Bachelorarbeit
Algorithmen für Baum- und Pfadzerlegungen von Graphen

Sebastian Küpper

14. Oktober 2010

Betreuer: Prof. Dr. Barbara König
Dr. H. J. Sander Bruggink
Dipl. Inform. Christoph Blume

Tag der Anmeldung: 25. August 2010
Tag der Abgabe: 25. Oktober 2010

Inhaltsverzeichnis

1 Einleitung

1.1 Pfadzerlegungen, Baumzerlegungen und ihre Anwendungen

Die vorliegende Arbeit beschäftigt sich mit dem Problem der Baum- und Pfadzerlegung von Graphen. Die Begriffe Baum- und Pfadzerlegung wurden 1986 ([15]) respektive 1983 ([16]) von Neil Robertson und Paul D. Seymour in einer Reihe von Veröffentlichungen zur Graph-Minoren-Theorie eingeführt (Hierfür und den Rest dieses Abschnitts: [3]). Pfad- und Baumzerlegungen können in einer Vielzahl von Anwendungsgebieten genutzt werden. So ist das Problem der Baumzerlegung eng verwandt mit dem der Cholesky-Zerlegung, die genutzt werden kann um Gleichungssysteme mit dünn besetzten symmetrischen Matrizen effizient zu lösen. Neben der Numerik findet das Konzept auch in der Molekularbiologie Anwendung. Die Suche nach einem perfekt-phylogenetischen Baum, einem Modell für die Vererbung zwischen Spezies, ist ein Problem, das äquivalent zur Suche nach einer speziellen Baumzerlegung ist. Aus Sicht der theoretischen Informatik sind die Probleme der Pfad- und der Baumzerlegung besonders auf Grund der Anwendungen in der Graph-Minoren-Theorie und deshalb interessant, weil für festgehaltene Pfad- beziehungsweise Baumweiten viele (nicht alle) NP-vollständige Probleme auf Graphen in polynomieller oder gar linearer Zeit lösbar sind. Es sei jedoch gleich darauf hingewiesen, dass sich bei den zugehörigen Polynomzeitalgorithmen eine Potenz in der Pfad- respektive Baumweite als multiplikativer konstanter Faktor verbirgt. So sind viele der nachfolgend kurz genannten Ergebnisse zwar theoretisch interessant, praktisch aber oftmals kaum zu gebrauchen.
Zu den NP-vollständigen Problemen, die bei festgehaltener Baumweite oder Pfadweite in polynomieller Zeit lösbar sind gehören beispielsweise das Problem des Hamilton-Kreises oder das Independent-Set-Problem. Darstellungen dieser Probleme finden sich zum Beispiel in [19], eine umfangreiche Auflistung von Quellen für diese und weitere derartige Ergebnisse in [3]. Da das Problem der Pfadzerlegung mit minimaler Pfadweite selbst ebenfalls ein NP-hartes Problem ist, kann im vierten Kapitel ein Beispiel für einen Algorithmus gefunden werden, der ein NP-hartes Problem bei festgehaltener Pfadweite in Linearzeit löst.
Im Bereich des maschinellen Lernens kann ein Lernalgorithmus auf Baumzerlegungen von Graphen, statt auf den Graphen selbst, angewandt werden. Je niedriger dabei die Baumweite ist, desto effizienter ist der entsprechende Lernalgorithmus ([8]).

1.2 Motivation und Aufbau der Arbeit

Ausgangspunkt für die Beschäftigung mit diesem Thema war es, automatisch günstige Eingaben für das von Christoph Blume in seiner Diplomarbeit [2]

vorgestellte Verfahren zur Spezifikation und Verifikation von Invarianten mit Hilfe von Graphsprachen zu generieren. Aus einer Pfadzerlegung kann mit geringem Aufwand eine passende Eingabe für das zugehörige Programm generiert werden. Zunächst wird kurz dargestellt, worum es bei der Verifikation mit Hilfe von Graphsprachen geht; dafür sind einige Grundbegriffe der Kategorientheorie notwendig. Anschließend werden grundlegende, in dieser Arbeit verwendete, Begriffe wie Graph oder Pfadzerlegung definiert.
Darauf aufbauend werden in den folgenden Kapiteln drei verschiedene Algorithmen zur Pfadzerlegung vorgestellt. Der erste Algorithmus garantiert Optimalität der Pfadzerlegung. Im Gegenzug ist dieser Algorithmus sehr aufwendig und bedarf zudem einer guten Baumzerlegung als Eingabe. Mithilfe des zweiten Algorithmus' kann eine Baumzerlegung gebildet werden, die nur logarithmisch von der optimalen Baumzerlegung abweicht. Basierend auf einer so gewonnenen Baumzerlegung kann dann eine Pfadzerlegung gewonnen werden. Der dazu benötigte Algorithmus wird ebenfalls im gleichen Kapitel vorgestellt. Da dieser Algorithmus trotz polynomieller Laufzeit für große Graphen kaum zu gebrauchen ist, wird außerdem ein schnellerer Algorithmus vorgestellt, der ebenfalls Pfadzerlegungen generiert, die allerdings unter Umständen sehr stark von der optimalen Pfadzerlegung abweichen.
Die beiden Approximationsalgorithmen wurden im Zuge dieser Arbeit auch implementiert, einige Anmerkungen zum Programmcode und der Verwendung des Programms werden im Anschluss an die Vorstellung der Algorithmen gegeben. Mithilfe dieser Implementierungen wurde außerdem ein Vergleichstest der Laufzeit und der Güte der Ergebnisse der beiden Ergebnisse für verschieden große Graphen angefertigt, die Ergebnisse dieses Tests finden sich ebenfalls in diesem Abschnitt.

2 Graphsprachen

2.1 Grundlagen der Kategorientheorie

Das nachfolgende Kapitel dient nur der Erklärung der angedachten Anwendung der Ergebnisse dieser Arbeit. Inhaltlich ist dieses Kapitel unabhängig von den übrigen Kapiteln und kann daher auch übersprungen werden.
Die von Christoph Blume in seiner Diplom-Arbeit verwendeten Graphsprachen basieren auf Begriffen aus der Kategorientheorie. Diese dient vornehmlich als Verallgemeinerung der Mengenlehre. Zunächst ist zu klären, was eine Kategorie überhaupt ist, außerdem benötigen wir zumindest die Definitionen des Pushouts und des Cospans, um zu verstehen, wozu die in den folgenden Kapiteln gesuchten Pfadzerlegungen gebraucht werden können. Die Definitionen in diesem Kapitel folgen [14].

Definition 2.1 ***Kategorie***
Eine Kategorie Cat ist ein Tupel aus einer Sammlung von Objekten O, einer Sammlung von Pfeilen P (oder auch Morphismen) und einem Kompositionsoperator $\circ$ für die gilt:
Jedem Pfeil f ist ein Definitionsbereich dom $f \in O$ und ein Bildbereich cod f $\in O$ zugeordnet. Für einen Pfeil f mit dom $f = A$ und cod $f = B$ schreiben wir auch $f : A \to B$. Der Kompositionsoperator $\circ$ verknüpft zwei Pfeile f und g genau dann als $g \circ f$, wenn cod f = dom g. Für den Kompositionsoperator gilt das Assoziativgesetz, das heißt für drei Pfeile f: $A \to B$, g: $B \to C$, h: $C \to D$ gilt $h \circ (g \circ f) = (h \circ g) \circ f$. Zudem gibt es für jedes Objekt $A \in O$ einen Identitätspfeil $id_A : A \to A$. Die Identitätspfeile genügen dem Identitätsgesetz, das heißt es gilt für jeden Pfeil $f : A \to B$: $id_B \circ f = f$ und $f \circ id_A = f$.

Eines der wichtigsten Konzepte der Kategorientheorie ist der Funktor, eine strukturerhaltende Abbildung zwischen verschiedenen Kategorien.

Definition 2.2 ***Funktor***
Seien C und D Kategorien, dann ist ein Funktor $F : C \to D$ eine Abbildung, die jedes C-Objekt A auf ein D-Objekt F(A) abbildet und jeden C-Pfeil $f : A \to B$ auf einen D-Pfeil $F(f) : F(A) \to F(B)$ abbildet, so dass für alle C-Objekte A und konkatenierbare C-Pfeile f und g gilt: $F(id_A) = id_{F(A)}$ und $F(g \circ f) = F(g) \circ F(f)$.

Definition 2.3 ***Pushout***
Seien $f : A \to B$ und $g : A \to C$ Pfeile einer Kategorie. Ein Pushout dieser Pfeile ist ein Objekt P und zwei Pfeile $f' : C \to P$ und $g' : B \to P$, so dass $f' \circ g = g' \circ f$ und wenn es ein Objekt X - das Pushoutobjekt - und zwei Pfeile $f'' : C \to X$ und $g'' : B \to X$ gibt, für die ebenfalls $f'' \circ g = g'' \circ f$ gilt, dann gibt es genau einen Pfeil $k : P \to X$, so dass $f'' = k \circ f'$ und $g'' = k \circ g'$.

Vgl. auch folgende Abbildung:

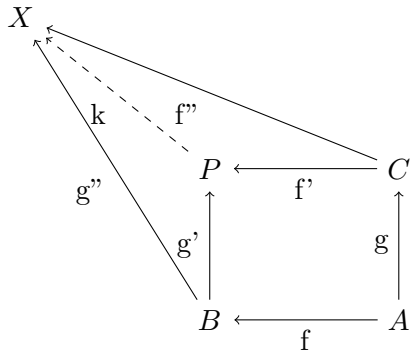


Abbildung 1: Pushout

Wichtig für das Verständnis von Graphsprachen ist schließlich noch der Begriff des Cospans. Diese Definition ist [7] entnommen.

Definition 2.4 ***Cospan und Cospan-Kategorie***
Sei C eine Kategorie in der für alle Paare von Pfeilen mit gleichem Definitionsbereich der Pushout existiert. Ein Cospan ist ein Paar von Pfeilen aus dieser Kategorie (c_L, c_R) *mit dem gleichen Bildbereich, wir schreiben:*
$c : J \overset{c_L}{\to} G \overset{c_R}{\leftarrow} K$.
Die Konkatenation zweier Cospans $c_1 : J \overset{c_{1_L}}{\to} G \overset{c_{1_R}}{\leftarrow} M$ *und* $c_2 : M \overset{c_{2_L}}{\to} H \overset{c_{2_R}}{\leftarrow} K$ *bilden wir wie folgt. Sei M' das Pushoutobjekt des Pushouts von* c_{2_L} *und* c_{1_R}, *f der Pfeil von G nach M' und g der Pfeil von H nach M'. Dann ist die Konkatenation von* c_1 *und* c_2 *der Cospan* $J \overset{f \circ c_{1_L}}{\to} G \overset{g \circ c_{2_R}}{\leftarrow} K$.
Wir nennen zwei Cospans $c_1 : J \overset{c_{1_L}}{\to} G \overset{c_{1_R}}{\leftarrow} K$ *und* $c_2 : J \overset{c_{2_L}}{\to} H \overset{c_{2_R}}{\leftarrow} K$ *äquivalent wenn es einen Isomorphismus zwischen G und H gibt.*
Mit dieser Definition ist auch Cospan(C), die Kategorie mit den Objekten von C als Objekten, den Äquivalenzklassen der Cospans über C als Pfeilen und der oben definierten Konkatenation, tatsächlich eine Kategorie. Die Identitätspfeile dieser Kategorie sind die Cospans die aus Paaren von zwei Identitätspfeilen aus C gebildet werden.

2.2 Erkennbare Graphsprachen

Die Definitionen dieses Kapitels folgen [7]. Zunächst definieren wir die Kategorie HGraph von Hypergraphen und Graphmorphismen. Somit erhalten wir auch ein erstes Beispiel für eine Kategorie.

Definition 2.5 ***Die Kategorie HGraph***
Ein Hypergraph ist ein Vier-Tupel $G = (V_G, E_G, att_G, lab_G)$. V_G *ist die Menge*

der Knoten, E_G die Menge der Kanten, $att : E_G \to V_G^*$ die Verknüpfungsfunktion (wobei V_G^* die Menge der Sequenzen von Elementen aus V_G ist) und $lab_G : E_G \to \Sigma$ die Label-Funktion. Ein Graphmorphismus f zwischen zwei Graphen G und H ist ein Paar von Funktionen (f_V, f_E) mit $f_V : V_G \to V_H$ und $f_E : E_G \to E_H$ so dass gilt:
$lab_H \circ f_E = lab_G$ und $att_H \circ f_E = f_V^* \circ att_G$.
Dabei ist f_V^* die Erweiterung von f_V auf Sequenzen, die sich ergibt, wenn man f_V elementweise auf Sequenzen anwendet.
Den Graphen mit leerer Knoten- und Kantenmenge nennen wir $\emptyset$.
Die Kategorie HGraph ist also die Kategorie, die Hypergraphen als Objekte und Graphmorphismen als Pfeile hat.

Wichtig ist vor allem die Kategorie Cospan(HGraph), also die Kategorie die Cospans von Hypergraphen als Pfeile hat. Mit dieser Vorbereitung können wir nun erkennbare Pfeilsprachen und schließlich erkennbare Graphsprachen definieren.

Definition 2.6 ***Erkennbare Pfeilsprache***
Sei C eine Kategorie in der die Sammlung der Pfeile zwischen jedem Paar von Objekten eine Menge, genannt homset, ist. Eine Teilmenge eines homsets ist eine Pfeilsprache. Sei nun A ein Automatenfunktor, das heißt:

- *$A : C \to Rel$ (Rel ist die Kategorie, die Mengen als Objekte und Relationen als Pfeile hat) ist ein Funktor der jedes Objekt X aus C auf eine endliche Menge A(X), der Menge an Zuständen von X, abbildet und jeden Pfeil $f : X \to Y$ auf eine Teilmenge von $(A(X) \times A(Y))$ abbildet.*

- *Jedes A(X) enthält eine ausgezeichnete Menge von Startzuständen $I_X^A \subset A(X)$ und eine ausgezeichnete Menge von Endzuständen $F_X^A \subset A(X)$.*

Dann bezeichnet $L_{J,K}(A)$ die (J,K)-Pfeilsprache die genau die Pfeile $f : J \to K$ enthält, für die es ein $s \in I_J^A$ und ein $t \in F_K^A$ gibt, so dass $(s,t) \in A(f)$. Ein Pfeilsprache $L_{J,K}$ ist genau dann erkennbar, wenn es einen Funktor $A : C \to Rel$ gibt mit $L_{J,K} = L_{J,K}(A)$.

Damit ist eine erkennbare Graphsprache schließlich eine erkennbare Pfeilsprache in der Kategorie Cospan(HGraph).

2.3 Ein Automatenfunktor zum Prüfen von Invarianten

Christoph Blume hat in seiner Diplomarbeit einen Automatenfunktor vorgestellt, der die Sprache aller Graphen erkennt, die einen bestimmten Subgraphen U enthalten. Um Zerlegungen von Graphen in atomare Graphenoperationen zu generieren, können Pfadzerlegungen genutzt werden. Die folgende

Darstellung folgt der Darstellung in [2]. Der Automatenfunktor verfügt für jeden Graphen G über die Zustandsmenge $A(G) = \{(U_G, f_G) \mid U_G \leq U, f_G : V_G \to V_{U_G}\}$. Dabei ist U_G ein Subgraph des Graphen U, der bisher erkannte Graph, und f_G bildet jeden Knoten aus G auf den entsprechenden Knoten aus U_G ab, sofern es einen solchen gibt, andernfalls ist der Funktionswert nicht definiert. Weiter ist die Startzustandsmenge definiert als:

$$I_G^A = \begin{cases} \{(\emptyset, \emptyset)\}, & \text{falls } G = \emptyset \\ \emptyset, & \text{sonst} \end{cases}$$

und die Endzustandsmenge als:

$$F_G^A = \begin{cases} \{(U, \emptyset)\}, & \text{falls } G = \emptyset \\ \emptyset, & \text{sonst} \end{cases}.$$

Als Pfeile werden nur die sechs Operationen res, perm, trans, fuse, edge und vertex zugelassen. Mit „Interface“ bezeichnen wir die momentane Arbeitsmenge an Knoten, die genutzt werden um die Operationen anzuwenden. Wenn Knoten das erste Mal verarbeitet werden, werden sie in diese Arbeitsmenge aufgenommen. Wenn sie nicht mehr benötigt werden, um Kanten des Graphen zu bilden, können sie aus dieser Arbeitsmenge entfernt werden. Die Operationen werden in Christoph Blumes Diplomarbeit formal als Cospans definiert. Da der Automatenfunktor nur der Motivation dient und nicht Mittelpunkt der vorliegenden Arbeit ist, wird im Folgenden nur einen Überblick darüber gegeben, was die Operatoren bewirken:

- **res**: Die Operation res_n (für Restriktion) entfernt den gemäß der aktuellen Aufzählung letzten Knoten aus dem Interface.
- **perm**: Mit der Operation $perm_n$ (für Permutation) wird die Aufzählung der Knoten im Interface so verändert, dass der letzte Knoten an die erste Stelle vorrückt und alle anderen eine Stelle nach hinten weiterrücken.
- **trans**: Wendet man die Operation $trans_n$ (für Transposition) an, so tauscht man die Position der ersten beiden Knoten in der Aufzählung der Knoten im Interface, alle anderen Knoten behalten ihre Position bei.
- **fuse**: Die Operation $fuse_n$ verschmilzt die ersten beiden Knoten der Knoten im Interface, so dass es hinterher nur noch einen Knoten gibt, der alle Verknüpfungen der beiden Ursprungsknoten mit Kanten beibehält. Wird also der Knoten v mit dem Knoten u des Interface-Graphen I_1 zum Knoten w des Interface-Graphen I_2 verknüpft, so gilt für alle $(x, (*, ..., *, v, *, ..., *)) \in att_{I_1}$ $(x, (*, ..., *, w, *, ..., *)) \in att_{I_2}$ und für alle $(x, (*, ..., *, u, *, ..., *)) \in att_{I_1}$ $(x, (*, ..., *, w, *, ..., *)) \in att_{I_2}$.

- **edge**: Mit der Operation $edge_n^{A,m}$ fügt man dem Interface eine Hyperkante mit m Knoten und der Beschriftung A hinzu. Waren vorher n Knoten im Interface, so sind dann nach Ausführung der Operation $n + m$ Knoten im Interface.

- **vertex**: Wendet man die Operation $vertex_n$ an, so wird dem Interface ein weiterer Knoten hinzugefügt.

Jede der obigen Operationen muss für jede Interface-Größe n einzeln definiert werden, daher tragen sie zusätzlich den Index n.

Interessant ist noch, welchen Einfluss die Operationen auf den bisher erkannten Graphen haben. Keine Änderung am bisher erkannten Graphen haben die Operationen $res, perm, trans, fuse$ (da diese vier keine neuen Informationen ins Interface hinzufügen, $fuse$ ist allerdings nur anwendbar, wenn die beiden zu vereinigenden Knoten auf den selben Knoten in U_G abbilden). Hingegen verändert die Operation *edge* den erkannten Graphen U_G. Nichtdeterministisch wird entschieden,

- ob die neue Kante im gesuchten Graphen U vorkommt, in dem Fall wird U_G um die Kante und die zugehörigen Knoten die noch nicht in U_G sind erweitert,

- ob die Kante zwar nicht im gesuchten Graphen U vorkommt, einige der Knoten aber schon, in dem Fall werden die neuen Knoten die noch nicht in U_G enthalten sind hinzugefügt,

- oder aber ob weder die neue Kante noch die zugehörigen Knoten im gesuchten Graphen vorkommen, in dem Fall wird U_G nicht verändert.

Es bleibt die Operation *vertex* zu betrachten, für die Ähnliches gilt wie für *edge*, es wird also nichtdeterministisch entschieden, ob der neu hinzugefügte Knoten im Graphen U vorkommt. Falls ja, wird er U_G hinzugefügt, sonst wird U_G nicht verändert.

Die Darstellung der kategorientheoretischen Aspekte, der erkennbaren Graphsprachen und des Automatenfunktors sind in dieser Arbeit nur sehr knapp erfolgt, um eine Idee für die Motivation hinter der Beschäftigung mit Pfadzerlegungen zu geben. Für wesentlich exaktere und umfangreichere Beschreibungen dieser Aspekte sei auf die in diesem Kapitel referenzierten Quellen [14], [7] und [2] verwiesen.

3 Einige Definitionen

3.1 Grundlegende Definitionen der Graphentheorie

Auch wenn wir zuvor bereits den Begriff des Hypergraphen kennen gelernt haben, definieren wir hier noch einmal Graphen. Im Folgenden betrachten wir nämlich nur Graphen, bei denen Kanten höchstens zwei Knoten miteinander verbinden. Der hier definierte Graph ist ein Spezialfall des zuvor definierten Hypergraphen, der die *Att*-Funktion (hier h genannt) auf Paare von Knoten einschränkt. Eine Kante kann also nur noch höchstens zwei statt beliebig vieler Knoten miteinander verbinden. Die folgenden Definitionen sind wenn nicht anders angegeben [18] entnommen.

Definition 3.1 ***Graph***
Seien V,E Mengen mit $V \cap E = \emptyset$ *und* $h : E \to V \times V$ *eine Abbildung. Dann ist (V,E,h) ein (gerichteter) Graph oder Digraph. Ist h injektiv, so identifizieren wir E auch mit h(E) und schreiben für* $e \in E$*:* $e = h(e) = (u, v)$ *für ein u und ein v aus E. V nennen wir die Knotenmenge (oder auch Eckenmenge), E die Kantenmenge (oder auch Bogenmenge) und h die Kantenabbildung.*

Bemerkung 3.2 ***Endlichkeit***
In dieser Arbeit betrachten wir nur endliche Graphen, das bedeutet, $|E| < \infty$ *und* $|V| < \infty$*, da unendliche Graphen mit Computern nur schwer zu handhaben sind und sich viele Prozesse bereits mit endlichen Graphen modellieren lassen. Des Weiteren betrachten wir der Einfachheit der Notation halber in dieser Arbeit nur Graphen mit injektiver Kantenabbildung. Daher identifizieren wir wie in der Definition die Kanten mit den Knoten, die sie verbinden und schreiben für einen Graphen G=(V,E,h) auch kurz G=(V,E). Zudem bezeichnen wir mit grad(v) für ein* $v \in V$ *die Anzahl der Kanten, die mit v inzidieren, formal:* $|\{(u, w) \in E | u = v\}| + |\{(u, w) \in E | w = v\}|$.

Beispiel 3.3 ***Beispielgraph***
Abbildung 2 stellt den Graphen

$$G = (\{A, B, C, D\}, \{(A, B), (B, C), (C, A), (C, D)\})$$

dar. Um die Kanten leicht bezeichnen zu können, haben sie im Beispiel Label erhalten. So ist die Kante (A,B) mit dem Label a, die Kante (B,C) mit dem Label b, die Kante (C,A) mit dem Label c und die Kante (C,D) mit dem Label d versehen.

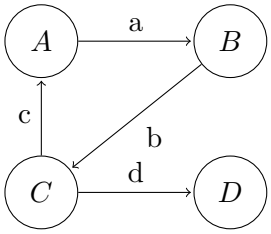


Abbildung 2: Beispielgraph

Definition 3.4 ***Isomorphie zwischen Graphen***
Wir nennen zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zueinander isomorph, wenn es eine bijektive Abbildung h von V_1 nach V_2 und eine bijektive Abbildung g von E_1 nach E_2 so gibt, dass für alle $e = (u, v) \in E_1$ gilt: $g(e) = (h(u), h(v))$.

Beispiel 3.5 ***Zwei isomporphe Graphen***
Abbildung 3 stellt den Graphen

$$G_1 = (\{A, B, C, D\}, \{(A, B), (B, C), (C, A), (C, D)\})$$

und Abbildung 4 den Graphen

$$G_2 = (\{A, B, C, D\}, \{(D, C), (C, B), (B, D), (B, A)\})$$

dar. Die Isomorphie ergibt sich durch Abbilden des Knotens A aus G_1 auf D aus G_2, von B aus G_1 auf C aus G_2, von C aus G_1 auf B aus G_2 und von D aus G_1 auf A aus G_2. Weiterhin werden die Kanten aus G_1 auf die Kanten gleichen Labels aus G_2 abgebildet.

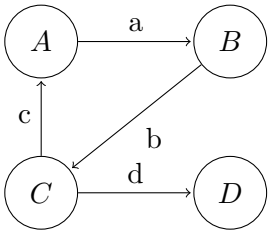


Abbildung 3: G_1

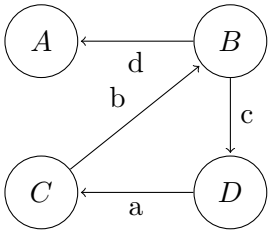


Abbildung 4: G_2

Definition 3.6 ***Kantenfolge, Weg und Kreis***
Für einen Graph G=(V,E) bezeichnen wir eine Folge von Kanten

$$\{(u_i, u_{i+i})\}_{i\in\{1,...,p\}}$$

als Kantenfolge. Wir schreiben für eine Kantenfolge auch $(u_1, u_2, ..., u_p)$*. Ein Weg ist eine Kantenfolge f für die gilt, dass für alle* $u_i, u_j \in f$ *mit* $i \neq j$ *gilt:* $u_i \neq u_j$*. Eine Kantenfolge* $f = (u_1, u_2, ..., u_p)$ *heißt Kreis, wenn* $u_1 = u_p$ *und* $(u_1, u_2, ..., u_{p-1})$ *ein Weg ist.*

Definition 3.7 ***Zusammenhangskomponente***
Sei $G = (V, E)$ *ein Graph,* V' *eine Teilmenge der Knotenmenge und* $V^* = V \setminus V'$*. Dann ist mit* $E' = \{(u, v) \in E \mid u \in V' \wedge v \in V'\}$ $G' = (V', E')$ *eine (Zusammenhangs-) Komponente des Graphen G, wenn gilt, dass:*
$\forall v \in V^*(\neg\exists u \in V'((u, v) \in E \vee (v, u) \in E))$ *und*
für alle u und v in V' *existiert ein (ungerichteter) Weg von u nach v in* G'
Ein Graph mit nur einer Zusammenhangskomponente heißt zusammenhängend.

Beispiel 3.8 ***Graph mit drei Zusammenhangskomponenten***
Der Graph im folgenden Bild hat drei Zusammenhangskomponenten, eine, die die Knoten A, B, C und D enthält, eine die die Knoten E und F enthält und eine die den Knoten G enthält.

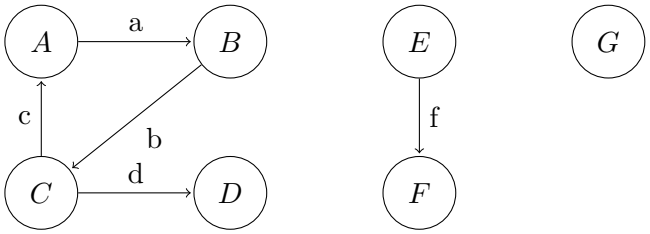


Abbildung 5: Graph mit drei Komponenten

Algorithmus 3.9 ***Finden einer Zusammenhangskomponente***
Sucht man die Zusamenhangskomponenten eines Graphen G=(V,E), so bietet sich folgendes Vorgehen an:

0. i:=1, markiere alle Knoten in V als noch nicht besucht.

1. Wähle einen beliebigen noch nicht als besucht markierten Knoten v aus V aus. $K_i = \{v\}$ ist dann die Initialisierung der i-ten Knotenmenge, die die i-te Komponente enthält. Markiere v als besucht.

2. Mache von v aus eine Tiefensuche und füge alle Knoten, die bei der Tiefensuche gefunden werden, zu K_i hinzu, markiere diese Knoten als besucht.

3. Gibt es noch einen Knoten in V, der nicht als besucht markiert wurde, setze i:=i+1 und fahre bei 1 fort, sonst gib die Menge aller gerade gebildeten $K_j, j = 1, ..., i$ zurück.

Definition 3.10 ***Baum***
Ein Graph, in dem es keinen Kreis gibt und in dem es zwischen jeweils zwei Knoten (genau) einen Weg gibt, heißt Baum. Ein Baum heißt Baum mit Wurzel oder Wurzelbaum, wenn es einen einzelnen als Wurzel ausgezeichneten Knoten gibt.

Beispiel 3.11 ***Beispielbaum***
In der folgenden Abbildung ist ein Graph zu sehen, der ein Baum ist. Der Knoten A kann als Wurzelknoten gewählt werden, dann handelt es sich um einen Baum mit Wurzel.

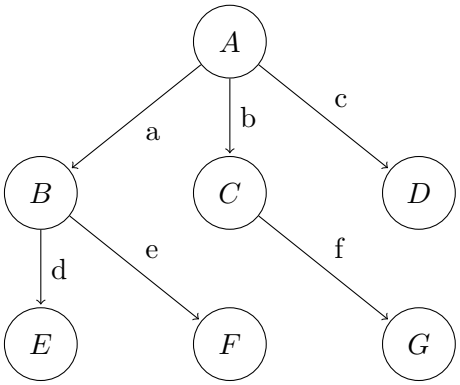


Abbildung 6: Beispielbaum

3.2 Pfad- und Baumzerlegungen

Für Pfad- und Baumzerlegungen ist nicht wichtig, in welche Richtung eine Kante des zu zerlegenden Graphen zeigt. Wir betrachten also im Folgenden statt Kanten (u, v) Äquivalenzklassen $[(u, v)]$ in denen sowohl (u, v) als auch

(v, u) enthalten sind. Wir identifizieren außerdem die Äquivalenzklassen mit ihren Vertretern.

Definition 3.12 ***Baumzerlegung***
Eine Baumzerlegung eines Graphen $G = (V, E)$ ist das Paar $(\{X_i \mid i \in I \subset \mathbb{N}\}, T = (I, F))$ mit den Eigenschaften:

- *T ist ein Baum*
- *$X_i \subset V$ für alle $i \in I$*
- *$\cup_{i \in I} X_i = V$*
- *Für jede Kante $(u, w) \in E$ des Graphen G gibt es ein $i \in I$ mit $v, w \in X_i$. Es gibt also für jede Kante eine Menge X_i, so dass die Knoten, die über die Kante verbunden werden, in ihr enthalten sind.*
- *Für alle $i, j, k \in I$ gilt: Liegt j in T auf dem Pfad von i nach k, so gilt $X_i \cap X_k \subset X_j$. Also gilt für zwei Knoten i und k des Baumes T: Ist ein Knoten des Graphen G in der durch i repräsentierten Menge X_i enthalten und ebenfalls in der durch k repräsentierten Menge X_k enthalten, so ist er auch in jeder durch einen Knoten j auf dem Pfad zwischen i und k repräsentierten Menge X_j enthalten. Nach [4]*

Eine Baumzerlegung ist eine Baumzerlegung mit Wurzel, wenn T ein Wurzel-Baum ist.
Baumzerlegungen können auch grafisch dargestellt werden. Man stellt eine Baumzerlegung grafisch dar wie einen Graphen, die Knoten dieses Graphen sind die Knoten von T und die Kanten die Kanten von T. Die Knoten werden allerdings groß und gestrichelt gezeichnet, so dass die Knoten des zerlegten Graphen, die von dem jeweiligen Knoten des Baums repräsentiert werden, in dem Knoten dargestellt werden können. Wir nennen die Knoten von T auch „Taschen“, da sie mehrere Knoten des Graphen repräsentieren, also wie eine Tasche fassen können.

Im folgenden Beispiel verwenden wir die graphische Darstellungsmöglichkeit für Baumzerlegungen, denn eine rein formale Darstellung ist wenig dienlich um die Idee hinter einer Baumzerlegung zu vermitteln. Die formale Definition ist vor allem für die Beschreibung der Algorithmen wichtig.

Beispiel 3.13 ***Baumzerlegung eines Beispielgraphen***
Wir betrachten den Graphen aus der folgenden Darstellung:

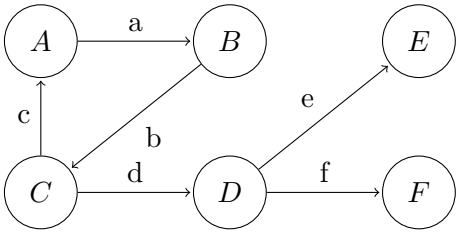


Abbildung 7: Zu zerlegender Beispielgraph

Eine mögliche Baumzerlegung dieses Graphen ist in Abbildung 8 zu sehen.

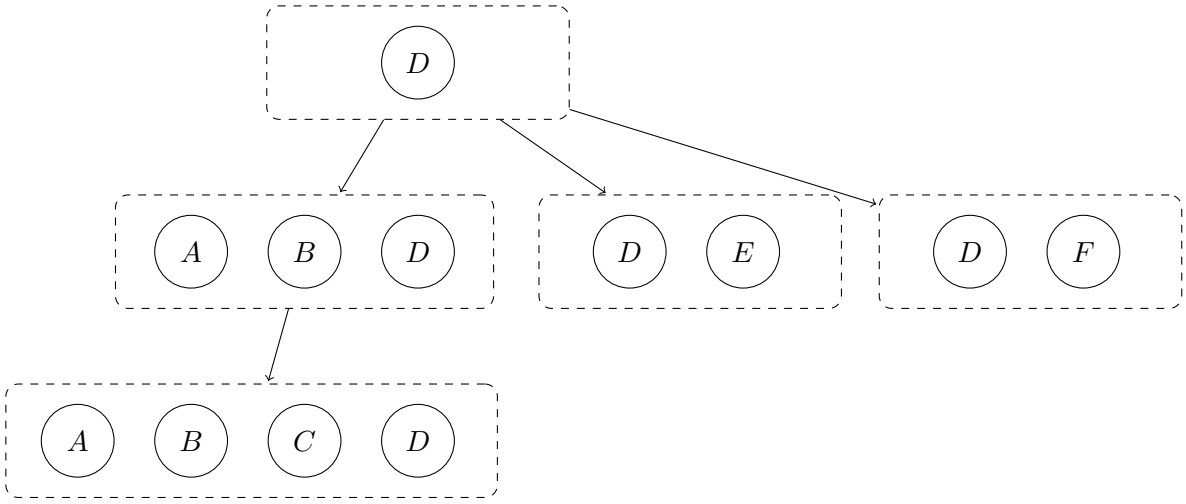


Abbildung 8: Baumzerlegung

Diese Baumzerlegung hat Baumweite 3, denn die größte Tasche ist die mit den Knoten A, B, C und D, diese enthält 4 Knoten und 4-1=3.

Obwohl Baumzerlegungen für viele Anwendungen durchaus interessant sind, benötigen wir hinsichtlich der Zerlegung eines Graphen in Operatoren noch eine lineare Form der Zerlegung. Die im Folgenden definierte Pfadzerlegung liefert genau das.

Definition 3.14 ***Pfadzerlegung und Pfadweite***
Eine Pfadzerlegung eines Graphen G ist eine Baumzerlegung $(\{X_i \mid i \in I\}, T = (I, F))$, so dass T ein Pfad ist. Das heißt, dass T ein Baum ist, so dass für jeden Knoten $x \in I$ gilt $grad(x) \leq 2$. Die Pfadweite dieser Zerlegung ist dann definiert als $max_{i \in I}|X_i| - 1$ und die Pfadweite von G ist die minimale Pfadweite einer Pfadzerlegung von G. Wir kürzen im Folgenden die Pfadweite eines Graphen G mit $pw(G)$ ab, analog ist die Pfadweite einer Pfadzerlegung

P abgekürzt mit pw(P), nach [4]. An einigen Stellen schreiben wir kurz für eine Pfadzerlegung nur die Liste $(X_{i_1}, ..., X_{i_n})$ der Liste der Elemente von X in der Reihenfolge, dass gilt, dass jeweils i_j in T mit i_{j+1} verbunden ist. Pfadzerlegungen werden auf die gleiche Weise wie Baumzerlegungen auch grafisch dargestellt.

Das Finden einer Pfadzerlegung mit kleiner Pfadweite ist Hauptziel der drei Algorithmen die in den nächsten drei Kapiteln vorgestellt werden. Findet man eine optimale Pfadzerlegung, so löst man damit auch das nachfolgend definierte Problem.

Definition 3.15 ***Das Pfadweitenproblem***
Gegeben sei ein Graph G und eine Konstante $k \in \mathbb{N}$. Das Pfadweitenproblem ist die Frage, ob der Graph G eine Pfadweite kleiner oder gleich k hat, also ob $pw(G) \leq k$ gilt.

Leider zeigt uns das Ergebnis des nächsten Satzes, dass ein exakter Algorithmus ohne Einschränkungen an das Problem für das Finden einer optimalen Pfadzerlegung wahrscheinlich nur mit unbrauchbarer Laufzeit zu finden ist.

Satz 3.16 ***Komplexität des Pfadweitenproblems***
Das Pfadweitenproblem ist NP-vollständig. Hält man k fest und nimmt nur den Graphen G als Eingabe, so ist das so modifizierte Pfadweitenproblem in linearer Zeit lösbar, allerdings sind nur Algorithmen mit einer Laufzeit bekannt, die einen konstanten Faktor von 2^k oder größer enthalten.

Wie bei der Baumzerlegung weiter oben wählen wir auch beim Beispiel zur Pfadzerlegung wieder die graphische Darstellungsvariante, die genauso funktioniert wie bei einer Baumzerlegung. Der einzige Unterschied liegt in der linearen Struktur, so dass es keine Verzweigungen in einer Pfadzerlegung gibt. Die gestrichelten Boxen stellen wieder die Taschen dar und die Kreise innerhalb der gestrichelten Boxen die in der jeweiligen Tasche enthaltenen Knoten.

Beispiel 3.17 ***Pfadzerlegung eines Beispielgraphen***
Wir betrachten wieder den Graphen aus dem vorangegangenen Beispiel:

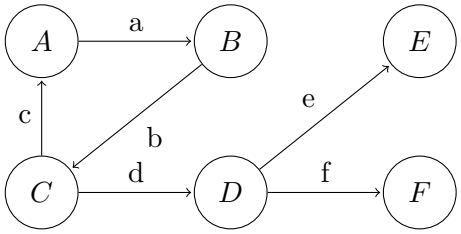


Abbildung 9: Zu zerlegender Graph

Eine mögliche Pfadzerlegung dieses Graphen ist in Abbildung 10 dargestellt. Diese Pfadzerlegung hat Pfadweite 3, denn die größte Tasche ist die

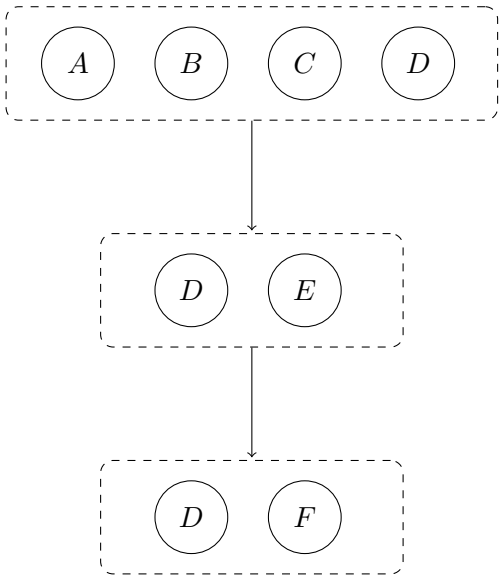


Abbildung 10: Pfadzerlegung

mit den Knoten A, B, C und D, diese enthält 4 Knoten und 4-1=3.

4 Exakter Algorithmus für festgehaltene Pfadweite

Die folgenden Definitionen und Aussagen folgen dem in [5] beschriebenen Weg zum gesuchten Algorithmus, der - bei fixierter Pfadweite k - eine Pfadzerlegung in linearer Zeit ermöglicht. Die Grundidee dieses Algorithmus' ist es, ausgehend von einer vorgegebenen Baumzerlegung eine Pfadzerlegung der Pfadweite k zu bilden. Dafür arbeitet man den Baum von den Blättern zur Wurzel hin ab und bildet für jeden Knoten auf dem Weg eine vollständige Menge von Charakteristiken, die alle Pfadzerlegungen der Pfadweite $\leq k$ repräsentieren. Erhält man auf diese Weise eine nicht-leere vollständige Menge von Charakteristiken für den Wurzelknoten der Baumzerlegung, so gibt es eine Pfadzerlegung der Pfadweite $\leq k$.

Definition 4.1 ***Schöne Baumzerlegung (Nice Tree-Decomposition)*** *Eine Baumzerlegung mit Wurzel, D=(S,T) mit $S = \{X_i \mid i \in I\}$ und T=(I,F), ist eine schöne Baumzerlegung, wenn T ein Binärbaum ist und gilt, dass für jeden Knoten i mit zwei Nachfolgern j und k $X_i = X_j = X_k$ und für jeden Knoten i mit nur einem Nachfolger j gilt, dass $|X_i| = |X_j| \pm 1$ und $X_j \subset X_i$ oder $X_i \subset X_j$. Das bedeutet, dass an allen Stellen an denen der Baum eine Listenstruktur hat, in jedem Schritt von der Wurzel weg entweder ein Knoten des Graphen in die durch den Knoten des Baumes repräsentierte Menge aufgenommen oder aus dieser entfernt wird.* [5]

Definition 4.2 ***Knotentypen***
Im Folgenden benennen wir die verschiedenen Typen von Knoten in einer schönen Baumzerlegung D=(X,T) eines Graphen G:

- ***„Start“*** *Wenn ein Knoten aus T ein Blatt des Baumes ist (das heißt, dass er keine Nachfolger hat), so nennen wir ihn einen Startknoten.*
- ***„Join“*** *Wenn ein Knoten aus T zwei Nachfolger hat, so nennen wir ihn einen Joinknoten, weil sich zwei Zweige des Baumes an diesem Knoten „treffen“.*
- ***„Forget“*** *Wenn ein Knoten i aus T (nur) einen Nachfolger hat, der eine um einen Knoten aus G größere Menge repräsentiert als i, so nennen wir ihn einen Forgetknoten, weil auf dem weiteren Weg vom Blatt zur Wurzel der Knoten, der nicht mehr in der durch i repräsentierten Menge enthalten ist, keine Rolle mehr spielt.*
- ***„Introduce“*** *Wenn ein Knoten i aus T (nur) einen Nachfolger hat, der eine um einen Knoten aus G kleinere Menge repräsentiert als i, so nennen wir ihn einen Introduceknoten, weil der Knoten i im Vergleich zum bisherigen Weg von dem Blatt zur Wurzel einen neuen Knoten aus G in die durch ihn repräsentierte Teilmenge von G aufnimmt.*

Aus der Baumzerlegung aus Beispiel 3.13 wird im folgenden Beispiel zum besseren Verständnis eine schöne Baumzerlegung gebildet.

Beispiel 4.3 ***Schöne Baumzerlegung eines Beispielgraphen***
Wir betrachten wieder den Graphen aus der folgenden Darstellung:

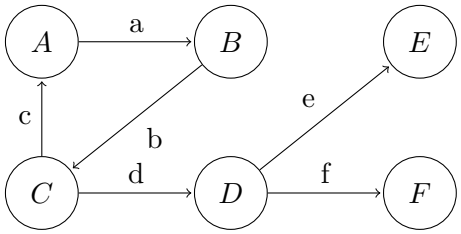


Abbildung 11: Beispielgraph für schöne Baumzerlegung

Eine schöne Baumzerlegung dieses Graphen ist in Abbildung 12 dargestellt. Links neben jeder Tasche ist in Klammern ein Buchstabe zu finden. Dieser

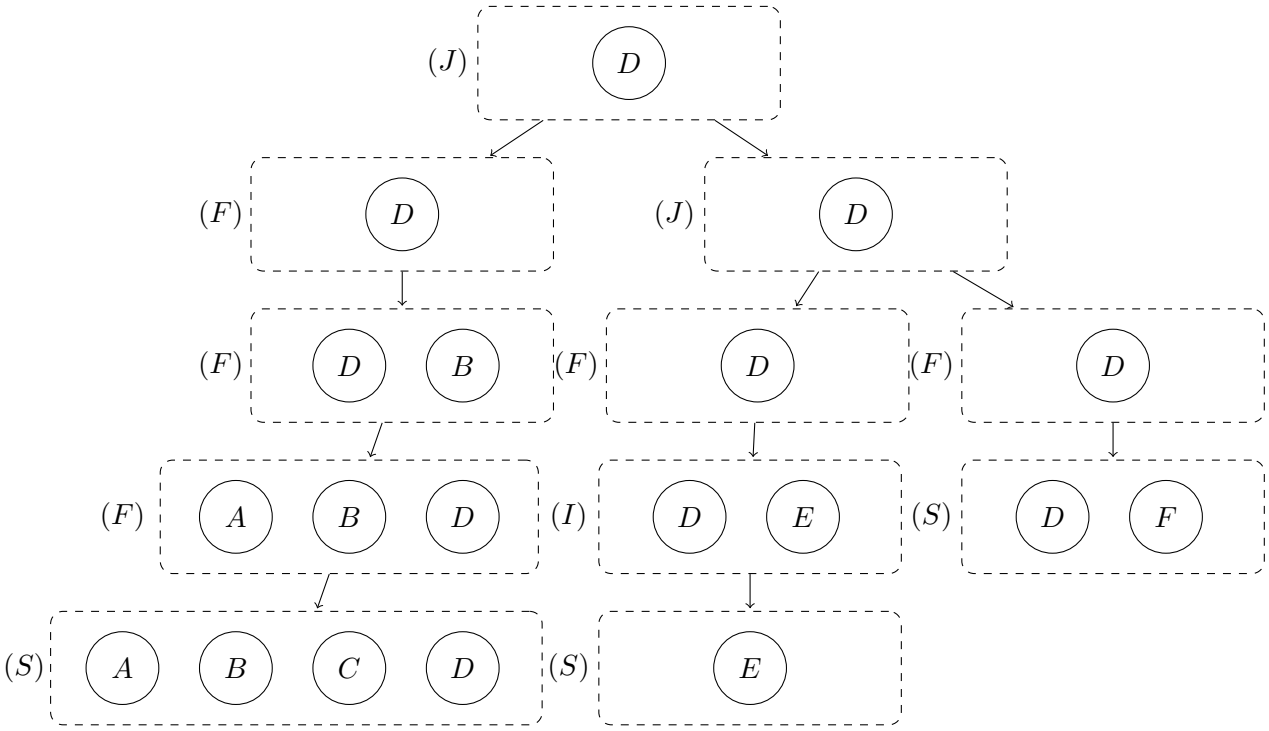


Abbildung 12: Schöne Baumzerlegung

zeigt an, um welchen Typ von Knoten es sich handelt. S steht für Startknoten, J für Joinknoten, I für Introduceknoten und F für Forgetknoten. Diese schöne

Baumzerlegung wurde aus der Beispielsbaumzerlegung weiter oben gewonnen, die Baumweite ist dabei unverändert.

Im Folgenden gehen wir davon aus, dass Startknoten nur einen Knoten enthalten. Hat eine Baumzerlegung einen Startknoten mit mehr als einem Knoten, so kann man diese Eigenschaft einfach herstellen, indem man einen Knoten auswählt, der im Startknoten bleibt und dann für jeden Knoten, der außerdem im Startknoten liegt, einen Introduce-Knoten vor den Startknoten hängt.

Definition 4.4 ***Partielle Pfadzerlegung***
Sei D=(S,T) eine Baumzerlegung des Graphen G mit Wurzel und i ein Knoten in T. Sei weiter T_i der Teilbaum von T mit der Wurzel i und X^i die zugehörige Teilmenge von S gemäß $X^i = \{X_j \mid j \in T_i\}$. Eine Pfadzerlegung des Teilgraphen $G_i = (V_i, E_i)$ von G, $V_i = \{v \in V \mid (\exists t \in T_i \mid v \in S_t)\}$, $E_i = \{(u,v) \in E \mid (\exists t \in T_i \mid u \in S_t \wedge v \in S_t)\}$, nennen wir eine partielle Pfadzerlegung von G ausgehend von i.

Beispiel 4.5 ***Beispiel einer partiellen Pfadzerlegung***
Wir betrachten den rechten Teilbaum des obigen Beispiels für eine schöne Baumzerlegung. Eine partielle Pfadzerlegung zu diesem Teilbaum ist im Folgenden dargestellt:

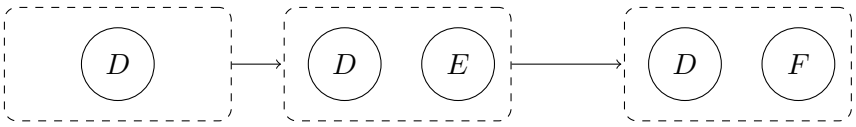


Abbildung 13: Partielle Pfadzerlegung

Definition 4.6 ***Einschränkung und Intervallmodell***
Sei $Y = (Y_1, ...Y_n)$ eine partielle Pfadzerlegung des Graphen G ausgehend von i. Die Einschränkung von Y ist die Pfadzerlegung die sich aus dem Schnitt aller Y_j mit X_i ergibt, also $Y^* = (Y_1 \cap X_i, ...Y_n \cap X_i)$. Wir ermitteln so also nur eine Pfadzerlegung des Teilgraphen von G, der durch die Knoten in X_i gebildet wird. Durch diese Operation entstehen unter Umständen Dopplungen, das heißt Y enthält unter Umständen Paare von Y_k^*, Y_{k+1}^* so dass $Y_k^* = Y_{k+1}^*$. Entfernt man für alle $r \in \{2, ..., n\}$ r-1 Vorkommen aller r-fach vorkommenden Y_k^*, so erhält man das Intervallmodell für Y des Knotens i.* [nach[5]]

Mit Hilfe des Intervallmodells können wir nun die Listenrepräsentation einer partiellen Pfadzerlegung definieren.

Definition 4.7 ***Listenrepräsentation einer partiellen Pfadzerlegung***
Sei $Y = (Y_1, ..., Y_n)$ eine partielle Pfadzerlegung ausgehend von Knoten i des Graphen G und $Z = (Z_1, ..., Z_m)$ das Intervallmodell von i. Dann ist (Z, [Y]) mit $[Y] = (Y^{(1)}, ..., Y^{(m)})$ die Listenrepräsentation von Y. $Y^{(j)}$ sei für $1 \leq j \leq m$ definiert als $\{Y_k \mid 1 \leq k \leq n \wedge Y_k \cap X_i = Z_j\}$.

Beispiel 4.8 ***Intervallmodell und Listenrepräsentation einer partiellen Pfadzerlegung***
Bildet man zu obigem Beispiel der partiellen Pfadzerlegung das Listenmodell, so muss man zunächst jede Tasche auf die Knoten einschränken, die bereits in dem Wurzelknoten des Teilbaums der Baumzerlegung enthalten waren, so erhalten wir zunächst die folgende Pfadzerlegung:

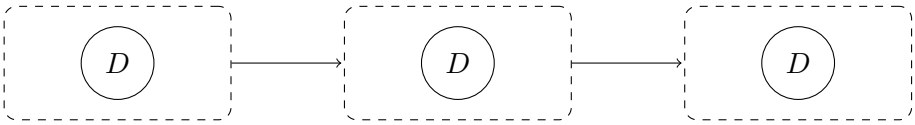


Abbildung 14: Pfadzerlegung nach Schneiden der Tascheninhalte mit X_i

Nun müssen wir doppelt vorkommende Taschen aus der resultierenden Pfadzerlegung streichen. Alle drei Taschen enthalten nur den Knoten D, daher ist das Intervallmodell dann einfach eine einzelne Tasche die nur den Knoten D enthält:

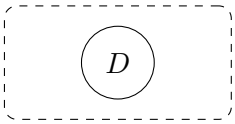


Abbildung 15: Intervallmodell

Die Listenrepräsentation dieser partiellen Pfadzerlegung lässt sich also schreiben als $([\{D\}], [\{D\}, \{D, E\}, \{D, F\}])$, wobei die erste Liste das gerade dargestellte Intervallmodell beschreibt und die zweite Liste die partielle Pfadzerlegung.

Definition 4.9 ***Typische Folge einer Ganzzahl-Folge***
Für eine (endliche) Folge von (nicht-negativen) Ganzzahlen $a = (a_1, a_2, ..., a_n)$ definieren wir l(a)=n als die Länge von a. Wir bilden die (eindeutige) typische Folge $\tau(a)$ als diejenige Folge, die entsteht, wenn wir die folgenden beiden Operationen so lange anwenden, bis keine der beiden Operationen mehr anwendbar ist:

- *Gilt für ein $0 < i < l(a)$ $a_i = a_{i+1}$, so lösche a_{i+1} aus der Folge.*
- *Gibt es ein $0 < i < l(a) - 1$ und ein $i + 1 < j \leq l(a)$ so dass für alle $i < k < j$ entweder $a_i \leq a_k \leq a_j$ oder $a_i \geq a_k \geq a_j$ gilt, so lösche alle a_k aus der Folge.*

Ein wichtiger Satz um einzusehen, dass die Laufzeit des hier vorgestellten Algorithmus' linear für festgehaltene Pfadweite ist, ist der folgende

Satz 4.10 *Es gibt aus der Menge* $\{1, 2, ..., n\}$ *höchstens* $\frac{8}{3} \cdot 2^{2 \cdot n}$ *verschiedene typische Ganzzahlfolgen.*

Beispiel 4.11 ***Typische Folgen***
Die typische Folge von $a := (1, 2, 2, 3, 4, 5, 5, 4, 3, 4)$ *ist* $\tau(a) = (1, 5, 3, 4)$. $b := (1, 6, 2, 5, 3, 4)$ *ist ein Beispiel für eine Folge der Länge 6, deren typische Folge ebenfalls die Länge 6 hat, denn auf b ist keine der beiden Operationen aus der vorhergehenden Definition anwendbar und somit gilt* $b = \tau(b)$.

Um mit Ganzzahlfolgen sinnvoll arbeiten zu können, benötigen wir einen Vergleichsoperator für Ganzzahlfolgen, das $\leq$.

Definition 4.12 $\leq$ ***für Ganzzahlfolgen***
Seien $a = (a_1, a_2, ...a_n)$ *und* $b = (b_1, b_2, ..., b_n)$ *zwei Ganzzahlfolgen gleicher Länge, so gilt* $a \leq b$ *genau dann wenn* $a_i \leq b_i$ *für alle* $i \in 1, 2, ..., n$ *gilt.*

Mit Hilfe dieser Definitionen können wir nun den für diesen Algorithmus zentralen Begriff der Charakteristik definieren.

Definition 4.13 ***Liste, Typische Liste und Charakteristik einer Pfadzerlegung***
Seien Y eine partielle Pfadzerlegung und (Z,[Y]) deren Listenrepräsentation mit $[Y] = (Y^{(1)}, Y^{(2)}, ..., Y^{(n)})$. *Wir definieren die* Liste *von Y als* $y^{(m)} := (|Y_1^{(m)}|, |Y_2^{(m)}|, ..., |Y_{|Y^{(m)}|}^{(m)}|)$ *für alle* $m \in \{1, 2, ..., n\}$. *Die* $y^{(m)}$ *sind offensichtlich Folgen von nichtnegativen Ganzzahlen.* $[y] := (y^{(1)}, y^{(2)}, ..., y^{(m)})$ *ist dann eine Liste von Folgen von Ganzzahlen. Wir wenden den Operator* τ *auf Listen von Folgen von Ganzzahlen an, indem wir ihn komponentenweise auf die Elemente der Liste anwenden, es ist also* $\tau([y]) = (\tau(y^{(1)}), \tau(y^{(2)}), ..., \tau(y^{(m)}))$. $\tau([y])$ *nennen wir dann die* typische Liste *von Y und die* Charakteristik *von Y ist dann definiert als* $C(Y) = (Z, r([y]))$.

Beispiel 4.14 ***Liste und Charakteristik einer Beispielspfadzerlegung***
Wir verwenden wieder die partielle Pfadzerlegung aus den vorhergehenden Beispielen, um Beispiele für die Begriffe Liste und Charakteristik einer Pfadzerlegung zu geben. Die Liste der Pfadzerlegung ist $[(1, 2, 2)]$ *und mit* $\tau([(1, 2, 2)]) = [(1, 2)]$ *folgt, dass die Charakteristik der Pfadzerlegung die folgende ist:* $([\{D\}], [(1, 2)])$.

Definition 4.15 ***Erweiterung einer Ganzzahlfolge und einer Liste von Ganzzahlfolgen***
Eine Erweiterung einer Ganzzahlfolge

$$a = (a_1, a_2, ..., a_n)$$

ist jede Folge, die durch das beliebig häufige Anwenden der folgenden Operation aus a entsteht: Für ein $i \in \{1, 2, ..., l(a)\}$ *ersetze* a_i *in a durch* a_i, a_i. *Das heißt, dass jede Folge, die aus a entsteht, indem man jedes Element von a beliebig oft in der Folge hintereinander setzt, eine Erweiterung von a ist. Die Menge aller Erweiterungen von a bezeichnen wir als E(a). Für eine Liste von Ganzzahlfolgen*

$$[z] = (z_1, z_2, ..., z_p)$$

definieren wir wieder

$$E([z]) = (E(z_1), E(z_2), ..., E(z_p)).$$

Erweiterungen von Ganzzahlfolgen ermöglichen es uns nun, eine Äquivalenzrelation für partielle Pfadzerlegungen anzugeben.

Definition 4.16 ***Äquivalenz von partiellen Pfadzerlegungen***
Seien Y und Z partielle Pfadzerlegungen die von dem gleichen Knoten ausgehen und mit dem gleichen Intervallmodell. $[y], [z]$ *seien die zugehörigen Listen von Y und Z. Es gilt genau dann* $Y \prec Z$, *wenn ein*

$$[y^*] = (y_1^*, y_2^*, ..., y_r^*) \in E([y])$$

und ein

$$[z^*] = (z_1^*, z_2^*, ..., z_r^*) \in E([z])$$

existieren, so dass

$$y_i^* \leq z_i^* \ \textit{für alle}\ i \in \{1, 2, ..., r\}.$$

Gilt $Y \prec Z$ *und* $Z \prec Y$, *so nennen wir Y und Z äquivalent und schreiben* $Y \equiv Z$

Der Algorithmus, der in diesem Abschnitt vorgestellt werden soll, basiert darauf, Mengen von Charakteristiken zu bilden. Diese Mengen nennen wir vollständige Mengen von Charakteristiken, wenn sie der folgenden Definition genügen.

Definition 4.17 ***Vollständige Menge von Charakteristiken***
Eine Menge von Charakteristiken partieller Pfadzerlegungen ausgehend von einem Knoten i mit einer Pfadweite $\leq k$, *FS(i), nennen wir vollständige Menge von Charakteristiken, wenn für jede partielle Pfadzerlegung Y ausgehend von i mit Pfadweite* $\leq k$ *eine Pfadzerlegung Y' mit* $Y' \prec Y$ *existiert, so dass die Charakteristik von Y' in FS(i) ist.*

Diese vollständigen Mengen von Charakteristiken haben einige günstige Eigenschaften, so gilt, dass eine vollständige Menge von Charakteristiken an der Wurzel einer Baumzerlegung von G genau dann nicht leer ist, wenn die Pfadweite von G höchstens k ist. Daher werden wir im Folgenden, weiter [5] folgend, für jeden Typ von Knoten einen Algorithmus angeben, um die zugehörige vollständige Menge von Charakteristiken zu ermitteln. Beweise der Korrektheit dieser Algorithmen finden sich in [5], werden hier aber nicht angegeben. Dabei gehen wir von einer schönen Baumzerlegung aus. X_i bezeichnet die in der Baumzerlegung dem (Baum-) Knoten i zugeordnete Menge von Knoten des zu zerlegenden Graphen G. Zudem ist im Folgenden p jeweils derjenige Knoten der Baumzerlegung, den wir aktuell untersuchen. Die Algorithmen werden von den Blättern des Baums zur Wurzel hin ausgeführt, so dass man bei der Berechnung von $FS(p)$ die $FS(q)$ aller direkten oder indirekten Nachfolgerknoten q verwenden kann.

Algorithmus 4.18 ***Vollständige Menge für einen Startknoten***
Da es sich um eine schöne Baumzerlegung handelt, ist in X_p nur ein Knoten, nennen wir diesen v. Dann gibt es für G_p nur vier mögliche minimale Pfadzerlegungen: $(\emptyset, \{v\}, \emptyset), (\emptyset, \{v\}), (\{v\}, \emptyset), (\{v\})$. Wir bilden die Charakteristiken dieser vier Möglichkeiten und die Menge, die diese vier Charakteristiken enthält, ist die vollständige Menge von Charakteristiken für diesen Knoten.

Um den Algorithmus für die vollständige Menge für einen Joinknoten anzugeben, müssen wir zunächst definieren, wie man zwei Ganzzahlfolgen addiert.

Definition 4.19 ***Ringsumme zweier Ganzzahlfolgen***
Sind

$$x = (x_1, x_2, ..., x_n) \text{ und } y = (y_1, y_2, ..., y_n)$$

zwei Ganzzahlfolgen gleicher Länge (also l(x)=l(y)) dann ist

$$x + y = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n).$$

Für zwei Ganzzahlfolgen a und b definieren wir die Ringsumme der beiden Folgen als

$$a \oplus b = \{a^* + b^* \mid a^* \in E(a) \wedge b^* \in E(b) \wedge l(a^*) = l(b^*)\}.$$

Algorithmus 4.20 ***Vollständige Menge für einen Joinknoten***
Seien q und r die beiden Nachfolger von p in der Baumzerlegung mit den vollständigen Mengen FS(q), FS(r). Wiederum wegen der Eigenschaft der Baumzerlegung, eine schöne Baumzerlegung zu sein, gilt $X_q = X_r = X_p$. Wir berechnen FS(p) indem wir die folgende Menge ermitteln:

$$FS(p) = \{(Z, \tau([c]) \mid (Z, \tau([a]) \in FS(q) \wedge (Z, \tau([b]) \in FS(r) \\ \wedge [c] \in [a^*] \oplus \tau([b]) \wedge max([c]) \leq k + 1\}.$$

Dabei ist die Liste $[a^*]$ *für die Charakteristik*

$$(Z, \tau([a])), \tau([a]) = (\tau(a^{(1)}), \tau(a^{(2)}), ..., \tau(a^{(n)}))$$

definiert als

$$[a^*] = (\tau(a^{(1)}) - |Z_1|, \tau(a^{(2)}) - |Z_2|, ..., \tau(a^{(n)}) - |Z_n|).$$

Auch wenn der Begriff der Konkatenation selbsterklärend ist, wird er nachfolgend der Vollständigkeit halber definiert.

Definition 4.21 ***Konkatenation von Ganzzahlfolgen***
Seien $a = (a_1, a_2, ..., a_n)$ *und* $b = (b_1, b_2, ..., b_m)$ *Ganzzahlfolgen. Die Konkatenation von a und b ist definiert als*

$$a \circ b = (a_1, a_2, ..., a_n, b_1, b_2, ..., b_m).$$

Algorithmus 4.22 ***Vollständige Menge für einen Forget-Knoten***
Sei also p ein Forget-Knoten mit Kindknoten q. Da es Knoten in einer schönen Baumzerlegung sind, unterscheiden sich X_p *und* X_q *nur um einen Knoten x, der in* X_q *enthalten ist, aber in* X_p *nicht. Man berechnet FS(p) aus FS(q) indem man für jedes*

$$(Z, \tau([y])) \in FS(q) \text{ mit } Z = (Z_1, Z_2, ..., Z_n)$$

wie folgt verfährt:
Es gibt ein $1 \leq i \leq n$ *für das gilt* $x \in Z_i$ *und, sofern i nicht 1 ist,* $x \notin Z_{i-1}$. *Auch gibt es ein* $1 \leq j \leq n$ *für das gilt* $x \in Z_j$ *und, sofern j nicht n ist,* $x \notin Z_{j+1}$. *Wir bilden*

$$Z' = (Z_1 \setminus \{x\}, Z_2 \setminus \{x\}, ..., Z_n \setminus \{x\})$$

und unterscheiden nun vier Fälle:

- $Z_i \setminus \{x\} \neq Z_{i-1}$ und $Z_j \setminus \{x\} \neq Z_{j+1}$:
  $(Z', \tau([y]))$ *ist Charakteristik von p. Wir fügen* $(Z', \tau([y]))$ *zu FS(p) hinzu.*

- $Z_i \setminus \{x\} = Z_{i-1}$ aber $Z_j \setminus \{x\} \neq Z_{j+1}$:
  Bilde

  $$\tau([y']) = (\tau(y^{(1)}), \tau(y^{(2)}), ..., (y^{(i-2)}), \tau(\tau(y^{(i-1)}) \circ \tau(y^{(i)})), \tau(y^{(i+1)}), ..., \tau(y^{(n)}))$$

  und füge $(Z', \tau([y'])$ *zu FS(p) hinzu.*

- $Z_i \setminus \{x\} \neq Z_{i-1}$ aber $Z_j \setminus \{x\} = Z_{j+1}$:
  Bilde

  $$\tau([y'']) = (\tau(y^{(1)}), \tau(y^{(2)}), ..., (y^{(j-1)}), \tau(\tau(y^{(j)}) \circ \tau(y^{(j+1)})), \tau(y^{(j+2)}), ..., \tau(y^{(n)}))$$

  und füge $(Z', \tau([y''])$ *zu FS(p) hinzu.*

- $Z_i \setminus \{x\} = Z_{i-1}$ und $Z_j \setminus \{x\} = Z_{j+1}$:
  Falls i=j so bilde

  $\tau([y''']) = (\tau(y^{(1)}), \tau(y^{(2)}), ..., (y^{(i-2)}), \tau(\tau(y^{(i-1)}) \circ \tau(y^{(i)}) \circ \tau(y^{(i+1)})), \tau(y^{(i+2)}), ..., \tau(y^{(n)}))$.

  Sonst bilde

  $\tau([y''']) = (\tau(y^{(1)}), \tau(y^{(2)}), ..., (y^{(i-2)}), \tau(\tau(y^{(i-1)}) \circ \tau(y^{(i)})), \tau(y^{(i+1)}), ...,$

  $(y^{(j-1)}), \tau(\tau(y^{(j)}) \circ \tau(y^{(j+1)})), \tau(y^{(j+2)}), ..., \tau(y^{(n)}))$.
  Füge $(Z', \tau([y''']))$ *zu FS(p) hinzu.*

Bodlaender und Kloks nutzen im letzten Schritt, der Erstellung der vollständigen Menge für einen Introduce Knoten, als ersten Schritt eine Berechnung aller minimalen Pfadzerlegungen für einen Subgraphen. Dabei wird nicht näher erklärt, wie diese Liste entstehen soll. Der letzte Schritt des exakten Algorithmus von Bodlaender und Kloks ist deshalb recht kompliziert und hauptverantwortlich für die ungünstige Laufzeit des Algorithmus. Daher wird die Berechnung der *Vollständigen Menge für einen Introduce-Knoten* hier nur skizziert. In [5] kann er ab Seite 20 im Detail gefunden werden. Zunächst wird eine Liste Q aller minimaler Pfadzerlegungen des Subgraphen, der durch X_p gebildet wird, erstellt. Weiter wird eine Menge Q' gebildet, in der alle Paare (Z, Z') sind, so dass Z aus Q ist und Z' aus Z entsteht, indem man den eingeführten Knoten entfernt und dann doppelt vorkommende Taschen entfernt. Weiterhin entfernt man aus Q' alle Paare (Z, Z') für die es keine Charakteristik für den Kindknoten q von p gibt, die Z' als Intervallmodell hat. Die übrig gebliebenen Z bilden die Pfadzerlegungen der vollständigen Menge für einen Introduce-Knoten. Die zugehörigen Charakteristiken ermittelt man mit Hilfe der Charakteristiken des jeweils zugehörigen Z'. Dabei muss zwischen den drei möglichen Fällen unterschieden werden, dass Z' genauso viele Taschen enthält wie Z, Z' eine Tasche weniger enthält als Z und dass Z' zwei Taschen weniger enthält als Z. Wesentlich ist die Überlegung, wieso dies die einzigen möglichen Fälle sind. In irgendeiner Tasche wird der Knoten v, der im Introduce-Knoten eingefügt wird, in die Pfadzerlegung aufgenommen, dann gibt es später in der Pfadzerlegung eine Tasche, bei der der Knoten v wieder entfernt wird, in allen Taschen dazwischen ist v ebenfalls enthalten. Bildet man aus Z nun Z', so könnten die beiden Taschen, bei denen v eingeführt und wieder entfernt wird, erhalten bleiben (wenn beide nicht identisch zu einer anderen Tasche sind), eine der beiden Taschen wird als Duplikat entfernt oder beide Taschen werden als Duplikate entfernt.

Dieser Schritt ist maßgeblich dafür, dass der Algorithmus in der Praxis vergleichsweise schlecht anwendbar ist, denn das Finden aller Paare (Z, Z') in Q' ist problematisch, da kein effizienter Algorithmus zum Finden der Z

bekannt ist. Hinzu kommt, dass die Anzahl der Charakteristika für einen Knoten zwar nur von der Pfadweite abhängt, von dieser allerdings exponentiell. Da man zudem, um diesen Algorithmus überhaupt ausführen zu können, eine gute Baumzerlegung als Eingabe benötigt, lohnt es sich, auch einen Blick auf Approximationsalgorithmen zu werfen. Der hier vorgestellte Algorithmus ist besonders hinsichtlich der theoretischen Laufzeitbetrachtung interessant. So ermöglicht er die Lösung eines NP-vollständigen Problems, das der Pfadzerlegung, in linearer Zeit, wenn man zusätzlich die Pfadweite als Eingabe verwendet. Leider ist der Algorithmus, ungeachtet seiner theoretischen Geschwindigkeit, in der Praxis kaum anwendbar. Die oben angesprochenen Probleme sorgen dafür, dass es sich anbietet, dennoch auch Approximationsalgorithmen zu betrachten.

Hat man schließlich die vollständige Charakteristik für den Wurzelknoten der Baumzerlegung berechnet, so muss man noch aus einer Charakteristik eine Pfadzerlegung gewinnen, das geschieht über den folgende Algorithmus:

Algorithmus 4.23 *Pfadzerlegung zu einer Charakteristik*
Gegeben sei eine Charakteristik einer Tasche p der Baumzerlegung: $c^p = ((Z_j)_{1\leq j\leq m}, \tau(y^{(j)})_{1\leq j\leq m})$. *Der Algorithmus gibt die Pfadzerlegung in der folgenden Repräsentation zurück. Für jede Tasche der Pfadzerlegung i gibt es eine Menge* L_i *der Knoten, die in Tasche i das erste Mal vorkommen und eine Menge* R_i *der Knoten, die in der Tasche i das letzte Mal vorkommen. Für jedes der* Z_j *gibt es einen Zeiger von* Z_j *zur ersten Tasche in der Pfadzerlegung* Y_i *und zur letzten Tasche* $Y_{i'}$ *in der Pfadzerlegung für die* $Z_j = Y_i \cap X_p$ *respektive* $Z_j = Y_{i'} \cap X_p$ *richtig ist.*
Sei schließlich $\tau(y^{(j)}) = (\tau(y_1^{(j)}), ..., \tau(y_{n_j}^{(j)}))$, *so gibt es noch für jedes der* $\tau(y_k^{(j)})$ *einen Zeiger auf die zugehörige Menge* Y_i *mit* $|Y_i| = \tau(y_k^{(j)})$. *Auf dieser Basis kann nun der rekursive Algorithmus beschrieben werden, der aus einer Charakteristik einer Tasche p die zugehörige Pfadzerlegung berechnet. Es wird wieder unterschieden nach Typ der Tasche p:*

- ***Startknoten****: Es gibt nur einen Knoten im zu zerlegenden Graphen, wähle also die zur Charakteristik passende Pfadzerlegung aus den vier möglichen Pfadzerlegungen aus.*

- ***Joinknoten****: Man berechnet zunächst die Pfadzerlegungen für die beiden Kindknoten q und r. Im Folgenden seien* $c = (c^1, ..., c^{n_c})$ *die Liste zur partiellen Pfadzerlegung von p,* $a = (a^1, ..., a^{n_a})$ *die Liste zur partiellen Pfadzerlegung von q und* $b = (b^1, ..., b^{n_b})$ *die Liste zur partiellen Pfadzerlegung von r. Weiter sei die Liste* $[a']$ *für die Charakteristik*

$$(Z, \tau([a])), \tau([a]) = (\tau(a^{(1)}), \tau(a^{(2)}), ..., \tau(a^{(n)}))$$

definiert als

$$[a^{'}] = (a^{(1)} - |Z_1|, a^{(2)} - |Z_2|, ..., a^{(n)} - |Z_n|).$$

Es gilt nach dem Algorithmus zur Berechnung der Vollständigen Menge für einen Joinknoten $[c] = [a^*] \oplus \tau([b])$. *Es gibt also eine Liste* $[c^0] \in E(c)$ *mit* $[c^0] = [a'] \oplus [b]$. *Seien nun* $[a^0]$ *und* $[b^0]$ *Erweiterungen von* $[a']$ *und* $[b]$ *so dass* $[c^0] = [a^0] + [b^0]$. *Bezeichnet nun A die Pfadzerlegung der Tasche q und B die Pfadzerlegung der Tasche r, so ermitteln wir schließlich die Pfadzerlegung zur Tasche p indem wir*

- *In A alle Taschen n-fach duplizieren, deren zugehörige Zahl aus* $[a]$ *bei der Berechnung von* $[a^0]$ *n-fach dupliziert werden. Die resultierende Pfadzerlegung nennen wir* A^0.
- *In B alle Taschen n-fach duplizieren, deren zugehörige Zahl aus* $[b]$ *bei der Berechnung von* $[b^0]$ *n-fach dupliziert werden. Die resultierende Pfadzerlegung nennen wir* B^0.
- *Die gesuchte Pfadzerlegung erhalten wir, indem wir (für i gleich 1 bis zur Anzahl der Taschen in* A^0*) jeweils die i-te Tasche der Zerlegung* A^0 *vereinigen mit der i-ten Tasche der Zerlegung* B^0 *und als i-te Tasche der Pfadzerlegung C hinzufügen.*

- ***Forgetknoten****: Man berechnet rekursiv die Pfadzerlegung für die Kindtasche q, die auch eine Pfadzerlegung für die Tasche p ist und passt die Zeiger entsprechend an.*

- ***Introduceknoten****: Für Introduceknoten haben Bodlaender und Kloks keinen Algorithmus angegeben. Es bietet sich beispielsweise folgendes Vorgehen an: Man berechnet zunächst die Pfadzerlegung für die Kindtasche q. Jetzt müssen die Taschen gefunden werden, in denen der neue Knoten in der Pfadzerlegung eingeführt beziehungsweise entfernt werden muss. Gemäß der Beschreibung des Algorithmus' zur Berechnung der vollständigen Menge für einen Introduceknoten können dadurch höchstens zwei neue Taschen zur Pfadzerlegung hinzugefügt werden, entfernt werden muss keine Tasche. Vergleichen der Intervallmodelle für die Kindtasche und für p ermöglicht, die möglichen Taschen, in denen der Knoten hinzugefügt werden muss, einzuschränken. Für die so erhaltenen Pfadzerlegungen kann man dann die Charakteristik berechnen und vergleichen, ob sie der gegebenen Charakteristik entsprechen.*

Abschließend sei noch einmal die Idee des Algorithmus' zusammengefasst. Ausgehend von einer schönen Baumzerlegung und einer vorgegebenen Pfadweite k berechnet man aufsteigend von den Blättern zur Wurzel die vollständige Menge von Charakteristiken für jede Tasche der Baumzerlegung. Ist die vollständige Menge von Charakteristiken für den Wurzelknoten der

Baumzerlegung nicht leer, so bestimmt man zu einer der darin enthaltenen Charakteristiken eine Pfadzerlegung der Pfadweite $\leq k$, andernfalls weiß man, dass es keine Pfadzerlegung der Pfadweite $\leq k$ gibt.

5 Approximationsalgorithmus von Bodlaender, Gilbert, Hafsteinsson und Kloks

In diesem Abschnitt soll ein Approximationsalgorithmus für das Pfadweitenproblem von Bodlaender, Gilbert, Hafsteinsson und Kloks vorgestellt werden. Dieser Algorithmus basiert stark auf einem Ergebnis von Leighton und Rao, daher wird hier zunächst deren wesentlicher Algorithmus vorgestellt, bevor dieser genutzt wird, um eine Baumzerlegung eines Graphen zu gewinnen. Mit Hilfe eines weiteren approximativen Algorithmus' kann diese dann in eine Pfadzerlegung umgesetzt werden. Der Algorithmus basiert darauf, den Graphen möglichst gleichmäßig zu zerschneiden und die sich so ergebenden Komponenten dann weiter zu zerlegen. Um diese Schnitte, so genannte b-balancierte Knotenschnitte, zu berechnen, verwenden Bodlaender, Gilbert, Hafsteinsson und Kloks Approximationalgorithmen von Leighton und Rao, [13]. Daher wird zunächst beschrieben, wie man nach Leighton und Rao, basierend auf einem Flussproblem mit mehreren Gütern b-balancierte Knotenschnitte berechnet, bevor im anschließenden Abschnitt der eigentliche Approximationsalgorithmus für die Baumzerlegung angegeben wird.

5.1 UMFP, Min Cut und balancierte Schnitte: Approximationsalgorithmen von Leighton und Rao

Die nachfolgenden Definitionen, Sätze und Algorithmen basieren, wenn nicht eine andere Quelle angegeben ist, auf [13]. Die Algorithmen von Leighton und Rao basieren auf einem Approximationsalgorithmus für ein verallgemeinertes Maximalflussproblem. Wenn nicht anders erklärt, bezeichnet G einen Graphen und n die Anzahl der Knoten in diesem Graphen.

Definition 5.1 ***Netzwerk***
Ein Netzwerk (G,C,s,t) ist ein (ungerichteter) Graph G mit einem ausgezeichneten Quellknoten s und einem ausgezeichneten Senkeknoten t sowie einer Kapazitätsfunktion C. C weist jeder Kante in G eine natürliche Zahl, ihre Kapazität, zu. [19]

Definition 5.2 ***Fluss in einem Netzwerk***
Ein Fluss in einem Netzwerk (G,C,s,t) ist eine Funktion F, die jeder Kante e in G einen Wert zwischen 0 und C(e) zuweist. Dabei gilt für jeden Knoten v in G, dass die Summe der ausgehenden Kapazitäten genau gleich der Summe der eingehenden Kapazitäten sein muss. Der Wert eines Flusses ist schließlich die Summe der eingehenden Kapazitäten in den Senkeknoten. [19]

Definition 5.3 ***Min-Cut eines Netzwerkes***
Der Min-Cut eines Netzwerkes N ist die minimale Anzahl an Kapazitäten,

die man aus N entfernen muss, damit der maximale Fluss in N genau den Wert 0 hat. Ford und Fulkerson haben bewiesen, dass für derartige Netzwerke der Min-Cut genau der Wert des maximalen Flusses ist. [19]

Die Intuition dieser Fragestellung ist es, den Fluss von Wasser durch Schleusen darzustellen. Auch den wirtschaftlichen Fluss von Gütern über Vertriebswege kann man auf diese Weise modellieren. Letztere Interpretation führt uns zu einem verallgemeinerten Flussproblem, das den Hauptgegenstand der Arbeit von Leighton und Rao darstellt:

Definition 5.4 ***Flussproblem mit mehreren Gütern***
Wie beim klassischen Flussproblem betrachtet man auch beim Flussproblem mit mehreren Gütern einen ungerichteten Graphen G=(V,E) mit einer Kapazitäsfunktion C, die jeder Kante eine natürliche Zahl, ihre Kazapzität, zuweist. In einem Flussproblem mit mehreren Gütern gibt es $k \geq 1$ verschiedene Güter, denen jeweils eine Quelle s_i, eine Senke t_i und ein Bedarf D_i zugeordnet sind. Das Ziel bei einem so definierten Flussproblem ist es, für jedes $1 \leq i \leq k$ mindestens D_i Einheiten des Gutes i von s_i zu t_i fließen zu lassen, ohne an einer Kante mehr Güter fließen zu lassen, als die Kapazität zulässt.
Wiederum muss für jeden Knoten gelten, dass die Menge (und auch die Art) von Gütern, die in den Knoten hineinfließt, genau gleich der Menge (und der Art) von Gütern sein muss, die aus dem Knoten herausfließen. Alternativ kann man auch den Fluss in einem solchen Netzwerk maximieren, wenn man eine Funktion, abhängig vom Fluss der verschiedenen Güter, angibt, die maximiert werden soll. Diese Funktion soll beispielsweise sicherstellen, dass kein Gut beim Maximieren des Gesamtflusses „zu kurz kommt“.

Definition 5.5 ***Minimaler Schnitt in einem Flussproblem mit mehreren Gütern***
Der minimale Schnitt ist definiert als

$$S = \min_{U \subset V} \frac{C(U, V \setminus U)}{D(U, V \setminus U)}.$$

Dabei ist

$$C(U, V \setminus U) = \sum_{e \in \langle U, V \setminus U \rangle} C(e),$$

also die Gesamtkapazität der Kanten zwischen U und der Komplementmenge von U und

$$D(U, V \setminus U) = \sum_{i | (s_i \in U \wedge t_i \in V \setminus U) \vee (s_i \in V \setminus U \wedge t_i \in U)} D_i,$$

also die Summe der Bedarfe von Gütern, deren Quellen in einer der Mengen U und $V \setminus U$ und deren Senke in der jeweils anderen dieser Mengen enthalten

ist. Der Min-Cut ist eine obere Schranke für den Max-Flow, da die Kapazität der Kanten zwischen U und $V \setminus U$ obere Schranke für den Fluss zwischen diesen beiden Teilen des Graphen ist.

Leighton und Rao haben einen speziellen Typ von Flussproblemen mit mehreren Gütern betrachtet, das *gleichförmige Flussproblem mit mehreren Gütern* (englisch: Uniform Multicommodity Flow Problem, kurz *UMFP*). Bei diesem Flussproblem gibt es ein Gut für jedes Paar von Knoten im Graph und jeder Bedarf D_i wird als 1 gesetzt. Der Bedarf über einem Schnitt lässt sich in diesem Fall etwas einfacher ausdrücken und zwar als $D(U, V \setminus U) = |U| \cdot |V \setminus U|$. Für diesen Typ von Graphen haben Leighton und Rao gezeigt, dass der Min-Cut höchstens $O(log(n))$ größer ist als der maximale Fluss. Die mit Hilfe von bekannten Algorithmen für das Max-Flow Problem gewonnenen Approximationen für den Min-Cut können, wie wir noch sehen werden, gewinnbringend für Bodlaenders Approximations-Algorithmus angewandt werden.

Im Folgenden bezeichnet für $U \subset V$

$$\frac{C(U, V \setminus U)}{D(U, V \setminus U)} = \frac{C(U, V \setminus U)}{|U| \cdot |V \setminus U|}$$

die relativen Kosten eines Schnitts - der Schnitt für den dieser Wert minimal ist, ist, gemäß obiger Definition, der minimale Schnitt.

Definition 5.6 ***Das duale Problem des UMFP***
Das duale Problem zum UMFP ist das Problem, jeder Kante e des zusammenhängenden *Graphen G=(V,E) ein Gewicht d(e) zuzuweisen, so dass* $\sum_{u,v \in V} d(u, v) \geq 1$ *gilt und das Gesamtgewicht* $W := \sum_{e \in E} C(e)d(e)$ *minimiert wird. Anschaulich handelt es sich, wenn man das Gewicht d als Entfernung interpretiert, also um das Problem, die kleinstmögliche Menge an Gesamtgewicht auf die Kanten zu verteilen, so dass die Summe der Abstände zwischen den Quellen und Senken eine untere Grenze nicht unterschreitet.*

Bei diesem Problem handelt es sich um ein Problem der linearen Optimierung. Bekannte Verfahren, um diese Art von Problemen zu lösen, sind beispielsweise das Simplex-Verfahren oder das Innere-Punkte-Verfahren. Generell ist es möglich, lineare Optimierungsprobleme in Polynomzeit zu lösen, in der Praxis hat sich allerdings trotz im Worst Case exponentieller Laufzeit das Simplex Verfahren als schnellstes erwiesen. In der Implementierung wurde das GNU Linear Programming Kit (GLPK) verwendet, um das Optimierungsproblem zu lösen.

$\sum_{u,v \in V} d(u, v) \geq 1$ ist allerdings so nicht als lineare Bedingung formuliert. Iri [10] hat eine Formulierung dieser Bedingung als lineares Gleichungssystem

gefunden. Man führt für jedes Gut i und jeden Knoten v eine Variable $z_{i,v}$ ein. Diese Variablen sollen den Abstand des Knotens v von dem Zielknoten des i-ten Gutes darstellen. Dann können wir $\sum_{u,v \in V} d(u,v) \geq 1$ ausdrücken als $\sum_{u,v \in V} z_{i,s_i} - z_{i,t_i} \geq 1$. Zusätzlich benötigt man dann die Nebenbedingungen:

$z_{i,s_i} \geq 0$
$z_{i,t_i} = 0$

für alle Güter i. s_i ist dabei die Quelle von Gut i und t_i der Senkeknoten von Gut i.
Außerdem muss für jede Kante (u,v) gelten:

$d((u,v)) + z_{i,v} \geq z_{i,u}$ und
$d((u,v)) + z_{i,u} \geq z_{i,v}$.

Umstellen liefert dann die Ungleichungen:

$d((u,v)) + z_{i,v} - z_{i,u} \geq 0$ und
$d((u,v)) + z_{i,u} - z_{i,v} \geq 0$.

Es sei aber noch einmal darauf hingewiesen, dass es wichtig ist, dass der Graph tatsächlich zusammenhängend ist, sonst ergibt dieses Optimierungsproblem kein sinnvolles Ergebnis für das UMFP.

Beispiel 5.7 ***Lösung einer Instanz des dualen Problems des UMFP*** *Betrachten wir ein weiteres Mal unser erstes Beispiel für einen Graphen, zur Erinnerung sei dieser im Folgenden noch einmal dargestellt:*

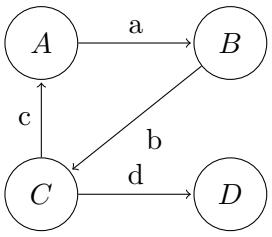


Abbildung 16: Beispielgraph

Suchen wir mit Hilfe eines Programms zur linearen Optimierung, in diesem Fall dem GLPK, eine Lösung für dieses Problem, so erhalten wir die Lösung, dass die Kante d mit dem Gewicht $\frac{1}{3}$ belegt wird und alle anderen Kanten mit dem Gewicht 0. Dazu muss das folgende lineare Optimierungsproblem gelöst werden:

Die Knoten werden im Folgenden alphabetisch von 1 bis 4 durchnummeriert, gleichfalls die Kanten. Es werden 6 Güter betrachtet. Das erste Gut soll von Knoten A zu Knoten B fließen, das zweite Gut von Knoten A zu Knoten C, das dritte Gut von Knoten A zu Knoten D, das vierte Gut von Knoten B zu Knoten C, das fünfte Gut von Knoten B zu Knoten D und das sechste Gut von Knoten C zu Knoten D.
Zu minimieren ist die Summe $y_1 + y_2 + y_3 + y_4$, *dabei sind die folgenden Bedingungen einzuhalten:*
$Z_{1,2} = Z_{2,3} = Z_{3,4} = Z_{4,3} = Z_{5,4} = Z_{6,4} = 0$ *um die* Z_{i,t_i} *auf 0 zu setzen.*
$Z_{1,1} \geq 0$, $Z_{2,1} \geq 0$, $Z_{3,1} \geq 0$, $Z_{4,2} \geq 0$, $Z_{5,2} \geq 0$ *und* $Z_{6,3} \geq 0$, *um sicherzustellen dass die* Z_{i,s_i} *nicht negativ sind.*
Schließlich folgen eine ganze Menge Bedingungen, die die Bedingungen:
$d((u,v)) + z_{i,v} - z_{i,u} \geq 0$ *und* $d((u,v)) + z_{i,u} - z_{i,v} \geq 0$
sicherstellen:
$y_1 - Z_{1,2} + Z_{1,1} \geq 0$ *und* $y_1 + Z_{1,2} - Z_{1,1} \geq 0$
$y_1 - Z_{2,2} + Z_{2,1} \geq 0$ *und* $y_1 + Z_{2,2} - Z_{2,1} \geq 0$
$y_1 - Z_{3,2} + Z_{3,1} \geq 0$ *und* $y_1 + Z_{3,2} - Z_{3,1} \geq 0$
$y_1 - Z_{4,2} + Z_{4,1} \geq 0$ *und* $y_1 + Z_{4,2} - Z_{4,1} \geq 0$
$y_1 - Z_{5,2} + Z_{5,1} \geq 0$ *und* $y_1 + Z_{5,2} - Z_{5,1} \geq 0$
$y_1 - Z_{6,2} + Z_{6,1} \geq 0$ *und* $y_1 + Z_{6,2} - Z_{6,1} \geq 0$
$y_2 - Z_{1,3} + Z_{1,2} \geq 0$ *und* $y_2 + Z_{1,3} - Z_{1,2} \geq 0$
$y_2 - Z_{2,3} + Z_{2,2} \geq 0$ *und* $y_2 + Z_{2,3} - Z_{2,2} \geq 0$
$y_2 - Z_{3,3} + Z_{3,2} \geq 0$ *und* $y_2 + Z_{3,3} - Z_{3,2} \geq 0$
$y_2 - Z_{4,3} + Z_{4,2} \geq 0$ *und* $y_2 + Z_{4,3} - Z_{4,2} \geq 0$
$y_2 - Z_{5,3} + Z_{5,2} \geq 0$ *und* $y_2 + Z_{5,3} - Z_{5,2} \geq 0$
$y_2 - Z_{6,3} + Z_{6,2} \geq 0$ *und* $y_2 + Z_{6,3} - Z_{6,2} \geq 0$
$y_3 - Z_{1,1} + Z_{1,3} \geq 0$ *und* $y_3 + Z_{1,1} - Z_{1,3} \geq 0$
$y_3 - Z_{2,1} + Z_{2,3} \geq 0$ *und* $y_3 + Z_{2,1} - Z_{2,3} \geq 0$
$y_3 - Z_{3,1} + Z_{3,3} \geq 0$ *und* $y_3 + Z_{3,1} - Z_{3,3} \geq 0$
$y_3 - Z_{4,1} + Z_{4,3} \geq 0$ *und* $y_3 + Z_{4,1} - Z_{4,3} \geq 0$
$y_3 - Z_{5,1} + Z_{5,3} \geq 0$ *und* $y_3 + Z_{5,1} - Z_{5,3} \geq 0$
$y_3 - Z_{6,1} + Z_{6,3} \geq 0$ *und* $y_3 + Z_{6,1} - Z_{6,3} \geq 0$
$y_4 - Z_{1,4} + Z_{1,3} \geq 0$ *und* $y_4 + Z_{1,4} - Z_{1,3} \geq 0$
$y_4 - Z_{2,4} + Z_{2,3} \geq 0$ *und* $y_4 + Z_{2,4} - Z_{2,3} \geq 0$
$y_4 - Z_{3,4} + Z_{3,3} \geq 0$ *und* $y_4 + Z_{3,4} - Z_{3,3} \geq 0$
$y_4 - Z_{4,4} + Z_{4,3} \geq 0$ *und* $y_4 + Z_{4,4} - Z_{4,3} \geq 0$
$y_4 - Z_{5,4} + Z_{5,3} \geq 0$ *und* $y_4 + Z_{5,4} - Z_{5,3} \geq 0$
$y_4 - Z_{6,4} + Z_{6,3} \geq 0$ *und* $y_4 + Z_{6,4} - Z_{6,3} \geq 0$

An diesem Beispiel sieht man bereits einen zu erwartenden Flaschenhals für die Laufzeit des Approximationsalgorithmus' von Bodlaender. Schon bei relativ kleinen Graphen sind ziemlich viele Bedingungen zu beachten und die Anzahl der Variablen ist in der Größenordnung $O(n^3)$. Bei großen

Graphen dauert die Lösung des Optimierungsproblems selbst mit sehr guten Implementierungen einige Zeit.

Die Approximation nach Leighton und Rao ergibt sich nun mit Hilfe der folgenden Algorithmen. Dabei wird im Folgenden angenommen, dass man obiges Optimierungsproblem bereits gelöst hat und jeder Kante ein entsprechendes Gewicht zugeordnet hat. Zunächst wird der grundlegende Algorithmus vorgestellt und anschließend erklärt, wie die einzelnen Schritte auszuführen sind:

Algorithmus 5.8 ***Approximation des Min-Cut auf Basis eines UMFP***
Es ist möglich, den Graphen so zu partitionieren (1), dass er in Komponenten zerfällt, deren Radius maximal $\frac{1}{2n^2}$ beträgt. Das heißt, dass zwei Knoten in einer Komponente maximal den Abstand $\frac{1}{n^2}$ haben (das entspricht der Vorstellung eines geometrischen Kreises, in dem zwei Punkte maximal den euklidischen Abstand von 2 Mal dem Radius haben). Zwei Knoten a und b haben den Abstand i, wenn der kürzeste Pfad - gemessen als Summe der Gewichte der Kanten auf dem Pfad - von a nach b die Länge i hat. Daraus lässt sich ein Schnitt mit relativen Kosten $O(W \cdot log(n))$ gewinnen oder eine Komponente mit dem Radius $\frac{1}{2n^2}$ finden, die mindestens $\frac{2}{3}$ der Knoten enthält (2). Im letzten Fall kann man daraus einen Schnitt mit relativen Kosten O(W) finden (3).

Diese Aussagen werden in dieser Arbeit nicht bewiesen, die nötigen Algorithmen, die für die Schritte (1), (2) und (3) notwendig sind, werden aber im Folgenden dargestellt. Der erste wichtige Schritt ist (1), das Zerlegen eines Graphen in Komponenten mit beschränktem Radius.

Algorithmus 5.9 ***Partitionierungsalgorithmus***
Ziel ist es, einen gegebenen Graphen G=(V,E) so in Komponenten zu zerlegen, dass jede einzelne Komponente maximal den Radius Δ (oder konkret im obigen Algorithmus Radius $\frac{1}{2n^2}$) hat und dass die Summe der Kapazitäten der Kanten, die die sich ergebenden Komponenten verbinden, maximal $4W \cdot \frac{log(n)}{\Delta}$ beträgt. Im Folgenden sei

$$C = \sum_{e \in E} C(e)$$

die Gesamtkapazität des Graphen G. Falls

$$\Delta \leq 4W \cdot \frac{log(n)}{C}$$

gilt, kann einfach jeder Knoten als einzelne Komponente aufgefasst werden und diese Unterteilung zurückgegeben werden. Andernfalls erstellen wir einen neuen Graphen G^+, der alle Knoten enthält, die G enthält, in dem aber

für jede Kante der Kapazität c ein Pfad von $\lceil C \cdot d(e)/W \rceil$ Kanten mit Gewicht 1 und Kapazität c existiert. Nun kann man die Komponenten in G^+ suchen und daraus die Komponenten in G gewinnen, indem man eine Menge von Knoten aus G genau dann in eine Komponente zusammenfasst, wenn die entsprechenden Knoten aus G^+ in eine Komponente zusammengefasst wurden.
Wähle nun einen Knoten $v \in G^+$ aus, der aus G stammt. Definiere weiter G_i^+ als die Menge aller Knoten in G^+, die einen Abstand von v kleiner gleich i haben. Da ausschließlich Kanten mit dem Gewicht 1 existieren, kann diese Menge über eine einfache Breitensuche gewonnen werden, den etwas allgemeiner nutzbaren Dijkstra-Algorithmus benötigt man daher nicht. Wir wählen die zu v gehörige Komponente nun aus, indem wir die Funktion C_i wie folgt definieren: $C_0 = 2C/n$ und für $i > 0$ sei C_i die Summe der Kapazitäten aus G_i^+. Wähle nun als erste Komponente G_j^+, wobei j definiert sei als kleinstes $i \geq 0$ so dass

$$C_{i+1} < (1+\epsilon) \cdot C_i, \epsilon = W \cdot \frac{log\ n}{\Delta \cdot C}.$$

Entferne G_j^+ aus G^+ und wiederhole dieses Verfahren so lange, bis alle Knoten aus G in einer Komponente untergebracht sind.

Als nächstes betrachten wir Schritt (2). Ziel ist es, einen Schnitt mit relativen Kosten $O(W \cdot log(n))$ zu gewinnen oder eine Komponente mit dem Radius $\frac{1}{2n^2}$ zu finden, die mindestens $\frac{2}{3}$ der Knoten enthält. Nach Anwendung des obigen Algorithmus hat man entweder eine Komponente erhalten, die mindestens $\frac{2}{3}$ aller Knoten enthält oder man teilt die Komponenten so in zwei Mengen auf, dass in jeder der beiden Mengen mindestens ein Drittel der Knoten enthalten ist. Im zweiten Fall haben wir bereits den gewünschten Schnitt gefunden, im ersten Fall sind wir bei Punkt (3) angelangt, dessen Lösung der folgende Algorithmus angibt:

Algorithmus 5.10 ***Zerlegen der Komponente mit 2/3 der Knoten***
Gegeben sei ein Graph G=(V,E) und eine Zerlegung des Graphen in zwei Komponenten mit einem Radius von maximal $\frac{1}{2n^2}$, wobei eine der beiden Komponenten, T, mindestens 2/3 der Knoten aus G enthält. Wir definieren für einen Knoten v in T G^+ und G_i^+ wie in dem Partitionierungsalgorithmus. Weiter sei V_i die Menge der Knoten in G, die auch in G_i^+ enthalten sind. Für den Schnitt $< V_i, V \setminus V_i >$ seien schließlich R_i die relativen Kosten dieses Schnittes. Bestimmt man nun dasjenige i, für das R_i minimal ist, so erhält man einen Schnitt mit den gewünschten Kosten.

Mit Hilfe dieses Approximationsalgorithmus, so haben Leighton und Rao gezeigt, kann man eine ganze Reihe weiterer Probleme gut approximieren. Zwei dieser Probleme, die aufeinander aufbauend gelöst werden können, bilden die Grundlage für den Approximationsalgorithmus von Bodlaender.

Definition 5.11 ***Dünnster Schnitt***
Der dünnste Schnitt eines Graphen G=(V,U) ist im ungerichteten Graphen und ohne Kanten- oder Knotengewichte eine Unterteilung von G in die Mengen U und $U' = V \setminus U$ für die der Quotient aus der Anzahl der Kanten von U nach U' k und dem Produkt aus Anzahl der Knoten in U und in U' minimal ist, also $\frac{k}{|U| \cdot |U'|}$ minimal ist. Dies kann - abermals mit Abweichung O(log(n)) vom Optimum - approximiert werden, indem man Nachfrage und Kapazität für alle Kanten und Güter auf 1 setzt und den Approximationsalgorithmus für das Problem des minimalen Schnitts eines Netzwerkes mit mehreren Gütern anwendet.

Bemerkung 5.12 ***Kantengewichte***
Gibt es Kantengewichte, verändert sich das Problem in der Art, dass der Quotient aus der Summe der Gewichte der Kanten von U nach U' und dem Produkt der Anzahl der Knoten in U und U' minimiert werden muss. Auch diese Verallgemeinerung lässt sich mit einer Abweichung vom Optimum logarithmisch in der Anzahl der Knoten mit Gewicht ungleich 0 approximieren. Dafür setzt man die Kapazität einer Kante als ihr Gewicht und wendet abermals den Algorithmus für das Problem des minimalen Schnitts eines Netzwerkes mit mehreren Gütern an.

Auf dieser Basis kann schließlich eine Approximation desjenigen Problems angegeben werden, das maßgeblich für Bodlaenders Approximationsalgorithmus ist, den b-balancierten Schnitt.

Definition 5.13 ***Balancierter Kantenschnitt***
Sei $b \in]0, 0, 5[$. Ein b-balancierter Kantenschnitt eines Graphen G=(V,E) ist ein Schnitt $< U, V \setminus U >$ für den gilt:

$$b \cdot |V| \leq |U| \leq (1 - b)|V|.$$

Leighton und Rao haben einen Algorithmus angegeben, mit dem man für $b \leq 1/3$ einen b-balancierten Kantenschnitt finden kann, dessen Größe wiederum höchstens logarithmisch vom Optimum abweicht. Auf Basis der bisherigen Algorithmen ist auch dieser Algorithmus relativ einfach.

Algorithmus 5.14 ***Näherung des b-balancierten Kantenschnitts***
Man finde eine Näherung $< U, U' >$ des dünnsten Schnitts mit dem oben genannten Algorithmus und nenne die kleinere der beiden Mengen U und U' im Folgenden U_0, die andere G_1. Auf der größeren Menge rufen wir wieder den Algorithmus für den dünnsten Schnitt auf. Im i-ten Schritt nennen wir jeweils die kleinere Menge des sich ergebenden Schnittes U_i, die größere G_{i+1} und fahren mit der größeren Menge fort, so lange bis ein j gefunden ist, so dass $|G_{j+1}| \leq (1 - b)|G|$. Dann bildet man die Menge $W = \bigcup_{i=0}^{j} U_i$ und gibt den gewünschten Schnitt als $< W, V \setminus W >$ zurück.

Damit haben wir eine gute Approximation des b-balancierten Kantenschnitts erhalten. Daraus kann man (nach [1]) ebenfalls wie folgt einen guten balancierten Knotenschnitt erhalten. Zunächst definieren wir, was ein balancierter Knotenschnitt ist, anschließend, wie man aus einem balancierten Kantenschnitt einen balancierten Knotenschnitt gewinnt.

Definition 5.15 ***b-balancierter Knotenschnitt***
Sei $b \in]0,1]$. Ein b-balancierter Knotenschnitt eines Graphen G=(V,E) ist eine Unterteilung von V in drei Mengen A, B und C, so dass folgende Bedingungen erfüllt sind. Der Graph $G' = (V' = V \setminus C, E')$, wobei E' genau diejenigen Kanten aus E enthält, die ausschließlich Knoten aus V' verbinden, ist der Gestalt, dass $V' = A \cup B$ und kein Paar von Knoten aus A und B liegt in der gleichen Zusammenhangskomponente von G'. Des Weiteren soll gelten, dass $b \cdot |V| \geq |A|$ und $b \cdot |V| \geq |B|$.

Man kann nun mit folgenden Algorithmus aus einem b-balancierten Kantenschnitt einen $(1-b)$-balancierten Knotenschnitt gewinnen:

Algorithmus 5.16 ***Approximation eines balancierten Knotenschnitts***
Seien A und B die Zusammenhangskomponenten des Graphen G, die durch einen b-balancierten Kantenschnitt entstanden sind. Wähle eine der beiden Mengen, o.B.d.A. A, und konstruiere den Knotenschnitt C. Wähle all diejenigen Knoten aus A aus, die mit einer der geschnittenen Kanten inzidieren (für die in G also eine Kante (u,v) existiert mit $u \in A$ und $v \in B$) und füge sie C hinzu. So erhält man einen 1-b balancierten Knotenschnitt als Unterteilung von G in die Mengen $A \setminus C, B$ und C, nach [1]

5.2 Anwendung des b-balancierten Knotenschnitts auf die Berechnung von Pfad- und Baumzerlegungen

Auf dieser Basis können wir jetzt den Approximationsalgorithmus von Bodlaender und Kloks nach [4] angeben. Zunächst erstellt man auf Basis des Algorithmus für b-balancierte Knotenschnitte eine Baumzerlegung, anschließend setzt man diese in eine Pfadzerlegung um. Diese ist maximal um einen logarithmischen Faktor abweichend von der optimalen Pfadzerlegung. Die folgenden beiden Darstellungen sind also stark an [4] angelehnt. Wenn im folgenden Algorithmus ein 2/3-balancierter Knotenschnitt gefordert wird, soll dieser mit obigem Algorithmus ausgerechnet werden und die Menge C als Schnitt zurückgegeben werden. Weiter ist $G[Z \cup W]$ der Graph, der aus G entsteht, wenn man alle Knoten aus G löscht, die nicht in Z oder W sind und die Kantenmenge entsprechend einschränkt, dass keine Kante zu einem Knoten mehr existiert, der nicht in $Z \cup W$ liegt.

Algorithmus 5.17 ***Näherung des Baumzerlegungsproblems***
input*: G=(V,E): Graph, Z, W: disjunkte Mengen von Knoten aus G.*
output*: Baumzerlegung von G.*

function zerlege(G,Z,W){
if$(3 \cdot |Z| \leq |W|)$
return Baumzerlegung, die nur einen Knoten enthält, dem $Z \cup W$ *zugeordnet ist;*
else {
S:= 2/3-balancierter Knotenschnitt von W *in* $G[Z \cup W]$*;*
S':= 2/3-balancierter Knotenschnitt von $G[Z \cup W]$*;*
Seien $G_1 = (V_1, E_1), ..., G_t = (V_t, E_t)$ *die Zusammenhangskomponenten von* $G' = (V' = Z \cup W \setminus (S \cup S'), E')$ *wobei E' alle Kanten in G sind, die nur Knoten in V' enthalten;*
for(int i=1; $i \leq t$*; ++i)* {
$Z_i := V_i \cap Z$*;*
$W_i := V_i \cap W$*;*
BZ_i *:=zerlege(*$Z_i, W_i \cup S \cup S', G$*);*
};
return Baumzerlegung mit Wurzel, die Knoten $W \cup S \cup S'$ *enthält und einer Kante zu den Wurzeln der* BZ_i*;*
};
}

Bemerkung 5.18 $\frac{2}{3}$ ***Schnitt einer Teilmenge der Knotenmenge***
Im obigen Algorithmus wird der $\frac{2}{3}$ *Schnitt einer Teilmenge der Knotenmenge gesucht (in der Zeile S:= 2/3-balancierter Knotenschnitt von W in G[Z∪W];). Das wurde bisher nicht thematisiert. Ein* $\frac{2}{3}$*-balancierter Knotenschnitt einer Teilmenge der Knotenmenge eines Graphen ist ein Knotenschnitt im Graph, so dass der Graph in zwei Komponenten zerfällt und in keiner Komponente mehr als* $\frac{2}{3}$ *der Knoten aus der Teilmenge enthalten sind. Einen solchen Schnitt kann man ebenfalls mit den von Leighton und Rao entwickelten Algorithmen finden, es müssen aber einige Anpassungen vorgenommen werden. Anstatt die Algorithmen noch einmal in dieser veränderten Form anzugeben, werden im Folgenden nur die nötigen Änderungen vorgestellt.*
Alle Knoten des Graphen G werden mit einem Knotengewicht belegt. Knoten, die in der Teilmenge W sind, werden mit dem Knotengewicht 1 versehen und alle anderen Knoten erhalten das Knotengewicht 0. Die Kennzahl p gibt für eine Menge von Knoten die Anzahl an Knoten zurück, die ein Knotengewicht von 1 haben.
Beim linearen Optimierungsproblem müssen die Bedarfe für alle Paare von Knoten aus W weiterhin 1 sein, alle anderen Bedarfe hingegen müssen 0 sein. Entsprechend muss auch die Bedingung $\sum_{(u,v)\in V} d(u,v) \geq 1$ *so angepasst werden, dass sie nur die Paare von Knoten in W umfasst:*
$\sum_{(u,v)\in W} d(u,v) \geq 1$.
Bei der Approximation des UMFP muss die Grenze des Radius $\frac{1}{2p^2}$ *lauten und die Genauigkeit des Ergebnisses ist ebenfalls von p statt n abhängig. Im Partitionierungsalgorithmus muss n durch p an jeder Stelle ersetzt werden*

und die G_i^+ werden nur für Knoten aus W gebildet. Bei der Näherung des b-balancierten Kantenschnitts muss schließlich die Größe der U_i und G_i anhand der Anzahl von Knoten mit Gewicht 1 berechnet werden.

Bemerkung 5.19 ***Beliebige Knotengewichte***
Die in der vorhergehenden Bemerkung angegebenen Modifikationen reichen aus, um die Algorithmen auch auf Graphen mit beliebigen nicht-negativen Knotengewichten anzuwenden. Das wird allerdings für den Algorithmus von Bodlaender nicht benötigt.

Beispiel 5.20 ***Approximativer Baumzerlegungsalgorithmus angewandt auf einen Beispielgraphen***
Um die Rekursion des Baumzerlegungsalgorithmus' verständlich zu machen, betrachten wir beispielhaft einen Rekursionsschritt auf einem Beispielgraphen. Wir betrachten dazu den im Folgenden dargestellten Beispielgraphen:

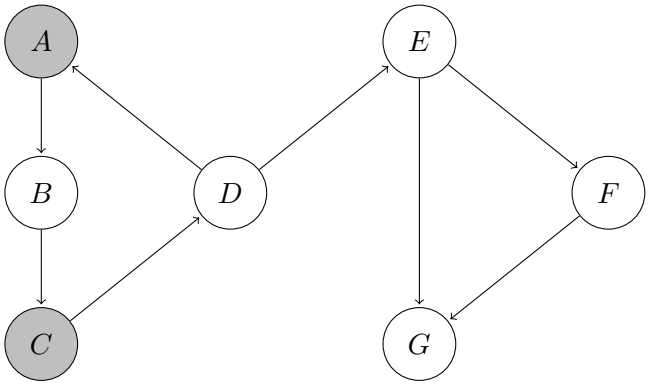


Abbildung 17: Beispielgraph für Rekursionsschritt

Die grau markierten Knoten A und C sind als W übergeben worden, die Knoten B, D, E, F und G sind als Z übergeben worden. Wir wählen als S' die Menge $\{D\}$, denn entfernt man D aus dem Graphen, so zerfällt dieser in zwei Komponenten, eine mit den Knoten A, B und C und eine mit den Knoten E, F und G. Es handelt sich also um einen balancierten Knotenschnitt. Weiter wählen wir als S die Menge $\{B, D\}$, entfernt man nämlich diese beiden Knoten, so zerfällt der Graph in drei Komponenten, eine die A enthält, eine die C enthält und eine die E, F und G enthält. Somit enthält jede Komponente höchstens einen Knoten aus W und daher keine Komponente mehr als $\frac{2}{3}$ der Knoten aus W. Wir bilden nun eine Tasche, die die Knoten aus S, S' und aus W enthält, also wird als Tasche $\{A, B, C, D\}$ gewählt. Die Kindtaschen werden nun durch weitere rekursive Aufrufe gebildet. Dafür müssen die Zusammenhangskomponenten in $G[W \cup Z] \setminus (S \cup S')$ betrachtet werden, derer es drei gibt.
Für die erste Komponente, die nur den Knoten A enthält, bilden wir $Z_1 = \emptyset$,

denn es ist kein Knoten in der Komponente, der aus Z stammt. W_1 enthält den einzigen Knoten der Komponente, der in W war, A. Die Funktion wird also mit den Eingaben $\emptyset$ und $\{A, B, D\}$ aufgerufen, um die erste Kindtasche zu erhalten.
Für die zweite Komponente, die nur den Knoten C enthält, bilden wir $Z_2 = \emptyset$, denn es ist kein Knoten in der Komponente, der aus Z stammt. W_2 enthält den einzigen Knoten der Komponente, der in W war, C. Die Funktion wird also mit den Eingaben $\emptyset$ und $\{B, C, D\}$ aufgerufen, um die zweite Kindtasche zu erhalten.
Für die dritte Komponente, die die Knoten E, F und G enthält, bilden wir $Z_3 = \{E, F, G\}$, denn alle drei Knoten stammen aus Z. W_3 enthält keinen Knoten, denn es gibt keinen Knoten in der Komponente, der aus W stammt. Die Funktion wird also mit den Eingaben $\{E, F, G\}$ und $\{B, D\}$ aufgerufen, um die dritte Kindtasche zu erhalten.
Schließlich wird die so erhaltene Baumzerlegung weiter nach oben an den Aufrufer gegeben.

Nun haben wir also eine Baumzerlegung des Graphen erhalten, gesucht war allerdings eine Pfadzerlegung. Diese lässt sich nun mit Hilfe eines von Ephraim Korach und Nir Soleil [11] vorgestellten Algorithmus' ermitteln.

Algorithmus 5.21 ***Umwandlung einer Baumzerlegung in eine Pfadzerlegung***
Wir fassen zunächst die Baumzerlegung als einfachen Baum auf und suchen eine Pfadzerlegung dieses Baums. Anschließend bilden wir die Pfadzerlegung des Ursprungsgraphen pz(G), indem wir für jede Tasche t der Pfadzerlegung des Baums eine Tasche in pz(G) bilden, die genau die Knoten aus G enthält, die die Taschen der Baumzerlegung, die durch t repräsentiert werden, enthalten. Wir verbinden die Taschen genau so, wie in der Pfadzerlegung des Baums.

Im obigen Algorithmus muss eine Pfadzerlegung eines Baums gebildet werden. Dafür verwenden wir den ebenfalls in [11] vorgestellten Algorithmus:

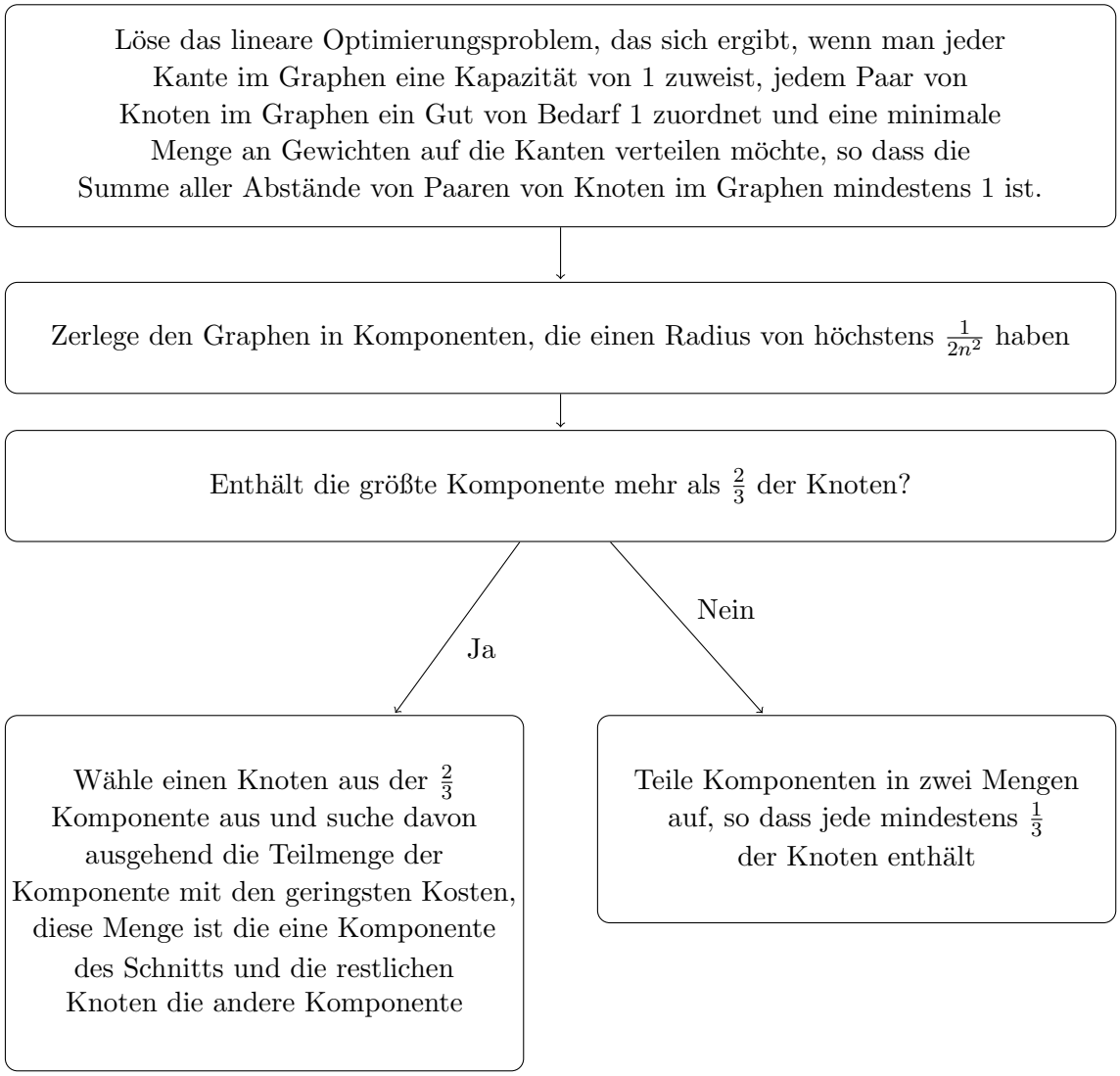
Algorithmus 5.22 ***Pfadzerlegung eines Baums / Waldes W***
Wir wählen eine Teilmenge der Knoten V von T aus, so dass wenn man diese aus T entfernt, T so in zwei Teile L und R zerfällt, dass keiner der Teile mehr als 2/3 der Knoten enthält. Auf Grund der Eigenschaft von W ein Wald zu sein, reicht dafür eine 0- oder 1-elementige Menge. Rufe diesen Algorithmus rekursiv auf L und R auf. Füge dann zur so erhaltenen Pfadzerlegung von L und R zu jeder Tasche V hinzu und verbinde ein Ende der Pfadzerlegung von L mit einem Ende der Pfadzerlegung von R. Die Rekursion bricht ab, wenn sie auf einen einzelnen Knoten trifft. Dessen Pfadzerlegung ist schlicht eine einzelne Tasche, die nur diesen Knoten enthält.

Somit erhalten wir nach Anwendung all dieser Schritte eine Pfadzerlegung des Graphen $G = (V, E)$, $|E| = n$, die maximal eine Pfadweite von $O(log(n) \cdot k \cdot log(n)) = O(log^2(n) \cdot k)$ hat, wobei k die Baumweite des Graphen ist. Da die Pfadweite größer gleich der Baumweite ist (jede Pfadzerlegung ist auch eine Baumzerlegung), weicht die Pfadweite der Approximation also höchstens $O(log^2(n))$ von einer optimalen Pfadzerlegung ab.

5.3 Zusammenfassung der Ergebnisse

Abschließend sei noch einmal die Idee des Algorithmus' in kompakter Form vorgestellt. Für einen gegebenen Graphen berechnen wir eine Baumzerlegung, indem wir den $\frac{2}{3}$-balancierten Knotenschnitt des Gesamtgraphen und den $\frac{2}{3}$-balancierten Knotenschnitt einer Teilmenge W der Knotenmenge des Graphen berechnen. Diese Schnitte teilen den Graphen in Komponenten auf, auf denen wir die Funktion rekursiv wieder aufrufen, das neue W für diesen Aufruf besteht aus den Knoten, die in den beiden Knotenschnitten enthalten sind, und den Knoten der Komponente, die schon vorher in W waren.
Um schließlich aus der Baumzerlegung eine Pfadzerlegung zu gewinnen, fassen wir die Baumzerlegung als einfachen Baum auf und ermitteln die Pfadzerlegung, indem wir eine Tasche auswählen, deren Entfernung den Baum in zwei möglichst gleichgroße Wälder zerfallen lässt, bilden rekursiv die Pfadzerlegung zu beiden Wäldern auf die gleiche Weise und fügen jeder Tasche der resultierenden Pfadzerlegungen die Knoten hinzu, die in der zuvor ausgewählten Tasche enthalten waren.
Es lohnt sich, sich noch einmal einen knappen Überblick über den relativ umfangreichen Prozess zur Ermittlung eines $\frac{2}{3}$-balancierten Knotensschnitts zu verschaffen. Man bestimmt zunächst den dünnsten Schnitt des Graphen und wiederholt das auf dem größeren der resultierenden Teilgraphen, bis der größere der beiden resultierenden Teilgraphen weniger als $\frac{1}{3}$ der Knoten enthält, die Vereinigung der kleineren Teilgraphen ergibt die eine Teilmenge des Schnitts, der Rest der Knotenmenge des Graphen die andere Teilmenge. Schließlich erhält man den balancierten Knotenschnitt, indem man eine der beiden Teilmengen des balancierten Kantenschnitts auswählt und die Menge der Knoten in dieser Teilmenge des balancierten Kantenschnitts, die adjazent zu einem Knoten aus der anderen Teilmenge sind, zusammenfasst.

Den Algorithmus zur Bestimmung des dünnsten Schnitts stellt das folgende Schaubild noch einmal kompakt dar:



Schwierigkeiten bei der Herausarbeitung dieses Algorithmus' ergaben sich sowohl im ersten als auch im zweiten Teil auf Grund sehr knapper Darstellungen in den Quellen. Bodlaender, Gilbert, Hafsteinsson und Kloks [4] zitieren Leighton und Raos Paper [12] zur Berechnung eines $\frac{2}{3}$-Knotenschnitts. Die Beschreibung des Vorgehens findet sich allerdings nur in [13] und auch nur für den $\frac{2}{3}$-Kantenschnitt - eine Umwandlung in einen Knotenschnitt musste noch vorgenommen werden. Die Knotenschnitte einer Teilmenge der Knotenmenge, für die Leighton und Rao ebenfalls zitiert werden, werden in dieser Form in beiden Arbeiten nicht vorgestellt, so dass der Algorithmus für den Kantenschnitt mit Knotengewichten genutzt werden musste, um die von Bodlaender, Gilbert, Hafsteinsson und Kloks gesuchten Knotenschnitte zu

finden. Bei der Bearbeitung der Arbeit von Leighton und Rao [13] hingegen erwies es sich als Schwierigkeit, dass das lineare Optimierungsproblem in der Form, wie es in dem Paper vorgestellt wird, kein lineares Optimierungsproblem ist, daher bedurfte es der Darstellung Iris [10], um die Lösung des Optimierungsproblems zu ermitteln.

6 Approximationsalgorithmus von Cattell, Dinneen und Fellows

Cattell, Dinneen und Fellows haben einen approximativen Algorithmus angegeben ([9]), der für ein fixiertes k auf einem eingegebenen Graphen in linearer Zeit entweder feststellt, dass die Pfadweite des Graphen größer als k ist, oder eine Pfadzerlegung der Pfadweite von maximal $O(2^k)$ angibt. Der Algorithmus basiert darauf, dass man versucht, einen vollständigen Binärbaum in den Graphen einzubetten. Gelingt die Einbettung, so ist die Pfadweite größer als k, gelingt sie nicht, so kann man aus den Zwischenschritten der Einbettung eine Pfadzerlegung gewinnen. Auch dieser Algorithmus bedarf einer gewissen begrifflichen Vorbereitung.

Definition 6.1 ***Partielle Ordnung***
Eine Relation $R \subset M \times M$ über der Menge M ist eine partielle Ordnung, wenn gilt:
1. Reflexivität: $(m, m) \in R$ gilt für alle $m \in M$.
2. Transitivität: Gilt $(m_1, m_2) \in R$ und $(m_2, m_3) \in R$ für $m_1, m_2, m_3 \in M$, so gilt auch $(m_1, m_3) \in R$.
3. Antisymmetrie: Gilt $(m_1, m_2) \in R$ und $(m_2, m_1) \in R$ für $m_1, m_2 \in M$, so gilt $m_1 = m_2$.

Definition 6.2 ***Verkleben von Graphen***
Wir sagen ein Graph G hat eine Grenze der Größe k, wenn es Knoten mit den Labeln 1, 2, ..., k gibt. Zwei Graphen mit Grenze der Größe k können verklebt werden, indem man die Knoten mit den Labeln 1, 2, ..., k der beiden Graphen identifiziert. Der so erhaltene Graph enthält alle Knoten des ersten und des zweiten Graphen, allerdings die Knoten mit den Labeln 1, 2, ..., k nur jeweils einmal.

Definition 6.3 ***Topologische Ordnung***
Ein Graph $G_1 = (V_1, E_1)$ heißt homöomorph in einen Graphen $G_2 = (V_2, E_2)$ einbettbar, wenn es eine injektive Abbildung h von E_1 nach $P(E_2)$ gibt (Dabei ist $P(E_2)$ die Menge aller Pfade über Kanten aus E_2), so dass für $e_1, e_2 \in E_1$ gilt: $e_1 \neq e_2 \rightarrow h(e_1) \cap h(e_2) = \emptyset$ und für alle $e \in E_1$ $h(e)$ ein Pfad in G_2 ist. Die topologische Ordnung der Graphen ergibt sich, indem man die Graphen gemäß ihrer homöomorphen Einbettbarkeit anordnet.

Definition 6.4 ***Minorordnung***
Eine alternative Definition zu obiger Definition der topologischen Ordnung ist die Minorordnung. Ein Graph $G_1 = (V_1, E_1)$ ist Minor eines Graphen $G_2 = (V_2, E_2)$, wenn ein zu G_1 isomorpher Graph aus G_2 durch wiederholtes Löschen von Knoten, Löschen von Kanten, sowie Anwenden der folgenden Kontraktionsregel gewonnen werden kann:
Für ein $(u, v) \in E_2$ führe folgende Schritte aus: Lösche (u,v) aus E_2, ersetze

(für alle $w \in V_2$) alle Kanten der Form (v,w) in E_2 durch (u,w), alle Kanten der Form (w,v) durch (w,u). Lösche v aus E_2.

Definition 6.5 ***Unteres Ideal***
$I \subset M$ ist ein unteres (Ordnungs-) Ideal zur partiellen Ordnung $\geq$ wenn aus $a \geq b$ mit $a \in I$ und $b \in M$ folgt: $b \in I$. Die Zerteilungsmenge von U und I ist $\{x \in M \setminus I \mid \neg\exists y \in M \setminus I(x \geq y \wedge \neg y \geq x)\}$.

Die Menge der Graphen G mit $pw(G) \leq k$ ist ein unteres Ideal zur topologischen Ordnung. Der hier vorgestellte Algorithmus nutzt die Feststellung, dass ein Graph mit n Knoten und einer Pfadweite von maximal k höchstens $nk - (k^2 + k)/2$ Kanten hat.

Definition 6.6 ***Vollständiger Binärbaum***
Ein Binärbaum T=(V,E) der Tiefe k ist vollständig, wenn jeder Knoten entweder einen Abstand zur Wurzel von k hat oder genau zwei Nachfolgeknoten hat. Ein vollständiger Binärbaum der Tiefe n hat $2^n - 1$ Knoten. Im Folgenden bezeichnen wir mit B_h den vollständigen Binärbaum der Tiefe h, mit h(t) das kleinste h so dass B_h Pfadweite t hat und mit f(t) die Anzahl der Knoten von $B_{h(t)}$.

Satz 6.7 ***Schranke für die Pfadweite eines vollständigen Binärbaums***
Ein vollständiger Binärbaum der Tiefe 2k+2 hat eine Pfadweite größer als k.

Diese Aussage ist die zentrale Aussage für den Algorithmus von Cattell, Dinneen und Fellows. Die Richtigkeit dieser Aussage sieht man wie folgt ein: Ein (nicht notwendigerweise binärer) Baum der Pfadweite größer als k lässt sich wie folgt konstruieren:
1. Eine einzelne Kante ist der einzige zusammenhängende Graph der Pfadweite 1. Ist eine einzelne Kante ein Minor eines Graphen G, so hat G eine Pfadweite größer als 0.
2. Seien T_1, T_2, T_3 Bäume der Pfadweite größer als t, dann hat der Baum, der entsteht, wenn man einen neuen Knoten v bildet und diesen mit den Wurzelknoten von T_1, T_2 und T_3 verbindet, Pfadweite größer als $t + 1$.
Der so konstruierte Baum T der Pfadweite größer als k ist ein Minor des vollständigen Binärbaums der Höhe $2k + 2$. Also hat der vollständige Binärbaum der Höhe $2k + 2$ eine Pfadweite größer als k. Dass T ein Minor von B_{2k+2} ist, kann man einsehen, indem man von der Wurzel von T aus in einer Breitensuche jeden Knoten v mit drei Nachfolgern gegen drei Knoten austauscht, wobei einer der Knoten die beiden anderen Knoten als Kinder hat und die Nachfolger von v auf die übrigen beiden Knoten verteilt werden. Bei dieser Operation wird auf jeder Ebene des Baums die Tiefe des Baums jeweils um 1 erhöht, T hat die Tiefe $k+2$ und insgesamt wird diese Operation auf k Ebenen durchgeführt. Insgesamt ergibt sich ein (nicht vollständiger) Binärbaum der Tiefe $k + k + 2 = 2k + 2$, der natürlich Minor von B_{2k+2} ist.

Mit dieser Vorbereitung kann nun der eigentliche Approximationsalgorithmus angegeben werden, der für ein festes k entweder feststellt, dass ein eingegebener Graph G eine Pfadweite größer als k hat, oder eine Pfadzerlegung mit Pfadweite maximal $O(2^k)$ angibt.

Bemerkung 6.8 ***Vorbemerkungen und Erklärungen zum Algorithmus***
Der Algorithmus sucht (für Pfadweite k auf einem eingegebenen Graphen G) eine Einbettung des vollständigen Binärbaums $B_{h(k)}$, wenn dies gelingt, ist die Pfadweite des Graphen größer als k. Die Knoten von $B_{h(k)}$ werden als Tokens repräsentiert. Diese Tokens sind Binärstrings, die den Weg durch den Binärbaum repräsentieren. Die Wurzel wird durch den leeren String ϵ dargestellt und wenn P der Token eines Vaterknotens zweier Kinder ist, dann repräsentiert der Token P0 den linken, der Token P1 den rechten Kindknoten von P. Es wird also an den String, der der Token des Vaterknotens ist, eine 1 oder eine 0 angehängt. Weiter sei $P[i]$ die Menge der Knoten in G, die zum Zeitpunkt i mit einem Token versehen sind. Diese Mengen zu speichern ist essentiell, da diese die gesuchte Pfadzerlegung darstellen, wenn die Pfadweite des Graphen mit diesem Algorithmus nicht als größer als k nachgewiesen werden kann. Als letzte Vorbemerkung sei angemerkt, dass alle Knoten in G eingefärbt werden. Zu Beginn des Algorithmus' sind alle Knoten blau gefärbt. Jedes Mal, wenn ein Token auf einem Knoten platziert wird, wird dieser Knoten automatisch rot gefärbt und ändert seine Farbe nicht mehr.

Zum besseren Verständnis sei beispielhaft im Folgenden der Token-Baum der Tiefe 3 dargestellt.

Beispiel 6.9 ***Token-Baum der Tiefe 3***

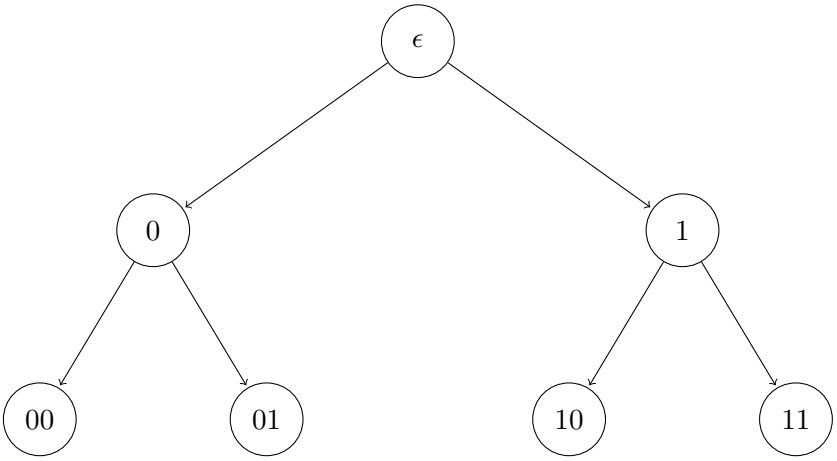


Abbildung 18: Token-Baum der Tiefe 3

Algorithmus 6.10 ***Approximation der Pfadzerlegung, zweite Version***

```
Knoten[] GrowTokenTree(){
    if(token ϵ noch nicht platziert){
        wähle beliebigen Knoten v aus G und platziere ϵ auf v;
    }
    while(Es gibt Knoten u ∈ G mit Token T und noch blauem Nachbar v
          und es gibt einen Nachfolger-Token T0 oder T1, der noch nicht
          platziert wurde){
        belege v mit dem (einem der) noch nicht platzierten Nachfolger-
              Token T0 bzw. T1;
    }
    return Menge der Knoten aus G, die mit Token belegt sind;
}

   Pfadzerlegung zerlege(Graph G){
    int i=0;
    P[i] = GrowTokenTree();
    do{
        T = ein Token, das ein nicht gesetztes Kindtoken hat;
        Entferne T aus G;
            if(T hatte ein Kindtoken, das gesetzt war){
                Ersetze alle Tokens T1S oder T0S durch TS;
            };
        ++i;
        P[i] = GrowTokenTree();
    }while(|P[i]| ≠ f(k)∧ Es gibt in G noch blaue Knoten);
    return [P[1], P[2], ..., P[i]]
}
```

Wenn in der Ausgabe dieses Algorithmus' alle Knoten von G enthalten sind, handelt es sich um eine korrekte Pfadzerlegung von G, sonst hat G eine Pfadweite größer als k.

Wenn im obigen Algorithmus die Einbettung des Graphen gelungen ist, so wird in dieser Version des Algorithmus' abgebrochen und zurückgegeben, dass der Graph nicht die eingegebene Pfadweite hat. Es ist aber in manchen Fällen möglich, den Algorithmus fortzusetzen und so noch eine Pfadzerlegung zu erhalten. Dann nämlich, wenn ein Blatt-Token auf einen Knoten gesetzt ist, dessen Nachbarknoten bereits alle rot sind, kann dieses Token entfernt und der Algorithmus weiter fortgesetzt werden. In der Implementierung des Algorithmus', die im Zuge dieser Arbeit erstellt wurde, wird diese Änderung vorgenommen.

Satz 6.11 ***Garantie des Algorithmus***
Der obige Algorithmus garantiert für Eingaben G und k, dass eine Pfadzerlegung mit einer Pfadweite von höchsten $2^{2 \cdot k+2}$ gefunden wird, falls die Pfadweite von G höchstens k ist. Andernfalls gibt es zwei mögliche Ausgaben. Entweder wird herausgefunden, dass der Graph eine Pfadweite größer als k hat, oder es wird eine Pfadzerlegung mit einer Pfadweite von höchsten $2^{2 \cdot k+2}$ gefunden.

Beispiel 6.12 ***Cattells / Dinneens / Fellows Approximationsalgorithmus angewandt***
Wir bemühen ein weiteres Mal den ersten Beispielgraphen:

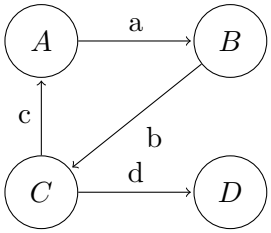


Abbildung 19: Beispielgraph

und wenden den obigen Algorithmus auf diesen Graphen an. Um eine vernünftige Pfadzerlegung erhalten zu können, wählen wir Pfadweite 0. Zwar hat der Graph offensichtlich nicht Pfadweite 0, doch kann dennoch eine valide Pfadzerlegung herauskommen. Wählt man eine andere Pfadweite, kann es bei einem so kleinen Graphen passieren, dass alle Knoten in einer Tasche landen. Im ersten Schritt wird das ϵ-Token auf den Knoten A gelegt und der Token-Baum darf wachsen. Der Token-Baum sieht nach dem ersten Schritt so aus:

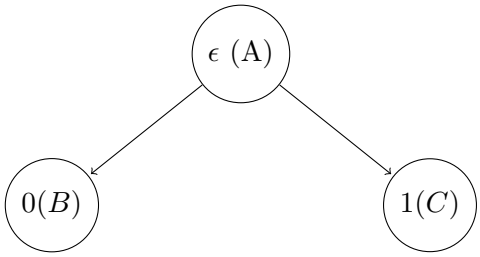


Abbildung 20: Token-Baum nach dem ersten Schritt

Eigentlich könnten wir an dieser Stelle aufhören und feststellen, dass der Graph eine größere Pfadweite als 0 hat, aber da uns die Pfadzerlegung mehr interessiert als die Pfadweite, prüfen wir, ob wir den Algorithmus noch weiter fortsetzen können.

Wir wählen Token 0 zum Löschen aus, denn alle Nachbarn von B wurden bereits mit einem Token versehen und erhalten so, nch einem Grow-Token-Tree Aufruf, der nichts am Token-Baum verändert, da alle Nachfolger von A schon rot sind, den folgenden Token-Baum:

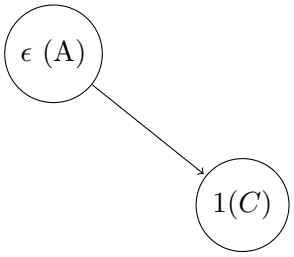


Abbildung 21: Token-Baum nach dem zweiten Schritt

Nun wählen wir Token ϵ zum Löschen aus, denn Token 0, einer der Nachfolgeknoten von A, ist zur Zeit nicht vergeben. Knoten C erhält nun das ϵ-Token und lässt man dann den Token-Baum wieder wachsen, so erhält man den folgenden Token-Baum:

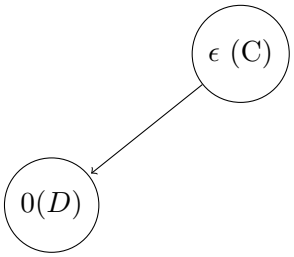


Abbildung 22: Token-Baum nach dem dritten Schritt

Anschließend sind keine Knoten mehr übrig, die noch nie ein Token hatten. Wir können also aufhören und die Pfadzerlegung als Folge derjeniger Knoten angeben, die nach den einzelnen Schritten ein Token besessen haben. Die so gewonnene Pfadzerlegung sieht dann wie folgt aus:

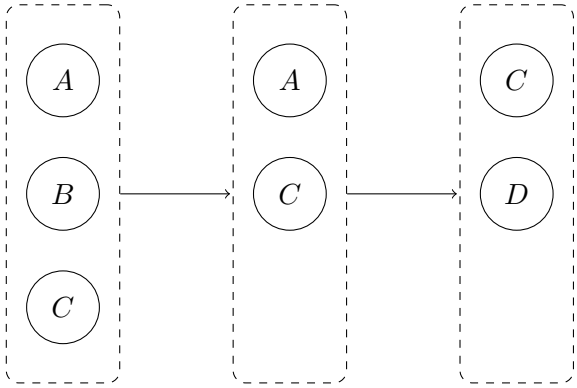
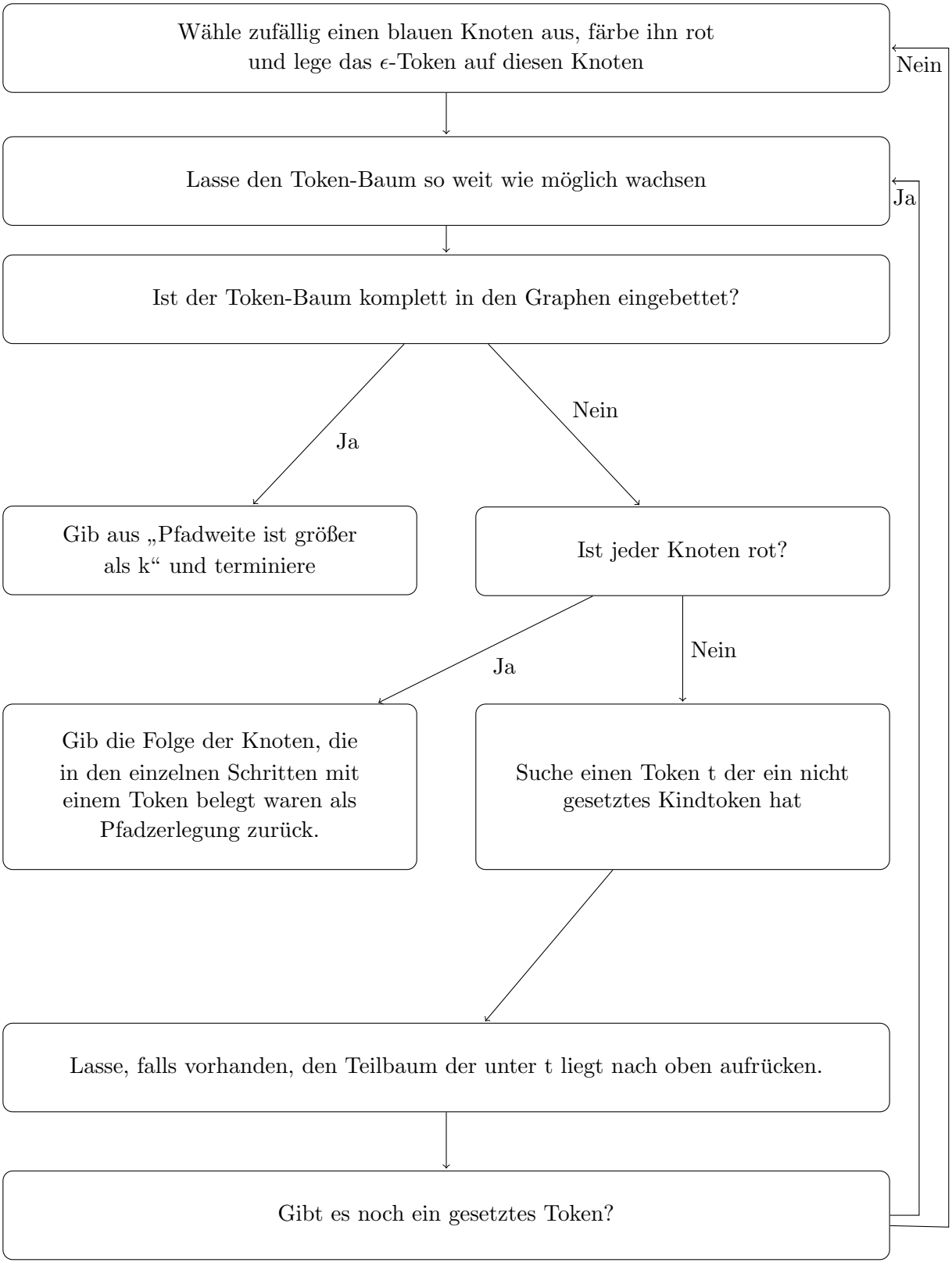


Abbildung 23: Resultierende Pfadzerlegung

Zusammenfassend versuchen wir in diesem Approximationsalgorithmus einen vollständigen Binärbaum einer vorgegebenen Tiefe in den Graphen einzubetten. Gelingt dies nicht vollständig und gibt es aber noch Knoten, die noch nie ein Token hatten, so wählen wir einen Knoten im Binärbaum, der ein nicht gesetztes Kindtoken hat. Derartige Knoten können wir dann aus der Einbettung herausnehmen und haben so vielleicht die Möglichkeit, den Baum in den Graphen einzubetten. Terminiert das Verfahren, ohne zwischendurch eine vollständige Einbettung des Binärbaums zu erhalten, so erhalten wir eine Pfadzerlegung indem wir die Knoten, die in den einzelnen Schritten Knoten aus dem Baum zugeordnet waren, zu Taschen zusammenfassen. Gelingt die Einbettung, so erhalten wir eine untere Schranke für die Pfadweite des Graphen. Dieses Vorgehen wird durch das folgende Schaubild noch einmal verdeutlicht:


Wähle zufällig einen blauen Knoten aus, färbe ihn rot und lege das ϵ-Token auf diesen Knoten
Nein
Lasse den Token-Baum so weit wie möglich wachsen
Ja
Ist der Token-Baum komplett in den Graphen eingebettet?
Ja
Nein
Gib aus „Pfadweite ist größer als k“ und terminiere
Ist jeder Knoten rot?
Ja
Nein
Gib die Folge der Knoten, die in den einzelnen Schritten mit einem Token belegt waren als Pfadzerlegung zurück.
Suche einen Token t der ein nicht gesetztes Kindtoken hat
Lasse, falls vorhanden, den Teilbaum der unter t liegt nach oben aufrücken.
Gibt es noch ein gesetztes Token?

7 Anmerkungen zur Implementierung

Im Rahmen dieser Arbeit wurde ein Programm entwickelt, das die beiden Approximationsalgorithmen umsetzt und Pfad- sowie Baumzerlegungen für einen vom Benutzer eingegebenen Graphen in eine Textdatei speichert. Das Programm wurde in der Programmiersprache C++ mit Hilfe der Dev-C++ Umgebung und des gcc-Compiler für Windows entwickelt.

7.1 Aufbau des Programmcodes

Der Programmcode ist auf vier Dateien verteilt, main.cpp, Graph.h, Graph.cpp und Pfadzerlegung.h. Graph.h enthält die Header der wichtigsten Graphen-Klassen. Insbesondere werden in dieser Datei die Klassen Graph, Gplus und Komponente deklariert. Die Klasse Graph dient der internen Repräsentation eines Graphen und bietet eine Reihe wichtiger Funktionalitäten wie das Suchen der Zusammenhangskomponenten oder den Dijkstra-Algorithmus zum Bestimmen der kürzesten Wege in einem Graphen. Alle Methoden der Klasse Graph sind in der Datei Graph.cpp definiert. Die Klasse Komponente fasst die besonders von Leighton und Rao benötigten Teilmengen der Knotenmenge eines Graphen und kann verwendet werden, um wichtige Kenngrößen wie die Kapazität der Komponente oder den Anteil an der Gesamtknotenmenge des zugehörigen Graphen zu berechnen. Gplus ist von der Klasse Graph abgeleitet und dient der Repräsentation der G^+-Graphen, die zum Beispiel im Partitionierungsalgorithmus von Leighton und Rao benötigt werden. Pfadzerlegung.h enthält Deklarationen und Definitionen der Pfadzerlegung und zugehörigen Taschen, sowie der Klasse BaumTasche, die verwendet wird, um Baumzerlegungen im Programm zu repräsentieren. Außerdem enthält diese Datei Hilfsklassen für die Berechnung einer Pfadzerlegung zu einer gegebenen Baumzerlegung.
Herzstück der Implementierung ist aber die main.cpp. In dieser Datei ist neben dem Hauptprogramm, in dem man einen Graphen eingeben kann und dann den Algorithmus der Wahl ausführen kann, die Implementierung der eigentlichen Algorithmen zur Baum- und Pfadzerlegung zu finden. Im Folgenden werden die drei wichtigsten Methoden für den Anwender knapp vorgestellt. Die Methode BaumTasche* BodlaenderKloks(Graph* g) ermittelt mit Hilfe des Algorithmus' von Bodlaender, Kloks, Gilbert und Hafsteinsson eine Baumzerlegung und gibt diese zurück. Pfadzerlegung* BaumZuPfad(BaumTasche* b, Graph* g) ermöglicht die Umwandlung einer beliebigen Baumzerlegung in eine Pfadzerlegung. Um zu einem Graphen g die Pfadzerlegung nach dem Algorithmus von Bodlaender, Kloks, Gilbert und Hafsteinsson zu erhalten, würde man also den Aufruf BaumZuPfad(BodlaenderKloks(g), g) verwenden. Schließlich kann man mit Hilfe

der Methode Pfadzerlegung& CattellDinneenFellows(Graph* g, int pfadweite) eine Pfadzerlegung nach dem Algorithmus von Cattell, Dinneen und Fellows berechnen.

7.2 Verwendung des Programms

Zwar sollte die Kompilierung und Verwendung des Programms auch unter anderen Betriebssystemen möglich sein, da keine Verwendung von systemspezifischen Befehlen gemacht wurde, die folgende Beschreibung orientiert sich aber an der Verwendung unter einem Windows-Betriebssystem. Um das Programm zu starten, muss die Datei „Graphen.exe“ doppelgeklickt werden. Anschließend erscheint ein Konsolenfenster, das den Benutzer zunächst auffordert, einen Graphen einzugeben. Um die Eingabe so leicht wie möglich zu gestalten, ist sie auf die für die Pfad- und Baumzerlegungen wesentlichen Informationen beschränkt. Zunächst wird der Anwender gefragt, wie viele Knoten der Graph enthalten soll. Diese werden dann direkt mit den Namen $1, 2, ..., n$, wobei n die eingegebene Zahl ist, angelegt. Anschließend wird nach der Anzahl der Kanten im Graph gefragt; da Loops für eine Pfadzerlegung uninteressant sind, sind nur Kanten zwischen zwei verschiedenen Knoten zulässig. Die Eingabe der Kanten erfolgt dann über die Namen der Knoten. Möchte man also beispielsweise durch die erste Kante den dritten Knoten mit dem vierten Knoten verbinden, so gibt man auf die Frage „Von welchem Knoten soll die 1. Kante ausgehen?“ eine 3 ein und auf die Frage „Zu welchem Knoten soll die 1. Kante gehen?“ eine 4. Hat man alle Kanten auf diese Weise eingegeben, wird der Graph auf der Konsole ausgegeben und man kann aus vier Optionen wählen. Die Aufforderung zur Wahl der Option sieht wie folgt aus:

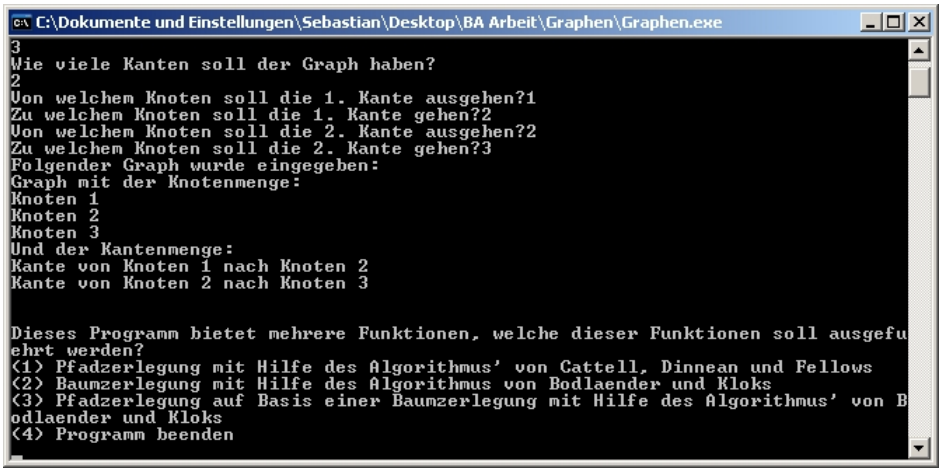


Abbildung 24: Screenshot des Programmmenüs

Durch Eingabe der entsprechenden Zahl kann man die Wahl treffen.

Wählt man eine der drei ersten Optionen, muss man wählen, unter welchem Dateinamen die Ergebnisse abgespeichert werden sollen und im Fall dessen, dass man die 1 gewählt hat, außerdem noch angeben, mit welcher Pfadweite als Eingabe der Algorithmus gestartet werden soll. Sobald die Berechnungen abgeschlossen sind, wird das Ergebnis der Berechnung auf dem Bildschirm und in die angegebene Datei ausgegeben und man landet wieder im Hauptmenü, das im obigen Screenshot zu sehen ist.

7.3 Besonderheiten bei der Implementierung

Der Großteil des Programmcodes sollte mit Hilfe der obigen Algorithmenbeschreibungen relativ einfach nachzuvollziehen sein. Eine Besonderheit findet sich allerdings bei der Implementierung des Algorithmus' von Cattell, Dinneen und Fellows. Die Token werden anders als in der Beschreibung des Algorithmus' weiter oben nicht mit Strings bezeichnet. Grund ist, dass sich natürliche Zahlen als Namen für die Tokens als angenehmer in der Implementierung erweisen, da man so leicht auf Arrays zurückgreifen kann. Die Wurzel des Baums wird nicht mit ϵ sondern mit 0 bezeichnet und für einen Knoten mit der Nummer i gilt: Das linke Kind des Knotens wird mit der Nummer $2 \cdot i + 1$ und das rechte Kind des Knotens wird mit der Nummer $2 \cdot i + 2$ versehen. Die Korrektheit dieser Nummerierung sieht man wie folgt ein:

Induktionsanfang: $i = 0$: $2 \cdot 0 + 1 = 1$ und $2 \cdot 0 + 2 = 2$. Für alle Knoten auf der ersten Ebene gilt die Aussage also.

Induktionsvoraussetzung: Die Aussage gelte für alle Ebenen von 1 bis k.

Induktionsschritt: Zeige: Die Aussage gilt auch für $k + 1$.
Ein vollständiger Binärbaum der Tiefe t hat $2^t - 1$ Knoten. Der letzte Knoten auf der Ebene k hat also den Index $2^k - 2$ und der letzte Knoten auf der Ebene $k+1$ hat den Index $2^{k+1} - 2$. Sei Knoten i auf der Ebene $k+1$ des Binärbaums, dann gilt, dass i sich schreiben lässt als $2^k - 2 + j$, $j \in \mathbb{N}$, Knoten i ist also der j-te Knoten auf der $k + 1$-ten Ebene. Jeder Knoten vor i auf der j-ten Ebene hat zwei Kinder, die ersten $2 \cdot (j - 1)$ Knoten der $k + 2$-ten Ebene sind also die Kinder der Knoten vor dem Knoten i und die nächsten beiden Knoten sind die Kinder des Knotens i. Das linke Kind hat also den Index $2^{k+1} - 2 + 2 \cdot (j - 1) + 1$ und das rechte Kind den Index $2^{k+1} - 2 + 2 \cdot (j - 1) + 2$. Nun gilt aber $2^{k+1} - 2 + 2 \cdot (j - 1) = 2 \cdot 2^k - 2 + 2 \cdot (j - 1) = 2 \cdot (2^k - 1 + j - 1) = 2 \cdot (2^k + j - 2) = 2 \cdot i$. Also gilt die Aussage auch für die Ebene $k + 1$.

7.4 Vergleichstest der Algorithmen

Die folgende Tabelle zeigt das Ergebnis eines Vergleichstests zwischen den beiden Approximationsalgorithmen für die optimale Pfadzerlegung. Insgesamt wurden Laufzeit und Güte anhand von fünf verschiedenen Graphen verglichen. Algorithmus 1 ist in der folgenden Tabelle der Algorithmus von Bodlaender, Kloks, Hafsteinsson und Gilbert, Algorithmus 2 der von Cattell, Dinneen und Fellows. Außerdem wird die Zeit, die für das Lösen des ersten linearen Optimierungsproblems im Algorithmus 1 zur Bestimmung von S' benötigt wird, aufgeführt. Diese Kennzahl wird aufgeführt, weil dieses Mal das einzige Mal ist, dass das Optimierungsproblem für den gesamten eingegebenen Graphen gelöst werden muss, in allen weiteren Rekursionsschritten werden nur noch Teilgraphen des Gesamtgraphen betrachtet.

Auf Grund des Zufallselements des zweiten Algorihmus' wurde dieser insgesamt fünf Mal je Graph ausgeführt und die Pfadweiten der so entstandenen Pfadzerlegungen angegeben. Für alle Graphen wurde als Eingabe $k = 1$ gewählt. Für den ersten Graphen findet der Algorithmus mit $k = 0$ hin und wieder eine Pfadzerlegung mit Pfadweite 0, meistens aber findet er überhaupt keine zulässige Pfadzerlegung. Die Ergebnisse zeigen, dass Algorithmus 1 für kompliziertere Graphen bessere Ergebnisse liefert als Algorithmus 2. Die Laufzeit hingegen wird bei Algorithmus 1 sehr schnell viel schlechter, so dass dieser für Graphen jenseits von 20 Knoten zur Zeit nur mit viel Geduld zu gebrauchen ist. Algorithmus 2 hingegen hat trotz mehrerer Textausgaben nie mehr als eine Sekunde für seine Berechnungen benötigt.

Die fünf Graphen lassen sich wie folgt beschreiben. G_1 ist ein Pfad und G_2 ein vollständiger Binärbaum der Tiefe 4. G_3 ist eine Clique der Größe 5, zudem ist jeder Knoten der Clique mit zwei Knoten des Grades 1 verbunden. Bei G_4 handelt es sich um einen Kreis und G_5 ist ein Dreieck, dessen Seitenmittelpunkte wiederum ein Dreieck bilden. Abbildungen 24 bis 28 zeigen die Graphen G_1 bis G_5.

	G_1	G_2	G_3	G_4	G_5
Anzahl Knoten:	20	15	15	10	6
Anzahl Kanten:	19	14	20	10	9
Zeit Algorithmus 1:	10:52	2:09	7:32	0:18	0:04
Zeit Algorithmus 2:	0:01	0:01	0:01	0:01	0:01
Zeit zur Berechnung des ersten S':	9:56	2:07	7:32	0:15	0:03
Pfadweite Algorithmus 1:	7	7	5	5	4
Pfadweite Algorithmus 2:	3-7	10	10	6	5
Baumweite Algorithmus 1:	4	7	5	4	4
Pfadweite des Graphen:	1	2	5	2	3

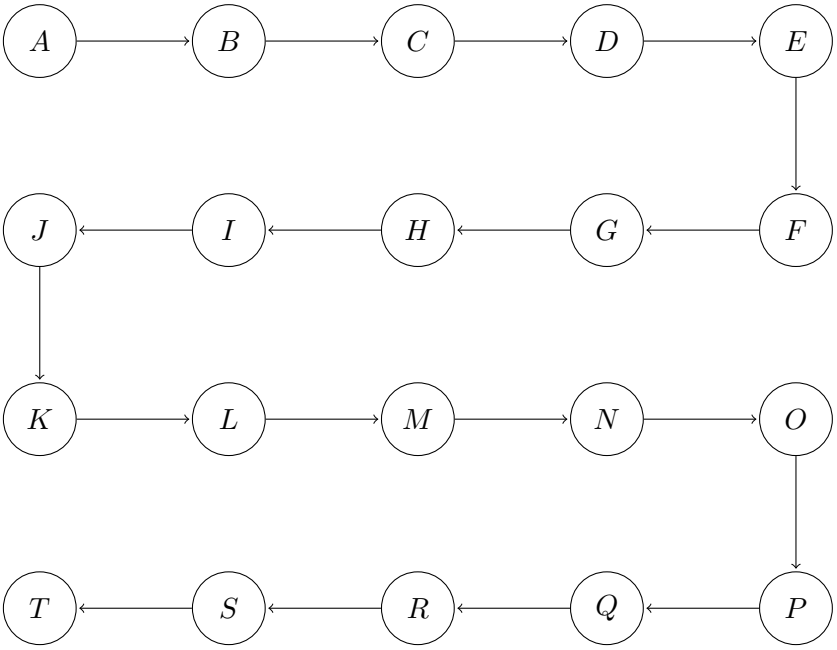


Abbildung 25: G_1

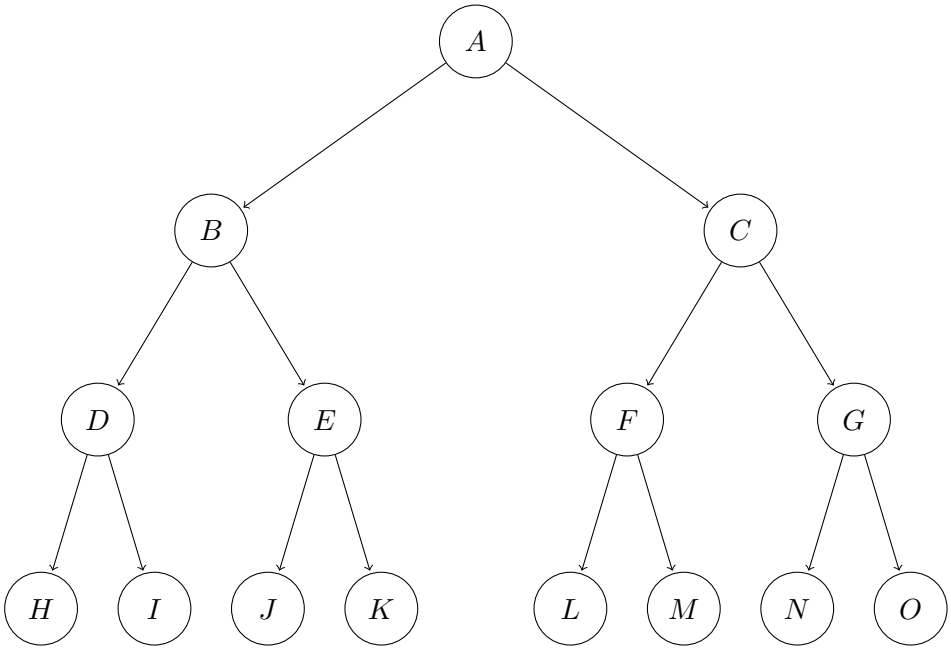


Abbildung 26: G_2

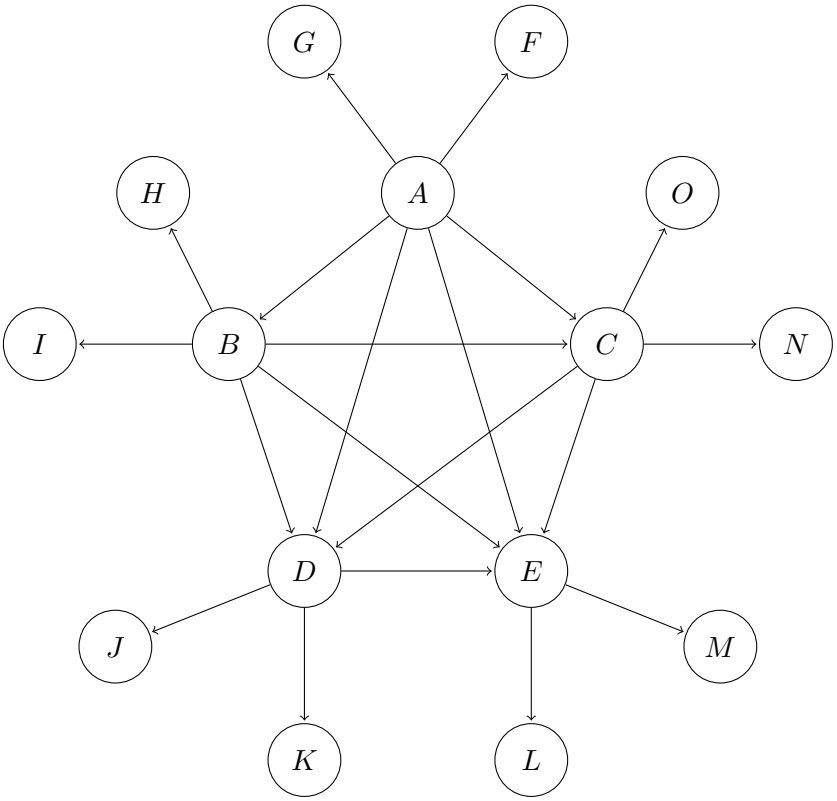


Abbildung 27: G_3

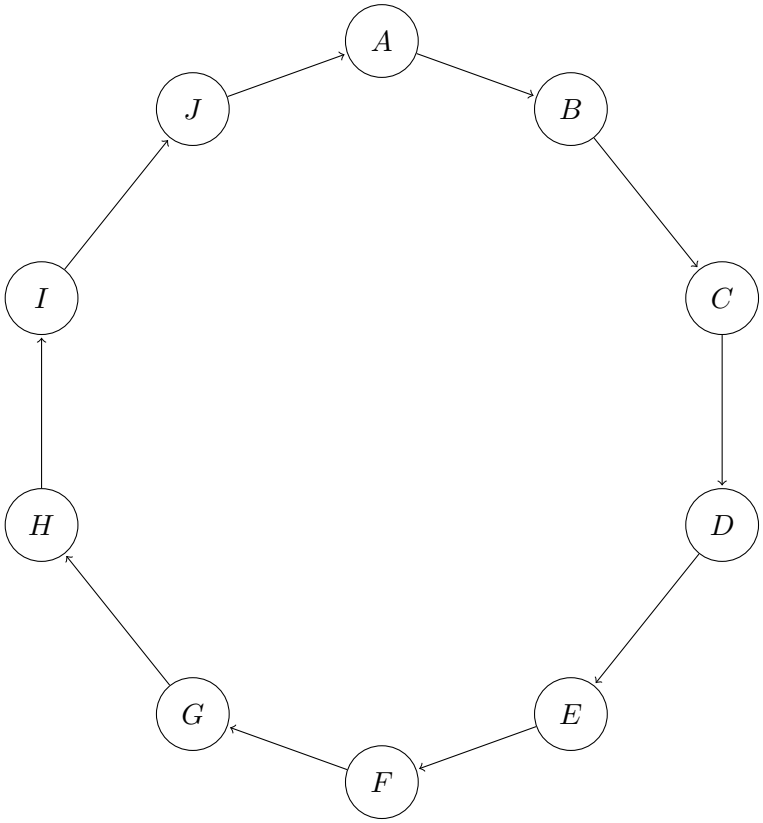


Abbildung 28: G_4

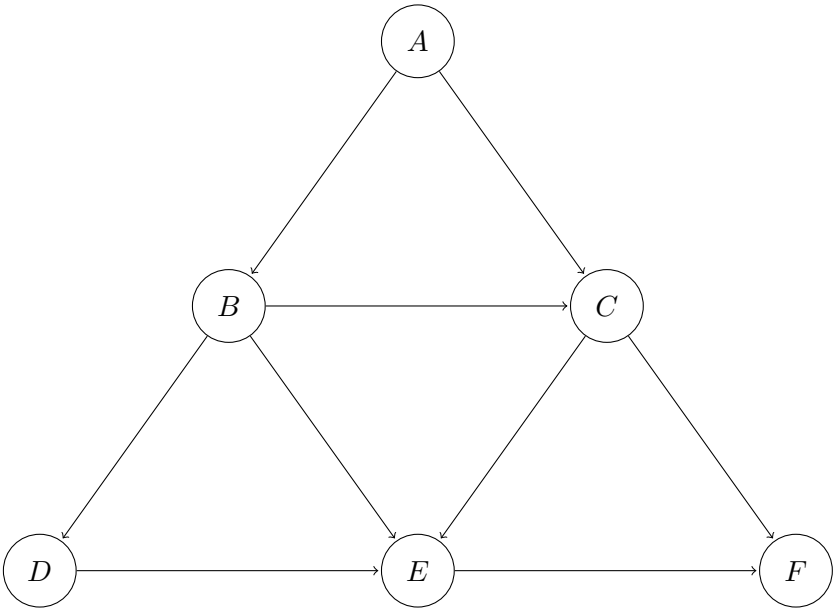


Abbildung 29: G_5

8 Ausblick

Im zweiten Kapitel wurde die zugrundeliegende Motivation für diese Arbeit gegeben, im Folgenden soll noch ein kurzer Überblick darüber gegeben werden, wie mit Hilfe einer Pfadzerlegung der Graphen G eine Folge von Operatoren gefunden werden kann, die den Graphen G bilden. Hypergraphen werden dabei als gewöhnliche Graphen aufgefasst, bei denen k-stellige Hyperkanten durch $\frac{k \cdot (k-1)}{2}$ Kanten zwischen den beteiligten Knoten ersetzt werden.
Die erste Tasche einer Pfadzerlegung wird in eine Folge von Operatoren umgewandelt, indem nacheinander alle Kanten von G, für die die beteiligten Knoten in der Tasche enthalten sind, mit der Operation *edge* hinzugefügt werden. Knoten, die auf diese Weise doppelt hinzugefügt wurden, werden anschließend (oder besser so früh wie möglich, um das Interface so klein wie möglich zu halten) mit $fuse$ verschmolzen, dafür sind die Umsortieroperatoren *perm* und *trans* zu beachten. Dann werden alle Knoten, die noch nicht im Interface sind, hinzugefügt. Ausgehend von einer Tasche, die bereits in Operatoren umgeformt wurde, gewinnt man aus der nächsten Tasche in der Pfadzerlegung die nächsten Operatoren, indem man zunächst mit *res* all diejenigen Knoten aus dem Interface entfernt, die in der nächsten Tasche nicht mehr enthalten sind und dann alle neu hinzugekommenen Kanten hinzufügt (ggf. wieder gleiche Knoten mit $fuse$ verschmelzen) und neu hinzugekommene Knoten, die noch mit keiner Kante verbunden sind, mit *vertex* hinzufügt. Das nächste Ziel ist es also, aus den Pfadzerlegungen auf ebendiese Weise Eingaben für das Programm zu Christoph Blumes Diplomarbeit zu gewinnen.

Die in dieser Arbeit vorgestellten Approximationsalgorithmen bieten noch Raum für Verbesserungen. Insbesondere die Laufzeit lässt sich teilweise erheblich verbessern. Ein Flaschenhals des Algorithmus' von Leighton und Rao ist das lineare Optimierungsproblem, das sich zwar in Polynomzeit lösen lässt, aber dennoch für große Graphen als sehr langwierig erweist. Farhad Shahrokhi und D. W. Matula [17] haben einen Approximationsalgorithmus für das Maximalflussproblem mit mehreren Gütern vorgestellt, der eine Beschleunigung des Gesamtverfahrens ermöglicht. Dieser Algorithmus benötigt allerdings enorm hohe Rechengenauigkeiten. Der Algorithmus von Shahrokhi und Matula basiert auf einer Distanzfunktion, die zunächst allen Kanten das gleiche Gewicht zuordnet. Anschließend werden die Pfade zwischen Knoten betrachtet, auf denen keine Kanten mit Gewicht Null liegen. Dann wird für ein Paar von Knoten von dem Pfad der (interpretiert man die Gewichte als Flüsse) den größten Fluss hat, Fluss auf denjenigen Pfad umverteilt, der den kleinsten Fluss zwischen den beiden Knoten hat.

Aufgrund der bereits zuvor genannten komplexitätstheoretischen Ergebnisse ist, falls $P \neq NP$, nicht zu erwarten, dass ein genauer Algorithmus für

das Pfadweitenproblem in Polynomzeit gefunden wird. Ungünstig erscheint, dass im Approximationsalgorithmus von Bodlaender sehr viele Hilfsprobleme gelöst werden müssen, um schließlich eine Approximation des Pfadweitenproblems zu finden, auch in dieser Hinsicht gibt es sicherlich Verbesserungspotential.

Der Algorithmus von Cattell, Dinneen und Fellows hingegen liefert oft sehr schlechte Pfadzerlegungen mit unnötig großen Pfadweiten. Eine Verbesserungsmöglichkeit wäre, auch vollständige Binärbäume zur Einbettung zu verwenden, die nicht eine Tiefe der Form $2 \cdot k + 2$, $k \in \mathbb{N}_0$ haben. Dadurch könnte man, falls man keine Pfadzerlegung mit Eingabe k findet und nur eine sehr schlechte Pfadzerlegung mit Eingabe $k + 1$ findet, in manchen Fällen eine bessere Pfadzerlegung erhalten. Dieses Vorgehen hätte allerdings den Nachteil, dass man im Fall des Misserfolgs eines solchen Durchlaufs keine Aussage über die Pfadweite treffen kann, die man nicht auch schon auf Grund des Durchlaufs mit Eingabe k treffen konnte.
Außerdem wird, wie bereits im Beispiel zum Algorithmus von Cattell, Dinneen und Fellows erwähnt, nicht sofort aufgegeben, wenn eine Einbettung des Baums in den Graphen gefunden wurde. Manchmal kann man dennoch eine valide Pfadzerlegung mit dem aktuellen Ansatz finden, dann nämlich, wenn eines der Blätter des Baums auf einen Knoten gemapt wird, der keine Knoten mehr als Nachbarn hat, die noch nicht mit einem Token belegt wurden. Dann kann man das Token von diesem Knoten einfach entfernen. In der Implementierung wird das ausgenutzt, der Anwender aber darüber informiert, dass bereits sicher ist, dass die Pfadweite des Graphen größer als die eingegebene Pfadweite ist.

Abgesehen von den hier vorgestellten Algorithmen sind auch noch eine ganze Reihe von Algorithmen bekannt, die Baum- oder Pfadzerlegungen für bestimmte Klassen von Graphen ermöglichen. So ist beispielsweise für triangulierte Graphen (das heißt, dass bei jedem Kreis, der vier oder mehr Knoten enthält zwei Knoten enthält, die im Kreis nicht nebeneinander liegen, aber adjazent zueinander sind) ein Verfahren bekannt, das mit Hilfe so genannter perfekter Eliminationsschemata die optimale Pfadzerlegung in Polynomzeit berechnet. Dieses Verfahren kann auch als Approximationsverfahren für alle anderen Graphen verwendet werden, indem man die Eigenschaft, trianguliert zu sein, durch zusätzliche Kanten herstellt [6].

9 Quellen

Für die Erstellung des C++-Programms zu dieser Arbeit wurden als Hilfsmittel verwendet:

- gcc Compiler für Windows
- Dev-C++ Entwicklungsumgebung für Windows
- GNU Linear Programming Kit (GLPK), vielen Dank an dieser Stelle auch an den Mitarbeiter „GLPK Xypron“ des Projekts GLPK, der mit hilfreichen Hinweisen die Nutzung dieses Pakets ermöglicht hat.

Literatur

[1] Guy Blelloch. Algorithms in 'Real World', lecture 5, 2002.

[2] Christoph Blume. Diplomarbeit Graphsprachen für die Spezifikation von Invarianten bei verteilten und dynamischen Systemen, 2008.

[3] H. L. Bodlaender. A tourist guide through treewidth. Technical Report RUU-CS-92-12, Department of Information and Computing Sciences, Utrecht University, 1992.

[4] Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms*, 18(2):238–255, 1995.

[5] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2):358–402, 1996.

[6] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial Optimization on Graphs of Bounded Treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[7] H.J. Sander Bruggink and Barbara König. On the recognizability of arrow and graph languages. In *Proc. of ICGT '08 (International Conference on Graph Transformation)*, pages 336–350. Springer, 2008. LNCS 5214.

[8] Terry Caelli and Tiberio Caetano. Graphical models for graph matching: approximate models and optimal algorithms. *Pattern Recognition Letters, [S.l*, 26:339–346, 2004.

[9] Kevin Cattell, Michael J. Dinneen, and Michael R. Fellows. A simple linear-time algorithm for finding path-decompositions of small width. In *also University of Victoria manuscript*, pages 05–94, 1996.

[10] Masao Iri. On an extension of the maximum-flow minimum cut theorem to multicommodity flows. *J. Operations Research Soc. of Japan*, 13(3):129–135, 1971.

[11] Ephraim Korach and Nir Solel. Tree-width, path-width, and cutwidth. *Discrete Applied Mathematics*, 43(1):97–101, 1993.

[12] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:422–431, 1988.

[13] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6), 1999.

[14] Benjamin C. Pierce, editor. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[15] Neil Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309 – 322, 1986.

[16] Neil Robertson and P.D. Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39 – 61, 1983.

[17] Farhad Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM (JACM)*, 37(2):318–334, 1990.

[18] Xinlong Zhou. Graphen und Digraphen, 2004.

[19] Xinlong Zhou. Graphenalgorithmen, 2004.

10 Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Duisburg, den 25.10.2010