6 und 7 auftreten. Typ 7 scheint dabei ein zufälliges Ergebnis zu sein, was den Piek beim 3×13 Gitter erklärt. Auch die Häufigkeit von Typ 6 unterliegt großen Schwankungen, tritt jedoch ab 3×11 beständig auf. Das Problem dieser beiden Typ ist, dass alle Gitter in einer Richtung zu schmal sind, d.h. die 3 in $3 \times n$ blockiert das Entstehen dieser Typen. Grund ist,

dass man Ketten der Länge 4 bilden muss und die Höhenbeschränkung genau dies erschwert. Es ist daher anzunehmen, dass bei einem allgemeinem $m \times n$ Gitter deren Anzahl auch systematischer anwächst, da dann genug Entstehungsfreiraum vorhanden ist. In dieser Beschränkung besteht eine noch zu große Abhängigkeit von der Kantenauswahl.

Größe	Grad-1	Grad-2	Serienr.	Parallelr.
3x1	2	0	0	0
3x2	1	0	3	1
3x3	11	14	9	15
3x4	17	30	16	44
3x5	66	105	79	132
3x6	133	281	151	298
3x7	664	1321	863	1419
3x8	3516	5970	2660	6632
3x9	20870	33044	19496	43694
3x10	19466	40434	11352	54667
3x11	367182	607629	214339	781565
3x12	831619	1680161	571190	1847122
3x13	2602399	5039101	2355335	6499677
3x14	14761674	27100164	8828871	37126445
3x15	44321835	102134233	25524148	118590876

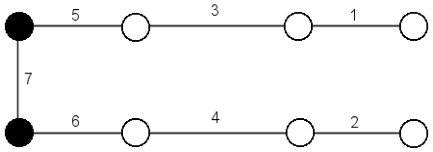
Tab. 4.1: Reduktionen für Gittergraphen

Größe	Polygon-Kette-Red.					
	Typ 1 und 2	Typ 3	Typ 4	Typ 5	Typ 6	Typ 7
3x1	0	0	0	0	0	0
3x2	0	1	0	0	0	0
3x3	2	2	0	0	0	0
3x4	5	2	0	0	0	0
3x5	11	26	0	0	0	0
3x6	21	54	4	0	0	0
3x7	230	275	0	0	8	1
3x8	542	1049	0	0	0	0
3x9	5939	6696	110	0	104	3
3x10	3470	2176	52	0	0	0
3x11	69221	29104	90	0	357	0
3x12	144584	123019	2900	0	844	0
3x13	711136	743740	1181	0	8390	1172
3x14	2313254	2605567	5516	0	684	2
3x15	7397292	6546268	31750	0	445845	0

Tab. 4.2: Polygon-Kette-Red. für Gittergraphen

4.2 Auswirkung der Kantenauswahl

Die bisherige unkontrollierte Auswahl von Kanten ist sicherlich nicht die optimale Variante, leicht an einem Beispielgraphen erkennbar (Terminals schwarz gekennzeichnet):



Wie zu sehen ist, würde nur ein Rekursionsschritt benötigt werden, wenn man die Kante '7' auswählt (durchführbare Reduktionen außer Acht gelassen). Arbeitet man jedoch die Kanten der Reihe nach (mit der kleinsten Nummer angefangen) ab, so erhält man eine Tiefe von 7 und hat insgesamt 2^7 Äste abzuarbeiten.
Es stellt sich die Frage, wie Kanten sinnvoller auszuwählen sind. Ziel sollte dabei sein, möglichst frühzeitig eine der Abbruchbedingungen zu erreichen. Es gibt verschiedene Optionen, von denen zwei hier diskutiert werden. Eine sinnvolle Verknüpfung der beiden wird ebenso angegeben.

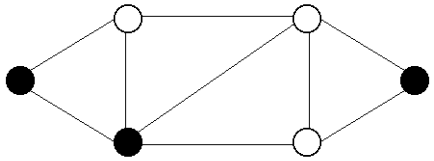
Überlegung 1: Knoten isolieren
Das Hauptproblem für eine lange Laufzeit der Rekursion liegt in den Entfernungszweigen der Dekompositionen. Dies zeigt sich sehr deutlich, wenn man die Rekursion einmal Schritt für Schritt durchläuft. Während durch die Kontraktionen häufig in wenigen Schritten Möglichkeiten zu Reduktionen entstehen, dauert es bei Entfernungen sehr lang. Ursache ist, dass nicht konsequent der Grad eines Knoten reduziert wird, sondern immer wieder an verschiedenen Stellen eine Kante entfernt wird.
Dieses Problem lässt sich auf simple Weise beheben. Anstatt dem Auswählen einer Kante aus der gesamten Kantenmenge, wird ein Knoten (aus der Terminalmenge) festgelegt und stets eine zu ihm adjazente Kante gewählt.
Jedoch kann diese Methode eine Verschlechterung für die Kontraktion zur Folge haben, da nicht mehr die Möglichkeit zu vielfältiger 'Gestaltung' besteht und somit besondere Strukturen wie z.B. Polygone seltener entstehen.

Überlegung 2: Anzahl der Terminals schnell senken
Eine andere Herangehensweise ist die Idee, möglichst häufig Kanten zu wählen, die zwischen zwei Terminals liegen. Für die Kontraktion ergibt sich der Vorteil, dass die Anzahl der Terminals wie überlegt sinkt und somit die Mächtigkeit der Terminalmenge früher den Wert 1 erreicht. Diese Vorgehensweise bringt auch für die Entfernung Vorteile. Durch diese Wahl der Kanten werden die Möglichkeiten, wie die Terminals verbunden, eingeschränkt. Allerdings ist dieser Effekt weniger stark ausgeprägt als bei der ersten Überlegung.

Überlegung 3: Kombination beider Überlegungen
Um das bestmögliche Ergebnis zu erhalten, sollten diese Überlegungen verknüpft wer-

den. Dies kann man realisieren, indem man sich einen Terminalknoten fest auswählt und von diesem aus zunächst alle Kanten benutzt, die zu einem anderen Terminal führen. Nur wenn es davon keine gibt, wählt man andere Kanten, um seinen Grad zu minimieren.
Dabei wird der ersten Überlegung mehr Beachtung geschenkt. Dies ist sinnvoll, da die Verkürzung der Entfernungszweige aus oben beschriebenen Gründen eine größere Einsparung bringt. Am besten wäre es, einen Knoten zu wählen, der einen möglichst kleinen Grad hat.

Auch wenn diese Überlegungen theoretisch erfolgversprechend klingen, so ist die Umsetzung stets an einen gewissen Mehraufwand geknüpft. Folglich muss man in jedem Fall sehr gut prüfen, ob der Mehraufwand gegenüber der Ersparnis wirklich gerechtfertigt ist.
Für diesen Graphen:



wurde eine Analyse durchgeführt. Da es sich um einen kleineren Graphen handelt, sind die Unterschiede sehr fein, jedoch bereits vorhanden.

normal	Überlegung 1	Überlegung 2	Überlegung 3
1,72ms	1,65ms	1,53 ms	1,61ms

Tab. 4.3: Laufzeitvergleich

Wie zu erwarten, senkt jede Einbindung einer Überlegung die Laufzeit. Erstaunlicherweise ist die jedoch die zweite Überlegung der dritten überlegen und liefert die beste Laufzeit. Schuld daran ist der vergrößerte Aufwand für die Suche nach den entsprechenden Kanten. Jedoch liefert diese dritte Variante schon ein leicht verbessertes Ergebnis gegenüber der Ersten. Bemerkenswert ist dies vor allem auch deshalb, da der Suchaufwand der Ersten am geringsten ist, dann folgt die Zweite und die aufwendigste Suche wird im dritten Fall betrieben.
Es lässt sich also vermuten, dass bei entsprechend größeren Graphen die letzte Variante auch die zweite überholen wird und es sich somit auszahlt, diese zu wählen, natürlich unter dem Vorbehalt, dass der Mehraufwand an sich gerechtfertigt sein muss.

Bei weiteren Untersuchungen zu diesem Sachverhalt entstand eine Variante, in der die Laufzeit bei unkontrollierter Auswahl wesentlich niedriger als bei allen Variationen lag. Der Grund dafür lag in der Nummerierung der Kanten, die im Vergleich zu einer gesteuerten Auswahl nahezu dieselbe Reihenfolge erzeugte. Natürlich ist dann eine Methode günstiger, bei der nicht noch zusätzlicher Aufwand betrieben werden muss.

Somit ergibt sich die Frage, ob eine gesteuerte Nummerierung oder eine gesteuerte Auswahl eine bessere Laufzeit ergeben. Für kleinere Graphen ist es sicherlich noch möglich, die Nummerierung als Benutzer zu optimieren. Doch bei größeren Problemen müsste man eine Automatisierung erzeugen, die wieder nach obigen Kriterien ablaufen kann.

4.3 Zufallsgraphen

In diesem letzten Abschnitt wird verglichen, wie sich für Zufallsgraphen die K-Zuverlässigkeit entwickelt.
Ein Zufallsgraph zeichnet sich dadurch aus, dass zwar seine Knotenmenge, jedoch nicht seine Kantenmenge sofort definiert ist. Vielmehr ist zusätzlich zur Kantenmenge V eine Wahrscheinlichkeit - hier mit c bezeichnet - gegeben und zwischen jedem Paar aus V wird dann mit Wahrscheinlichkeit c eine Kante eingesetzt.
Für das folgende Beispiel wird ein Zufallsgraph mit 10 Knoten benutzt, von denen 4 zufällig als Terminalknoten ausgewählt wurden. Die Wahrscheinlichkeit für jede gesetzte Kante, intakt zu sein, betrage p.
Im folgenden wurden die Werte von c und p zwischen 0.1 und 0.9 variiert und für jede Kombination die K-Zuverlässigkeit in der folgenden Tabelle notiert:

p|c	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.1	0	0.001	0	0.001	0.003	0.024	0.01	0.033	0.036
0.2	0	0	0.005	0.049	0.041	0.175	0.117	0.221	0.371
0.3	0	0	0.01	0.035	0.138	0.365	0.33	0.717	0.729
0.4	0	0.01	0.053	0.375	0.542	0.534	0.861	0.848	0.907
0.5	0	0.063	0.061	0.456	0.803	0.908	0.94	0.971	0.984
0.6	0	0.17	0.824	0.746	0.902	0.98	0.987	0.928	0.998
0.7	0	0.34	0.927	0.931	0.901	0.986	0.993	0.991	0.999
0.8	0	0.647	0.728	0.999	0.991	1	0.999	1	1
0.9	0	0.783	0.862	0.996	0.998	1	1	1	1

Tab. 4.4: K-Zuverlässigkeitswerte für Zufallsgraphen

Wie nicht anders zu erwarten, ergeben sich für kleine Werte von c und p ebenso kleine Werte für die K-Zuverlässigkeit und je mehr eine der beiden Größen ansteigt, umso mehr steigt auch deren Wert.
Betrachtet man jede Zeile für sich, wird jedoch noch ein weiterer Effekt sichtbar. Jede Zeile startet mit einem sehr geringen Wert und wächst dann an, jedoch verschieden stark. Das heißt, umso größer der Wert von p, umso stärker steigt der Wert der K-Zuverlässigkeit an. Während bis $p = 0.3$ nie ein Wert größer als 0.9 auftaucht, liegen bei $p = 0.6$ schon mehr als die Hälfte in dem Bereich zwischen 0.9 und 1.
Für Zufallsgraphen gilt also, je größer die Wahrscheinlichkeit ist, dass Kanten zwischen

Knoten existieren, umso größer ist der Wert der K-Zuverlässigkeit. Und mit sinkender Ausfallhäufigkeit steigt nicht nur der Wert, sondern es entsteht auch ein immer steileres Gefälle von niedrigen zu hohen Werten.

Anhang A: Programmcode

Das Programm besteht aus zwei Dateien. Die erste Datei ist ist 'connect.py', in er die Anfangsparallelreduktion, eventuell auftretende Initialisierungen, die Rekursion, der Aufruf der Reduktionen und der Test, ob sich die Terminalmenge in einer Komponente befindet, mit Komponentenreduktion enthalten sind. Die zweite Datei ist 'reduktionen.py', darin enthlaten ist die Suche nach Grad-1-, Grad-2-, Serien- und Polygon-Ketten-Reduktionen und die jeweilige Umsetzung dieser.

Benutzung des Programms

Um das Programm nutzen zu können, wählt man aus dem File 'connect.py' entweder:
hom_k_con(G,K,p), falls alle Kanten dieselbe Wahrscheinlichkeit p haben sollen, mit der sie intakt sind.
inhom_k_con(G,K,p), falls Kanten verschiedene Wahrscheinlichkeiten haben sollen. p ist dann vom Typ Dictonary, $p[e]$ die Wahrscheinlichkeit der e.
Beide Male ist G vom Typ MultiGraph und K vom Typ Set und beinhaltet nur Elemente der Knotenmenge von G.

connect.py

```
from mgraph import*
from reduktionen import*

"""K-TERMINAL"""
def hom_k_con(G,K,p):
    """Fkt. zum Start der Berechnung der K-Zuverlässigkeit für Graphen
    G mit Terminalmenge K und für alle Kanten gleicher Wahrscheinlich-
    keit p, das sie funtkionieren. Es wird eine Gesamtparallelred. &
    das Erstellen der Wsl.-zuordnung (Wsl. zu Kante) durchgeführt."""

    #Überprüfung, ob K in einer Komponente von G. wenn ja, Berechnung,
    wenn nein, Rückgabe 0
    if(K_connect(G,K)):
        P=dict() #Wsl.-zuordnung für weitere Berechnung
        q=dict() #q[(x,y)] - Wsl, dass alle Kanten zw. x und y
          ausfallen
        H=set() #Knotenpaare, zwischen denen Kanten existieren
        Z=dict() #Z[(x,y)] - Kanten zwischen x und y

        for e in G.EdgeSet():
            ends=G.Ends(e)
            if(len(ends)==2):
```

```
                #Initialisierung für alle Knotenpaare
                Z[ends]=[]
                q[ends]=1
                H.add(ends)
            else:
                #Schlingen löschen
                G.DeleteEdge(e)

        for e in G.EdgeSet():
            #Zuordnung - Kantennummer, Knotenpaar; Berechnung Wsl.,
            dass Knotenpaar nicht verbunden
            ends=G.Ends(e)
            Z[ends].insert(0,e)
            q[ends]=q[ends]*(1-p)

        for e in H:
            #Löschen der Kanten nach Parallelreduktion, Erstellen
            Wsl.-zuordnung für weitere Berechnung (Wsl. mit Umrech-
            nung durch Parallelreduktion)
            k=len(Z[e])
            for i in range(1,k):
                G.DeleteEdge(Z[e][0])
                Z[e].remove(Z[e][0])
            P[Z[e][0]]=1-q[e]

        return k_con(G,K,P,1)
    else:
        return 0

def inhom_k_con(G,K,p):
    """Fkt. zum Start der Berechnung der K-Zuverlässigkeit für
    Graphen G mit Terminalmenge K und Wsl.-zuordnung p. Es wird
    eine Gesamtparallelred. durchgeführt."""

    #Überprüfung, ob K in einer Komponente von G. wenn ja, Be-
    rechnung, wenn nein, Rückgabe 0
    if(K_connect(G,K)):
        q=dict() #q[(x,y)] - Wsl, dass alle Kanten zw. x und y
          ausfallen
        H=set() #Knotenpaare, zwischen denen Kanten existieren
        Z=dict() #Z[(x,y)] - Kantne zwischen x und y

        for e in G.EdgeSet():
```

```
            ends=G.Ends(e)
            if(len(ends)==2):
                #Initialisierung für alle Knotenpaare
                Z[ends]=[]
                q[ends]=1
                H.add(ends)
            else:
                #Schlingen löschen
                G.DeleteEdge(e)

        for e in G.EdgeSet():
            #Zuordnung - Kantennummer, Knotenpaar; Berechnung
            Wsl., dass Knotenpaar nicht verbunden
            ends=G.Ends(e)
            Z[ends].insert(0,e)
            q[ends]=q[ends]*(1-p[e])

        for e in H:
            #Löschen der Kanten, Umrechnung der Wsl. nach
            Parallelreduktion
            k=len(Z[e])
            for i in range(1,k-1):
                G.DeleteEdge(Z[e][0])
                Z[e].remove(Z[e][0])
                del(p[Z[e]][0])
            p[Z[e][0]]=1-q[e]

        #Aufruf der rekursiven Berechnung
        return k_con(G,K,p,1)
    else:
        return(0)

def k_con(G,K,p,i):
    """Vorschrift zur rekursiven Berechnung der K-Zuverlässig-
    keit von G. Es werden Reduktionen durchgeführt"""

    #Überprüfung, ob K nur einen Knoten enthält. wenn ja, K
    zusammenhängend in G - Rückgabe 1, wenn nein, Berechnung
    if(len(K)==1):
        return 1
    else:
        #Überprüfung, ob K in einer Komponente in G. wenn ja,
         Berechnung, wenn nein, Rückgabe 0; unter bestimmten
```

```
 Umständen ist vor Prüfung bekannt, ob K in einer
 Komponente, dann ist i=1
if(i or K_connect(G,K)):
    r=1 #allg. Reduktionsfaktor, in dem alle auftre-
    ten Reduktionsfaktoren vereint sind (via Multi-
    plikation)
    PK=set() #Knoten, die für Polygon-Kette-Red. in
    Frage kommen

    U=G.VertexSet() #Knoten, für die geprüft werden
    muss, ob Reduktionen möglich
    while(U!=set()):
        r=nRed(G,K,U,PK,p,r) #Suche nach Grad1,Grad2,
        Serienred. in dieser Funktion

        #PK ist Menge der Knoten in Ketten, die für
        Polygon-Kette-Red. in Frage kommen, wenn nicht
        leer, Such ob Red. möglich
        if(PK != set()):
            r,U=PolySuche(G,K,PK,p,r) #Suche nach Poly-
            gon-Kette-Red. in dieser Funktion

    #Rekursion
    #Überprüfung, ob K nur einen Knoten enthält. wenn
    ja, Rückgabe r, wenn nein, Berechnung
    if (len(K)==1):
        return r
    else:
        #Festlegen der Kante, die zur Rekusion benutzt
        werden soll
        e=next(iter(G.EdgeSet()))

        #Speicherung der Wsl. der Kante
        pe=p[e]
        del(p[e])

        #Setzen von u und v auf Endknoten der Kante
        u=min(G.Ends(e))
        v=max(G.Ends(e))

        #Kontrahieren des Graphen mit Parallelreduktion
        H=G/e
        Kn=K.copy() #Terminals des kontrahierten Graphen
```

```
                pn=p.copy() #Wsl.-zuordnung des kontrahierten Graphen

                #durch Kontraktion wird v gelöscht, Veränderung in
                Terminalmenge beachten
                if v in Kn:
                    Kn.remove(v)
                    Kn.add(u)

                X=G.Neighbors(u)&G.Neighbors(v) #zwischen u und den
                Knoten in X liegt ein Paar paralleler Kanten vor

                #Parallelreduktion zwischen u und den Knoten aus X
                for w in X:
                    a=-1
                    b=-1
                    for f in H.IncidentEdges(u):
                        if(w in H.Ends(f)):
                            if a==-1:
                                a=f
                            else:
                                b=f
                                break
                    H.DeleteEdge(b)
                    pn[a]=1-(1-pn[a])*(1-pn[b])
                    del(pn[b])

                #Rekusion
                return r*(pe*k_con(H,Kn,pn,1)+(1-pe)*k_con(G-e,K,p,0))
        else:
            return 0

def K_connect(G,K):
    """Funktion zur Überprüfung, ob K in einer Komponente von G liegt.
    Mit Komponentenreduktion"""

    u=next(iter(K)) # Startknoten für Untersuchung
    X={u} #Startmenge
    Y=G.Neighbors(u) #Nachbarn des Startknotens

    #Suchen der Komponente, in der sich u befindet
    while ((Y-X)!=set()):
        Z=Y-X #in der Nachbarschaft von Knoten der Komponente befind-
        liche Knoten, die noch nicht in der Komponente
```

```
        X=X|Y #Komponente um diese erweitern
        Y=Y.union(*[G.Neighbors(x) for x in Z]) #derzeitige Knoten
        der         Komponente und Nachbarn der neu hinzugenommenen

    #Überprüfung, ob alle Knoten aus K in X. wenn ja, Komponentenred.
    und Rückgabe 1, wenn nein, Rückgabe 0
    if (X >= K):
        #alle Knoten aus Knotenmenge des Graphen ohne die Menge X
        liegen in anderen Komponenten und können entfernt werden
        for v in G.VertexSet()-X:
            G.DeleteVertex(v)
        return 1
    else:
        return 0
```

reduktion.py

```
from mgraph import*

def nRed(G,K,U,PK,p,r):
    """Funktion, die für einen Graphen G mit Terminalmenge K und
    Wsl.-zuordnung p für alle Knoten aus U untersucht, ob Grad-1
    ,Grad-2 oder Serienred. möglich. Durch Red. wird der allg.
    Reduktionsfaktor r verändert und bei Auffinden Knoten in Ket-
    ten, die für PKR interessant, findet Speicherung dieser in
    PK statt"""
    while (U != set()):
        #Auswahl eines Knoten aus U für Untersuchung, Bestimmung
        seines Grads
        v=next(iter(U))
        U.remove(v)
        d=G.Degree(v)

        if d==1:
            #Grad-1-Red. ist durchzuführen
            #Nachbarn müssen erneut überprüft werden, da ihr Grad
            sinkt, falls sie in PK waren, ist ihr Grad nur noch 1,
            also auch daraus
           entfernen
            N=G.Neighbors(v)
            U|=N
```

```
            PK-=N
            r,U=Grad1(G,K,PK,U,v,p,r) #Durchführen der Grad-1-Red.

        if d==2:
            N=G.Neighbors(v) #Nachbarschaft des Knoten

            #Überprüfung, ob Knoten in K. wenn ja, Grad-2-Red oder
            in PK einfügen, wenn nein, Serienred.
            if (v in K):
                #Überprüfung, ob Nachbarschaft in K. wenn ja, Grad-
                2-Red., wenn nein, in PK einfügen
                if(K>=N):
                    #Grad-2-Red. ist durchzuführen
                    #durch vorangehende Veränderung könnte v ur-
                    sprünglich zu PK hinzugekommen sein, daraus
                    entfernen und Nachbarn nochmals untersuchen
                    PK-={v}
                    U|=N
                    r=Grad2(G,K,PK,v,p,r) #Durchführen der Grad-2
                    -Red.
                else:
                    #Hinzufügen v zu PK
                    PK.add(v)
            else:
                #Serien-red. ist durchzuführen, Nachbarn nochmals
                betrachten
                U|=N
                Serien(G,PK,v,p) #Durchführen der Serienred.
    return r

def Grad1(G,K,PK,U,v,p,r):
    """Funktion zur Durchführung der Grad-1-Red. für Knoten v aus
    Graph G mit Terminalmenge K und Wsl-zuordnung p, wodurch allg.
    Reduktionsfaktor r verändert wird. Auch werden Kettenelemente
    PK benötigt, um     unter Umständen Menge U zu verändern"""
    #da nur eine Kante zu v adjazent, ist e die Kante, die entfernt
    wird
    for e in G.IncidentEdges(v):
        #Überprüfung, ob v in K liegt. wenn ja, Veränderung r und K
        if v in K:
            r=r*p[e] #Umrechung r
            K.remove(v) #v aus K entfernen
            for w in G.Neighbors(v):
```

```
                K.add(w) #Nachbarn von v hinzufügen
                new=G.Neighbors(w)&PK #wenn w neu zu K hinzukommt,
                könnte für Knoten der Nachbarschaft, die in PK ein-
                geordnet waren, eine Grad-2 möglich werden, deshalb
                neu untersuchen
                U|=new
        del(p[e])
    G.DeleteVertex(v) #Löschen des Knoten v
    return r,U

def Grad2(G,K,PK,v,p,r):
    """Funktion, in der Grad-2-Red. für Knoten v in Graphen G mit
    Terminals K mit Wsl-zuordnung p durchgeführt wird. Dabei werden
    r und unter Umständen auch PK verändert."""
    E=[] #Kanten, die mit v verbunden sind, werden in E gespeichert
    i=0
    #Suchen&Speichern der beiden zu v inzidenten Kanten
    for e in G.IncidentEdges(v):
        E.insert(i,e)
        i=i+1

    V=[] #Nachbarknoten von v werden in V gespeichert
    i=0
    #Suchen&Speichern der beiden zu v adjanzenten Knoten
    for w in G.Neighbors(v):
        V.insert(i,w)
        i=i+1

    r=r*(1-(1-p[E[0]])*(1-p[E[1]])) #Umrechnung von r

    p[E[0]]=(p[E[0]]*p[E[1]])/(1-(1-p[E[0]])*(1-p[E[1]])) #Index
    E[0] wird behalten für die neu einzufügende Kante, Wsl. dieser
    Kante wird hier bereits darauf gespeichert
    del(p[E[1]])

    N=-1
    #Durchsuchen, ob es eine Kante zwischen den Nachbarn von v gibt
    (es kann
     maximal eine geben), in N speichern
    for e in G.IncidentEdges(V[0]):
        if V[1] in G.Ends(e):
            N=e
            break;
```

```
    #Überprüfung, ob eine gefunden wurde. wenn ja, Parallelreduktion
    if N is not -1:
        p[E[0]]=1-(1-p[E[0]])*(1-p[N]) #die Kante, die übrig bleibt
        nach der Parallelred. hat Index E[0], deshalb wir Wsl. wieder
        darauf gepseichert G.DeleteEdge(N) #Kante N löschen
        del(p[N])
        PK-={V[0],V[1]} #falls die Nachbarn vorher in PK waren,
        hatten sie Grad 2. Durch die Parallelreduktion ist ihr Grad
        nur noch 1

    G.DeleteVertex(v) #Löschen von v aus G
    K.remove(v) #Löschen von v aus K
    G.InsertEdge(E[0],V[0],V[1]) #Einfügen einer neuen Kante zwischen
    den Endknoten mit Index E[0]
    return r

def Serien(G,PK,v,p):
    """Funktion, die für Knoten v aus Graphen G mit Terminals K und
    Wslzuordnung p die Serienred. durchführt. Dabei kann PK verändert
    werden."""

    E=[] #Kanten, die mit v verbunden sind, werden in E gespeichert
    i=0
    #Suchen&Speichern der beiden zu v inzidenten Kanten
    for e in G.IncidentEdges(v):
        E.insert(i,e)
        i=i+1

    V=[] #Nachbarknoten von v werden in V gespeichert
    i=0
    #Suchen&Speichern der beiden zu v adjazenten Knoten
    for w in G.Neighbors(v):
        V.insert(i,w)
        i=i+1

    p[E[0]]=(p[E[0]]*p[E[1]]) #Index E[0] wird behalten für die neu
    einzufügende Kante, Wsl. dieser Kante wird hier bereits darauf
    gespeichert
    del(p[E[1]])

    N=-1
    #Durchsuchen, ob es eine Kante zwischen den Nachbarn von v gibt
    (es kann maximal eine geben), in N speichern
```

```
    for e in G.IncidentEdges(V[0]):
        if V[1] in G.Ends(e):
            N=e
            break;
    #Überprüfung, ob eine gefunden wurde. wenn ja, Parallelreduktion
    if N is not -1:
        p[E[0]]=1-(1-p[E[0]])*(1-p[N]) #die Kante, die übrig bleibt
        nach der Parallelred. hat Index E[0], deshalb wir Wsl. wieder
        darauf gespeichert
        G.DeleteEdge(N) #Kante N löschen
        del(p[N])
        PK-={V[0],V[1]} #falls die Nachbarn vorher in PK waren, hat-
        ten sie Grad 2. Durch die Parallelreduktion ist ihr Grad nur
        noch 1

    G.DeleteVertex(v) #Löschen von v aus G
    G.InsertEdge(E[0],V[0],V[1]) #Einfügen einer neuen Kante zwischen
    den Endknoten mit Index E[0]

def PolySuche(G,K,PK,p,r):
    """Funktion, um aus den Knoten von PK die Ketten zu finden, die
    sich im Graphen G mit den Terminals K und der Wsl-zuordnung p
    befindet. Dann werden die Polygone gesucht und die Reduktionen
    angewendet, wodurch r verändert wird. Wenn der Grad der Endknoten
    auf 2 sinkt, müssen sie wieder untersucht werden, daher wird eine
    neue Menge U kreiert und zurückgegeben."""
    global Poly
    Poly=dict() #Speichert zu jedem Paar (u,v) die gefundenen Ketten
    in einem Vektor
    U=set() #Menge der Knoten, die nochmals untersucht werden müssen

    while(PK !=set()):
        v=next(iter(PK)) #Wahl eines Knoten aus PK
        PK.remove(v)
        #Suchen der Nachbarn und speichern als n und m
        N=G.Neighbors(v)
        n=next(iter(N))
        N.remove(n)
        m=next(iter(N))
        #einer der Nachbarn könnte auch in PK sein, deshalb ebenfalls
        entfernen, falls darin enthalten
        PK.discard(m)
        PK.discard(n)
```

```
j=(min(n,m),max(n,m)) #j ist das Endknotenpaar der Kette, in
der v liegt, nach Größe geordnet
k=[j[0],v,j[1]] #k ist die Kette in der v liegt

#Überprüfung, ob der Kleinere den Grad 2 hat (d.h. es liegt
eine 4er Kette vor). Wenn ja, den Knoten in Kette einordnen,
wenn nein, anderen Endknoten überprüfen
if (G.Degree(j[0])==2):
    #den anderen Nachbarn des Knoten finden (v ist der 1.),
    das ist l
    N=G.Neighbors(j[0])
    N.remove(v)
    l=next(iter(N))
    #Überprüfung, ob l nicht in der Kette k ist (sonst ist
    es keine Kette sondern ein Kreis). wenn ja, l zur Kette
    hinzufügen und die Kette in Poly einordnen
    if(l not in k):
        #das Paar j muss verändert werden, da l hinzukommt.
        Außerdem soll das Paar j der Größe nach geordnet
        sein.
        #Überprüfung, ob l kleiner ist als j[1]. wenn ja,
         l an den Anfang der Kette und den Anfang des Paares
         setzen, wenn nein, Kettenreihenfolge & Paarreihen-
         folge tauschen
        if(l<j[1]):
            k.insert(0,l)
            j=(l,j[1])
        else:
            k=[j[1],k[1],j[0],l]
            j=(j[1],l)
        #Überprüfung, ob zu dem Paar j noch keine Kette
        gefunden wurde. wenn ja, ist k die erste und Poly[j]
        wird mit [k] initialisiert, wenn nein, k in den
        Vektor einfügen
        if(j  not in Poly):
            Poly[j]=[k]
        else:
            Poly[j].insert(0,k)
else:
    #Überprüfung, ob der Größere den Grad 2 hat (d.h. es
    liegt eine 4er Kette vor). Wenn ja, den Knoten in Kette
    einordnen, wenn nein, liegt 3er Kette vor und diese wird
```

```
in Poly aufgenommen
if(G.Degree(j[1])==2):
    #den anderen Nachbarn des Knoten finden (v ist der
    1.), das ist l
    N=G.Neighbors(j[1])
    N.remove(v)
    l=next(iter(N))
    #Überprüfung, ob lnicht in der Kette k ist (sonst ist
    es keine Kette sondern ein Kreis). wenn ja, l zur Ket-
    te hinzufügen und die Kette in Poly einordnen
    if(l not in k):
        #das Paar j muss verändert werden, da l hinzu-
        kommt. Außerdem soll das Paar j der Größe nach
        geordnet sein.
        #Überprüfung, ob l kleiner ist als j[1]. wenn ja,
        l an den Anfang der Kette und den Anfang des
        Paares setzen, wenn nein, Kettenreihenfolge &
        Paarreihenfolge tauschen
        if(j[0]<l):
            k.insert(3,l)
            j=(j[0],l)
        else:
            k=[l,j[1],k[1],j[0]]
            j=(l,j[0])
        #Überprüfung, ob zu dem Paar j noch keine Kette
        gefunden wurde. wenn ja, ist k die erste und
        Poly[j] wird mit [k] intialisiert, wenn nein, k
        in den Vektor einfügen
        if(j  not in Poly):
            Poly[j]=[k]
        else:
            Poly[j].insert(0,k)
else:
    #es liegt eine 3er Kette vor
    #Überprüfung, ob zu dem Paar j noch keine Kette
    gefunden wurde. wenn ja, ist k die erste und Poly[j]
    wird mit [k] intialisiert, wenn nein, k in den Vektor
    einfügen
    if(j  not in Poly):
        Poly[j]=[k]
    else:
        Poly[j].insert(0,k)
```

```
#für alle Paare, zwischen denen mind. eine echte Kette gefunden
wurde, gibt es in Poly einen Eintrag. Es werden alle gefunden
Paare betrachtet.
for j in Poly:
    Anz=len(Poly[j]) #Anzahl der echten Ketten zwischen dem Paar
    deg0=G.Degree(j[0]) #Grad des 1. Knoten des Paars j[0]
    deg1=G.Degree(j[1]) #Grad des 2. Knoten des Paars j[1]

    #zuerst werden Polygone betrachtet, die aus zwei echten
    Ketten bestehen. zwischen zwei Paaren existiert ein solches
    Poylgon, wenn Anz größer 1 ist
    while(Anz>1):
        #Überprüfung, ob es noch genau 2 Ketten gibt. wenn ja,
        Test ob beide Endknoten einen Grad größer 2 vor Durch-
        führung der PKR, wenn nein, PKR durchführen
        #Dies ist wichtig, da wenn der Grad von mindestens einem
        Endknoten gleich 2 ist, man nicht mehr von einem Polygon
        spricht. Solang es aber mehr als 2 Ketten gibt, können
        die Grade nicht 2 sein.
        if(Anz==2):
            #Test, ob beide noch einen Grad größer 2 haben
            if (deg0>2):
                if (deg1>2):
                    r=PolyKette2(G,K,Poly[j][0],Poly[j][1],p,r)
                    #Finden des Typs & Durchführen der Reduktion
                    für die Ketten die zu j gespeichert
                    #Anz,deg0,deg1 sinken durch das Durchführen
                    einer PKR um genau 1
                    Anz-=1
                    deg0-=1
                    deg1-=1
                    #die Ketten, die für das Polygon genutzt
                    wurden, aus Poly entfernen
                    del(Poly[j][1])
                    del(Poly[j][1])
                else:
                    break;
            else:
                break;
        else:
            r=PolyKette2(G,K,Poly[j][0],Poly[j][1],p,r) #Finden
            des Typs & Durchführen der Reduktion für die Ketten
            die zu j gespeichert
```

```
            #Anz,deg0,deg1 sinken durch das Durchführen einer
            PKR um genau 1
            Anz-=1
            deg0-=1
            deg1-=1
            #die Ketten, die für das Polygon genutzt wurden,
            aus Poly entfernen
            del(Poly[j][1])
            del(Poly[j][1])

    #an dieser Stelle gibt es noch genau eine echte Kette
    zwischen dem Paar j. Es muss geprüft werden, ob es noch eine
    Kante zwischen den Endknoten gibt. Allerdings muss dafür der
    Grad beider Endknoten wieder größer 2 sein
    if (deg0>2):
        if (deg1>2):
            #k ist echte Kette, n ist Länge der Kette
            k=Poly[j][0]
            n=len(k)
            N=-1
            #Suchen, ob eine Kante existiert
            for e in G.IncidentEdges(k[0]):
                if k[n-1] in G.Ends(e):
                    N=e
                    break;
            #Überprüfung, ob eine Kante vorhanden ist. wenn ja,
            PKR durchführen
            if N is not -1:
                r=PolyKette1(G,K,k,N,n,p,r) #Finden des Typs &
                Durchführen der Reduktion für die Kette k und die
                Kante N
                #deg0,deg1 sinken durch das Durchführen einer PKR
                um genau 1
                deg0-=1
                deg1-=1

    #wenn jeweils der Grad eines Knoten 2 ist, muss er wieder
    untersucht werden
    if (deg0==2):
        U.add(j[0])
    if (deg1==2):
        U.add(j[1])
```

```
    return r,U

#für Polygone, die aus einer echten Kette und einer Kante bestehen
def PolyKette1(G,K,Kette,Kante,n,p,r):
    """Funktion, die für eine echte Kette (Kette genannt) der Länge
    n und eine Kante (kante genannt) den Typ bestimmt und die zuge-
    hörige PKR durchführt, wodurch r verändert. Dies findet in einem
    Graphen G mit Terminals K und der Wsl-zuordnung p statt."""
    #es kommen Typ 1,2,5 in Frage
    #Überprüfung, ob die Kette Länge 3 hat. wenn ja, Typ 1 oder 2,
    wenn nein, Typ 5
    if (n==3):
    #Typ 1,2 trifft zu, es ist nicht nötigt dies weiter aufzutren-
    nen
        E=[] #Kanten des Polygons werden in E gespeichert, die Kan-
        te in E[0] ist dabei zwischen Kette[0] und Kette[1], E[1]
        zwischen Kette[1] und Kette[2]
        #Suchen&Speichern der Kanten von Kette
        for e in G.IncidentEdges(Kette[1]):
            if Kette[0] in G.Ends(e):
                E.insert(0,e)
            else:
                E.insert(1,e)

        #Umrechnungsfaktoren von 1 und 2 sind identisch zu berechnen
        a=(1-p[E[0]])*p[E[1]]*(1-p[Kante])
        b=p[E[0]]*(1-p[E[1]])*(1-p[Kante])
        c=p[E[0]]*p[E[1]]*p[Kante]*(1+(1-p[E[0]])/p[E[0]]+(1-p[E[1]])
        /p[E[1]]+(1-p[Kante])/p[Kante])

        G.DeleteEdge(Kante) #Kante aus G entfernen
        del(p[Kante])

        #Überprüfung, ob Kette[2] nicht in K liegt. wenn ja, erfolgt
        Berechnung standardmäßig, wenn nein, ist Kette[0] auf jeden
        Fall nicht in K und man muss die Berechnung 'umdrehen', d.h.
        a und b tauschen (entspricht alpha und beta aus der Theorie)
        if (Kette[2] not in K):
            return Klasse1(E,a,b,c,p,r) #Umrechnung der Wsl und des
            Reduktionsfaktors
        else:
            return Klasse1(E,b,a,c,p,r) #Umrechnung der Wsl und des
            Reduktionsfaktors
```

```
else:
#Typ 5
    #Kanten suchen
    #Kanten des Polygons werden in E gespeichert, die Kante in
    E[0] ist dabei zwischen Kette[0] und Kette[1], E[1] zwischen
    Kette[1] und Kette[2], E[2] zwischen Kette[2] und Kette[3]
    for e in G.IncidentEdges(Kette[1]):
        if Kette[0] in G.Ends(e):
            E.insert(0,e)
        else:
            E.insert(1,e)
    for e in G.IncidentEdges(Kette[2]):
        if e not in E:
            E.insert(2,e)

    #Überprüfung, ob K mehr als zwei Elemente besitzt. Dies ist
    folgt aus der Theorie. je nach dem erfoglt die Berechnung auf
    verschiedene Weise
    if(len(K)>2):

        #Umrechungsfaktoren berechnen
        a=(1-p[E[0]])*p[E[1]]*p[E[2]]*(1-p[Kante])
        b=p[E[0]]*(1-p[E[1]])*p[E[2]]*(1-p[Kante])
        c=p[E[0]]*p[E[1]]*(1-p[E[2]])*(1-p[Kante])
        d=p[E[0]]*p[E[1]]*p[E[2]]*p[Kante]*(1+(1-p[E[0]])/p[E[0]]
        +(1-p[E[1]])/p[E[1]]+(1-p[E[2]])/p[E[2]]+(1-p[Kante])
        /p[Kante])

        G.DeleteEdge(Kante) #Kante löschen
        del(p[Kante])

        return Klasse2(E,a,b,c,d,p,r) #Umrechnung der Wsl und
        des Reduktionsfaktors
    else:
        #die Berechnung erfolgt über nur eine Faktor o
        #in der neuen Struktur gibt es nur noch die Kante, nicht
        mehr die Kette
        o=p[E[1]]+p[E[0]]*(1-p[E[1]])*p[E[2]]
        p[Kante]=(p[E[1]]+p[E[0]]*(1-p[E[1]])*p[E[2]]*p[Kante])
        /o #Umrechnung der Wsl. der Kante

        #Knoten der Kette aus G und K entfernen
        G.DeleteVertex(Kette[1])
```

```
            G.DeleteVertex(Kette[2])
            K.remove(Kette[1])
            K.remove(Kette[2])
            #die Endknoten zu K hinzufügen
            K.add(Kette[0])
            K.add(Kette[3])
            del(p[E[0]])
            del(p[E[1]])
            del(p[E[2]])

            #Umrechnung des Reduktionsfaktors erfolgt direkt während
            der Rückgabe
            return r*o

#für Polygone, die aus zwei echten Ketten bestehen
def PolyKette2(G,K,K1,K2,p,r):
    """Funktion, die für zwei Ketten K1 und K2 (zusammen ein Polygon)
    den Typ bestimmt und PKR durchführt, wodurch r verändert wird.
    Dies findet in einem Graphen G mit Terminals K und Wsl-zuordnung
    p statt"""
    global Poly
    #Typ 3,4,6,7
    m=len(K1) #Länge K1
    n=len(K2) #Länge K2

    #insgesamt können Typen 3,4,6,7 auftreten
    #Überprüfung, ob eine der Ketten Länge 3 hat. wenn ja, sind Typ
    3,4,6 möglich, wenn nein, Typ 7
    if(min(m,n)==3):
        #Überprüfung, ob beide Kette Länge 3 haben. wenn ja, sind Typ
        3,4 möglich, wenn nein, Typ 6
        if(max(m,n)==3):
            #Überprüfung, ob einer der Endknoten in K liegt. wenn ja,
            Typ 3, wenn nein, Typ 4
            if ((K1[0] in K) or (K1[2] in K)):
                #Überprüfung, ob der vordere Endknoten K1[0] in K
                liegt. wichtig, da in der Berechnung eine Rolle
                spielt, welche Kanten zu welchen Knoten inzident
                sind
                if(K1[0] in K):
                    E1=[] #speichert die Kanten zu Kette K1
                    #Suchen der Kanten, E1[0] ist zwischen K1[0] und
                    K1[1], E1[1] zwischen K1[1] und K1[2]
```

```
    for e in G.IncidentEdges(K1[1]):
        if K1[0] in G.Ends(e):
            E1.insert(0,e)
        else:
            E1.insert(1,e)
    E2=[] #speichert die Kanten zu Kette K2
    #Suchen der Kanten, E2[0] ist zwischen K2[0] und
    K2[1], E2[1] zwischen K2[1] und K2[2]
    for e in G.IncidentEdges(K2[1]):
        if K2[0] in G.Ends(e):
            E2.insert(0,e)
        else:
            E2.insert(1,e)
else:
    E1=[] #speichert die Kanten zu Kette K1
    #Suchen der Kanten, E1[1] ist zwischen K1[0] und
    K1[1], E2[0] zwischen K1[1] und K1[2]
    for e in G.IncidentEdges(K1[1]):
        if K1[0] in G.Ends(e):
            E1.insert(1,e)
        else:
            E1.insert(0,e)
    E2=[] #speichert die Kanten zu Kette K2
    #Suchen der Kanten, E2[1] ist zwischen K2[0] und
    K2[1], E2[0] zwischen K2[1] und K2[2]
    for e in G.IncidentEdges(K2[1]):
        if K2[0] in G.Ends(e):
            E2.insert(1,e)
        else:
            E2.insert(0,e)

#Umrechnungsfaktoren; durch die Speicherung der Kan-
ten ist dies unabhängig von der Verteilung der Termi-
nalknoten
a=p[E1[0]]*(1-p[E1[1]])*(1-p[E2[0]])*p[E2[1]]+
(1-p[E1[0]])*p[E1[1]]*p[E2[0]]*(1-p[E2[1]])+
(1-p[E1[0]])*p[E1[1]]*(1-p[E2[0]])*p[E2[1]]
b=p[E1[0]]*(1-p[E1[1]])*p[E2[0]]*(1-p[E2[1]])
c=p[E1[0]]*p[E1[1]]*p[E2[0]]*p[E2[1]]*(1+(1-p[E1[0]])
/p[E1[0]]+(1-p[E1[1]])/p[E1[1]]+(1-p[E2[0]])/p[E2[0]]
+(1-p[E2[1]])/p[E2[1]])

#Löschen von K2 durch Löschen des inneren Knoten von
```

```
        K2 aus G und K
        G.DeleteVertex(K2[1])
        K.remove(K2[1])
        del(p[E2[0]])
        del(p[E2[1]])
        Poly[(K1[0],K1[2])].insert(0,K1) #Kette K1 ist die
        Ersatzstruktur für das Polygon aus K1 und K2 und muss
        somit zur Menge der Ketten zwischen den Endknoten
        wieder hinzugefügt werden

        return Klasse1(E1,a,b,c,p,r) #Umrechnung der Wsl und
        des Reduktionsfaktors
    else:
        E1=[] #Kanten von K1 werden in E1 gespeichert
        #Suchen der Kanten, E1[0] verläuft zwischen K1[0] und
        K1[1], E1[1] zwischen K1[1] und K1[2]
        for e in G.IncidentEdges(K1[1]):
            if K1[0] in G.Ends(e):
                E1.insert(0,e)
            else:
                E1.insert(1,e)
        E2=[] #Kanten von K2 werden in E2 gespeichert
        #Suchen der Kanten, E2[0] verläuft zwischen K2[0] und
        K2[1], E2[1] zwischen K2[1] und K2[2]
        for e in G.IncidentEdges(K2[1]):
            if K2[0] in G.Ends(e):
                E2.insert(0,e)
            else:
                E2.insert(1,e)

        #Umrechnungsfaktoren
        a=(1-p[E1[0]])*p[E1[1]]*(1-p[E2[0]])*p[E2[1]]
        b=p[E1[0]]*(1-p[E1[1]])*(1-p[E2[0]])*p[E2[1]]+
        (1-p[E1[0]])*p[E1[1]]*p[E2[0]]*(1-p[E2[1]])
        c=p[E1[0]]*p[E1[1]]*(1-p[E2[0]])*(1-p[E2[1]])
        d=p[E1[0]]*p[E1[1]]*p[E2[0]]*p[E2[1]]*(1+(1-p[E1[0]])
        /p[E1[0]]+(1-p[E1[1]])/p[E1[1]]+(1-p[E2[0]])/p[E2[0]]
        +(1-p[E2[1]])/p[E2[1]])

        #neue Struktur ist eine 4er Kette, die von K1[0] zu
        K1[1] zu K2[1] zu K2[2] verläuft. Bereits existieren-
        de Kanten werden beibehalten, und der Index E1[1]
        wird für
```

```
            die neue Kante benutzt
            G.DeleteEdge(E2[0])
            del(p[E2[0]])
            G.DeleteEdge(E1[1])
            G.InsertEdge(E1[1],K1[1],K2[1])
            Poly[(K1[0],K1[2])].insert(0,[K1[0],K1[1],K2[1],K1[2]])
             #neue Kette in Verzeichnis aufnehmen

            E=[E1[0],E1[1],E2[1]] #Kanten in Reihenfolge aus der
            Kette, für die Umrechnung wichtig
            return Klasse2(E,a,b,c,d,p,r) #Umrechnung der Wsl und
            des Reduktionsfaktors

    else:
        #Überprüfung, ob K1 kürzer als K2. Je nach dem findet
        Unterscheidung für Finden der Kanten und Erstellen der
        Ersatzstruktur fest
        if(m<n):
            E1=[] #Kanten von K1 werden in E1 gespeichert
            #Suchen der Kanten, E1[0] verläuft zwischen K1[0] und
            K1[1], E1[1] zwischen K1[1] und K1[2]
            for e in G.IncidentEdges(K1[1]):
                if K1[0] in G.Ends(e):
                    E1.insert(0,e)
                else:
                    E1.insert(1,e)
            E2=[] #Kanten von K2 werden in E2 gespeichert
            #Suchen der Kanten, E2[0] verläuft zwischen K2[0] und
            K2[1], E2[1] zwischen K2[1] und K2[2], E2[2] zwischen
            K2[2] und K2[3]
            for e in G.IncidentEdges(K2[1]):
                if K2[0] in G.Ends(e):
                    E2.insert(0,e)
                else:
                    E2.insert(1,e)
            for e in G.IncidentEdges(K2[2]):
                if e not in E2:
                    E2.insert(2,e)

            #neue Struktur, Löschen der Kette K1 durch entfernen
            des inneren Knotens aus G und K
            G.DeleteVertex(K1[1])
            K.remove(K1[1])
```

```
        Poly[(K1[0],K1[2])].insert(0,K2) #neue Kette (Ersatz-
        struktur) in Verzeichnis aufnehmen, diese ist K2
    else:
        E1=[] #Kanten von K2 werden in E1 gespeichert
        #Suchen der Kanten, E1[0] verläuft zwischen K2[0] und
        K2[1], E1[1] zwischen K2[1] und K2[2]
        for e in G.IncidentEdges(K2[1]):
            if K2[0] in G.Ends(e):
                E1.insert(0,e)
            else:
                E1.insert(1,e)
        E2=[] #Kanten von K1 werden in E2 gespeichert
        #Suchen der Kanten, E2[0] verläuft zwischen K1[0] und
        K1[1], E2[1] zwischen K1[1] und K1[2], E2[2] zwischen
        K1[2] und K1[3]
        for e in G.IncidentEdges(K1[1]):
            if K1[0] in G.Ends(e):
                E2.insert(0,e)
            else:
                E2.insert(1,e)
        for e in G.IncidentEdges(K1[2]):
            if e not in E2:
                E2.insert(2,e)

        #neue Struktur, Löschen von K2 durch Löschen des
        inneren Knoten aus G und K
        G.DeleteVertex(K2[1])
        K.remove(K2[1])
        Poly[(K1[0],K1[3])].insert(0,K1) #neue Kette (Ersatz-
        struktur) in Verzeichnis aufnehmen, diese ist K1

    #Umrechnungsfaktoren sind einheitlich für beide Fälle
    durch die Wahl von E1 und E2
    a=(1-p[E2[0]])*p[E2[1]]*p[E2[2]]*(1-p[E1[0]])*p[E1[1]]
    b=p[E2[0]]*(1-p[E2[1]])*p[E2[2]]*(p[E1[0]]*(1-p[E1[1]])+
    (1-p[E1[0]])*p[E1[1]])+p[E2[1]]*((1-p[E2[0]])*p[E2[2]]*
    p[E1[0]]*(1-p[E1[1]])+p[E2[0]]*(1-p[E2[2]])*(1-p[E1[0]])
    *p[E1[1]])
    c=p[E2[0]]*p[E2[1]]*(1-p[E2[2]])*p[E1[0]]*(1-p[E1[1]])
    d=p[E2[0]]*p[E2[1]]*p[E2[2]]*p[E1[0]]*p[E1[1]]*
    (1+(1-p[E1[0]])/p[E1[0]]+(1-p[E1[1]])/p[E1[1]]+
    (1-p[E2[0]])/p[E2[0]]+(1-p[E2[1]])/p[E2[1]]+(1-p[E2[2]])
    /p[E2[2]])
```

```
            del(p[E1[0]])
            del(p[E1[1]])

            return Klasse2(E2,a,b,c,d,p,r) #Umrechnung der Wsl und
            des Reduktionsfaktors
    else:
        E1=[] #Kanten von K1 werden in E1 gespeichert
        #Suchen der Kanten, E1[0] verläuft zwischen K1[0] und K1[1],
         E1[1] zwischen K1[1] und K1[2], E1[2] zwischen K1[2] und
         K1[3]
        for e in G.IncidentEdges(K1[1]):
            if K1[0] in G.Ends(e):
                E1.insert(0,e)
            else:
                E1.insert(1,e)
        for e in G.IncidentEdges(K1[2]):
            if e not in E1:
                E1.insert(2,e)
        E2=[] #Kanten von K2 werden in E2 gespeichert
        #Suchen der Kanten, E2[0] verläuft zwischen K2[0] und K2[1],
        E2[1] zwischen K2[1] und K2[2], E2[2] zwischen K2[2] und
        K2[3]
        for e in G.IncidentEdges(K2[1]):
            if K2[0] in G.Ends(e):
                E2.insert(0,e)
            else:
                E2.insert(1,e)
        for e in G.IncidentEdges(K2[2]):
            if e not in E2:
                E2.insert(2,e)

        #Umrechnungsfaktoren
        a=(1-p[E1[0]])*p[E1[1]]*p[E1[2]]*(1-p[E2[0]])*p[E2[1]]
        *p[E2[2]]
        b=p[E1[0]]*(1-p[E1[1]])*p[E1[2]]*((1-p[E2[0]])*p[E2[1]]
        *p[E2[2]]+p[E2[0]]*(1-p[E2[1]])*p[E2[2]]+p[E2[0]]*p[E2[1]]
        *(1-p[E2[2]]))+p[E1[0]]*p[E1[1]]*(1-p[E1[2]])*p[E2[2]]
        *(p[E2[0]]*(1-p[E2[1]])+(1-p[E2[0]])*p[E2[1]])+(1-p[E1[0]])
        *p[E1[1]]*p[E1[2]]*p[E2[0]]*((1-p[E2[1]])*p[E2[2]]+p[E2[1]]
        *(1-p[E2[2]]))
        c=p[E1[0]]*p[E1[1]]*(1-p[E1[2]])*p[E2[0]]*p[E2[1]]
        *(1-p[E2[2]])
```

```
        d=p[E1[0]]*p[E1[1]]*p[E1[2]]*p[E2[0]]*p[E2[1]]*p[E2[2]]
        *(1+(1-p[E1[0]])/p[E1[0]]+(1-p[E1[1]])/p[E1[1]]+(1-p[E1[2]])
        /p[E1[2]]+(1-p[E2[0]])/p[E2[0]]+(1-p[E2[1]])/p[E2[1]]
        +(1-p[E2[2]])/p[E2[2]])

        #neue Struktur, Löschen von K2 durch Löschen der inneren
        Knoten aus G und K
        G.DeleteVertex(K2[1])
        G.DeleteVertex(K2[2])
        K.remove(K2[1])
        K.remove(K2[2])
        del(p[E2[0]])
        del(p[E2[1]])
        del(p[E2[2]])
        Poly[(K1[0],K1[3])].insert(0,K1) #neue Kette (Ersatzstruktur)
        in Verzeichnis aufnehmen, diese ist K1

        return Klasse2(E1,a,b,c,d,p,r) #Umrechnung der Wsl und des
        Reduktionsfaktors

def Klasse1(E,a,b,c,p,r):
    """Funktion zur Umrechnung der Wsl aus p für die Kanten in E und
    r ahand der Faktoren a,b,c"""
    p[E[0]]=c/(a+c)
    p[E[1]]=c/(b+c)
    o=(a+c)*(b+c)/c
    #Umrechnung des Reduktionsfaktors erfolgt direkt während der
    Rückgabe
    return r*o

def Klasse2(E,a,b,c,d,p,r):
    """Funktion zur Umrechnung der Wsl aus p für die Kanten in E und
    r anhand der Faktoren a,b,c,d"""
    p[E[0]]=d/(a+d)
    p[E[1]]=d/(b+d)
    p[E[2]]=d/(c+d)
    o=(a+d)*(b+d)*(c+d)/(d*d)
    #Umrechnung des Reduktionsfaktors erfolgt direkt während der
    Rückgabe
    return r*o
```

Literaturverzeichnis

[1] Tittmann, Peter: Graphentheorie - Eine anwendungsorientierte Einführung; Fachbuchverlag Leipzi;, 2003

[2] Tittmann, Peter: Die Analyse der Zuverlässigkeit von Kommunikationsnetzen; Hochschule Mittweida; 1997

[3] Wood, R. Kevin; Satyanarayana, A.: Polygon-to-chain reductions and network reliability; Operations Research Center, University of California, Berkley, California; 1982

[4] Wood, R.Kevin: A Factoring Algorithm Using Polygon-to-Chain Reductions for Computing K-Terminal Network Reliability; Department of Operations Research, Naval Postgraduate School, Monterey, California; 1984

[5] Resende, Mauricio G. C.: A Program for Reliability Evaluation of Undirected Netwroks via Polygon-to-Chain Reductions; Operations Research Center, University of California, Berkley, California; 1985

[6] Ball, Michael O.: Computational Complexity of Network Reliability Analysis: An Overview; College of Business & Management, University of Maryland, College Park, Maryland; 1985

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 17.02.2012