INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Skript

zur Vorlesung

Effiziente Algorithmen und Datenstrukturen II

von

Prof. A. Steger

im

Sommersemester 1997

Vorläufige Version
24. November 1998

Inhaltsverzeichnis

Kapitel 1

Flüsse in Netzwerken

2./9.Mai Ausarbeitung: *Thomas Schickinger*

1.1 Einführung und Definitionen

In diesem Kapitel betrachen wir sogenannte *Netzwerke* $N = (V, A, c, s, t)$. Hierbei sei

$D = (V, A)$	ein gerichteter Graph,
$s \in V$	die *Quelle* (engl.: source),
$t \in V$	die *Senke* (engl.: target) und
$c : A \to \mathbb{R}_{\geq 0}$	eine nicht-negative *Kapazitätsfunktion.*

Intuitiv stelle man sich unter einem Netzwerk beispielsweise ein Röhrensystem vor, bei dem die Knoten des Graphen den Kreuzungspunkten der Rohre, d. h. der Kanten, entsprechen. Es soll nun z. B. eine Flüssigkeit von der Quelle s zur Senke t transportiert werden. Diese Flüssigkeit darf nur an der Quelle bzw. Senke eingespeist werden bzw. abfließen. An allen anderen Kreuzungspunkten geht keine Flüssigkeit verloren. Jedes Rohr transportiert die Flüssigkeit nur in eine vorgegebene Richtung. Außerdem kann jedes Rohr nur bis zu einer gewissen Obergrenze, der Kapazität, Flüssigkeit aufnehmen. Formal läßt sich dies wie folgt beschreiben:

Definition 1.1 *Eine Abbildung $f : A \to \mathbb{R}$ heißt* s–t–Fluß, *falls*

1. $\sum_{u \in \Gamma^+(v)} f(v, u) = \sum_{u \in \Gamma^-(v)} f(u, v) \quad \forall v \in V \setminus \{s, t\}$ (Flußerhaltung)
2. $0 \leq f(a) \leq c(a) \quad \forall a \in A$ (Zulässigkeit)

Zur Wiederholung sei hier noch einmal erwähnt, daß $\Gamma^+(v)$ die von v aus erreichbaren Nachbarknoten bezeichnet, also $\Gamma^+(v) := \{u \in V : (v, u) \in A\}$. Analog bezeichnet $\Gamma^-(v)$ die Menge der Knoten, von denen aus v direkt erreicht werden kann, also $\Gamma^-(v) := \{u \in V : (u, v) \in A\}$.

Definition 1.2 *Der* Wert $w(f)$ *eines Flusses f ist definiert durch*

$$w(f) = \sum_{u \in \Gamma^+(s)} f(s, u) - \sum_{u \in \Gamma^-(s)} f(u, s)$$

Anschaulich gibt also der Wert an, um wieviel die an der Quelle ins Netzwerk gepumpte Flüssigkeitmenge die dort abfließende Menge übertrifft. Diese Definition ist äquivalent zur Flüssigkeitsmenge, die an der Senke mehr zu- als abfließt:

Proposition 1.3 *Für ein Netzwerk mit gegebenem s–t–Fluß f gilt*

$$w(f) = \sum_{u \in \Gamma^-(t)} f(u, t) - \sum_{u \in \Gamma^+(t)} f(t, u)$$

Beweis: Folgt unmittelbar aus

$$\sum_{v\in V}\sum_{u\in\Gamma^+(v)} f(v,u) = \sum_{v\in V}\sum_{u\in\Gamma^-(v)} f(u,v)$$

durch Umsortieren der Summanden und Anwendung von Teil 1 der Definition 1.1. ∎

Definition 1.4 *Ein s–t–Fluß f_0 mit $w(f_0) = \max\{w(f) : f \text{ ist s–t–Fluß}\}$ heißt* maximaler Fluß.

In den folgenden Abschnitten betrachten wir nun das

MAXFLOW-Problem:

Geg.: Ein Netzwerk mit nicht-negativer Kapazitätsfunktion $N = (V, A, c, s, t)$
Ges.: Bestimme einen maximalen Fluß f.

Diese Fragestellung spielt bei zahlreichen Anwendungen ein Rolle. Man denke hierbei z. B. an ein Transportnetzwerk, bei dem durch optimale Zuordnung der zu transportierenden Güter zu den einzelnen Transportwegen ein maximaler Gesamtdurchsatz erreicht werden soll. Neben solchen direkten Anwendungen tritt das MAXFLOW-Problem aber auch oft als Subproblem anderer Probleme auf[1].

Die nächste Proposition stellt sicher, daß die Suche nach einem maximalen Fluß immer sinnvoll ist.

Proposition 1.5 *In jedem Netzwerk gibt es einen maximalen Fluß.*

Beweis: Betrachte $\mathcal{F} = \{ f : f \text{ ist s–t–Fluß} \} \subset \mathbb{R}^{|A|}$.
Diese Menge ist abgeschlossen und beschränkt. Daraus folgt, daß jede Funktion auf $\mathcal{F}$, also insbesondere auch $f : \mathcal{F} \to \mathbb{R}, f \mapsto w(f)$ auf $\mathcal{F}$ ihr Maximum annimmt. ∎

1.2 Der Algorithmus von Ford–Fulkerson

Diesem Algorithmus liegt folgende einfache Idee zugrunde: Starte mit $f \equiv 0$ und vergrößere $w(f)$ schrittweise, solange dies möglich ist.

Dazu betrachten wir spezielle Pfade im Graphen.

1.2.1 Augmentierende Pfade

Definition 1.6 *Für einen Fluß f in einem Netzwerk N heißt eine Folge $s = v_0, v_1, \ldots, v_{k-1}, v_k = x$ ein* augmentierender $s-x$–Pfad, *falls für alle $i \in \{0, \ldots, k-1\}$ gilt:*

1. $e = (v_i, v_{i+1}) \in A$ *und* $f(e) < c(e)$ *(e heißt* Vorwärtskante*)*
   oder
2. $e = (v_{i+1}, v_i) \in A$ *und* $f(e) > 0$ *(e heißt* Rückwärtskante*)*

Anschaulich besagt dies, daß man entlang eines augmentierenden Pfades mehr Flüssigkeit transportieren kann, als dies im aktuellen Fluß f der Fall ist. Auf den Vorwärtskanten ist die Kapazität noch nicht erschöpft, d. h. dort kann noch mehr „durchgepumpt“ werden. Auf den Rückwärtskanten liegt ein von Null verschiedener Fluß entgegen des augmentierenden Pfades vor, der echt verringert werden kann.

Abbildung 1.1 zeigt ein Beispiel für dieses Vorgehen. Die Beschriftung der Kanten besteht aus Paaren $(c(e), f(e))$. Entlang des gestrichelt markierten augmentierenden Pfades kann der Fluß um eins erhöht werden. Dadurch steigt der Wert des Flusses von eins auf zwei und erreicht dadurch den Maximalwert für diesen Graphen.

Das folgende Lemma präzisiert die Eigenschaften der augmentierenden Pfade.

[1] Siehe hierzu das Kapitel über Zusammenhang von Graphen in diesem Skript.

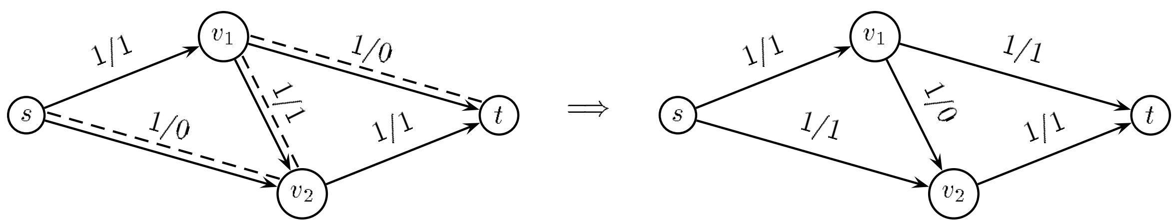


Abbildung 1.1: Erhöhung des Flusses entlang eines augmentierenden Pfades

Lemma 1.7 *Sei f ein s–t–Fluß und $s = v_0, v_1, \dots, v_{k-1}, v_k = t$ ein augmentierender s–t–Pfad P. Dann gilt:*

Für jedes $\delta > 0$ mit

$$\delta \leq \min(\{c(a) - f(a) : a \in P, a \text{ Vorwärtskante}\} \cup \{f(a) : a \in P, a \text{ Rückwärtskante}\})$$

ist

$$f_\delta(a) = \begin{cases} f(a) + \delta & \text{falls } a \in P, a \text{ Vorwärtskante} \\ f(a) - \delta & \text{falls } a \in P, a \text{ Rückwärtskante} \\ f(a) & \text{sonst} \end{cases}$$

ein s–t–Fluß mit Wert $w(f_\delta) = w(f) + \delta$.

Beweis: Zu überprüfen ist die Definition 1.1:
Die Zulässigkeit folgt unmittelbar aus der Definition von f_δ.
Zum Nachweis der Flußerhaltung betrachten wir einen beliebigen Knoten $v \in V \setminus \{s, t\}$:

1. Fall: v liegt nicht auf P. Hier ist nichts zu zeigen.

2. Fall: $v = v_i \in P$ für ein $i \in \{1, \dots, k-1\}$.

Je nachdem, ob es sich bei den an v angrenzenden Kanten auf dem Pfad um Vorwärts- oder Rückwärtskanten handelt, sind vier Fälle zu unterscheiden, die in Abbildung 1.2 dargestellt sind.

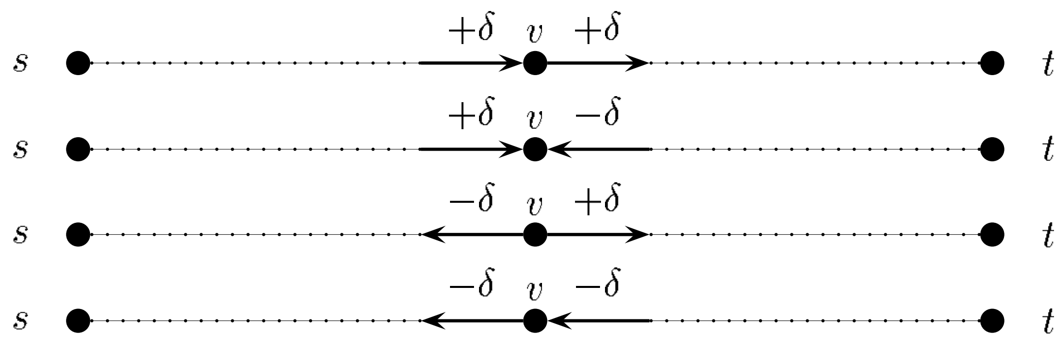


Abbildung 1.2: Verschiedene Fälle bei der Modifikation von f entlang eines augmentierenden Pfades

Offensichtlich bleibt in jedem Fall die Flußerhaltung erfüllt.

$w(f_\delta) = w(f) + \delta$ folgt sofort aus der Veränderung von f auf der Kante (s, v_1). ∎

Die vorangegangenen Überlegungen führen schließlich zu folgendem Algorithmus:
Alg. von Ford–Fulkerson

(1) Setze $f \equiv 0$;

(2) Finde augmentierenden s-t Pfad P bzgl. f;
Falls keiner existiert: STOP, gib f aus;

(3) Setze

$$\delta := \min(\{c(a) - f(a) : a \in P, a \text{ Vorwärtskante}\} \cup \{f(a) \mid a \in P, a \text{Rückwärtskante}\});$$

und

$$f(a) \ := \begin{cases} f(a) + \delta & a \in P, a \text{ Vorwärtskante} \\ f(a) - \delta & a \in P, a \text{ Rückwärtskante} \\ f(a) & \text{sonst.} \end{cases}$$

Zurück zu (2);

1.2.2 Korrektheit für ganzzahlige Netzwerke, Max–Flow–Min–Cut–Theorem

Bevor wir die Korrektheit des Algorithmus von Ford–Fulkerson untersuchen, benötigen wir noch eine neue Definition:

Definition 1.8 *Sei N ein Netzwerk und $W \subset V$ eine Knotenmenge mit $s \in W$ und $t \notin W$. Dann heißt $(W, \overline{W})$ ein* s–t–Schnitt.

Die Kapazität *des Schnitts ist definiert als*

$$c(W, \overline{W}) = \sum_{\substack{(v,u) \in A \\ v \in W, u \notin W}} c(v, u)$$

Zwischen Flüssen und Schnitten besteht ein direkter Zusammenhang:

Proposition 1.9 *Für jeden s–t–Fluß f und jeden s–t–Schnitt $(W, \overline{W})$ gilt:*

$$w(f) = \sum_{\substack{(v,u) \in A \\ v \in W, u \notin W}} f(v, u) \ - \sum_{\substack{(u,v) \in A \\ u \notin W, v \in W}} f(u, v) \ \le \ c(W, \overline{W})$$

Beweis:

$$\begin{aligned} w(f) &= \sum_{u \in \Gamma^+(s)} f(s, u) \ - \sum_{u \in \Gamma^-(s)} f(u, s) \\ &= \sum_{v \in W} \sum_{u \in \Gamma^+(v)} f(v, u) \ - \sum_{v \in W} \sum_{u \in \Gamma^-(v)} f(u, v) \qquad \text{(wegen Flußerhaltung)} \\ &= \sum_{\substack{(v,u) \in A \\ v \in W}} f(v, u) \ - \sum_{\substack{(u,v) \in A \\ v \in W}} f(u, v) \\ &= \sum_{\substack{(v,u) \in A \\ v \in W, u \notin W}} f(v, u) \ - \sum_{\substack{(u,v) \in A \\ u \notin W, v \in W}} f(u, v) \\ &\le \sum_{\substack{(v,u) \in A \\ v \in W, u \notin W}} f(v, u) \ \le \ c(W, \overline{W}) \end{aligned}$$

■

Anschaulich besagt die Proposition 1.9, daß jeder Fluß auf dem Weg von der Quelle zur Senke die Grenze zwischen einer Partition $(W, \overline{W})$ der Knoten mit $s \in W$ vollständig passieren muß. Die Bilanz zwischen dem Fluß aus W hinaus und in W hinein entspricht genau dem Fluß, der an der Quelle entsteht bzw. an der Senke abfließt.

Da die Ungleichung in Proposition 1.9 für alle Flüsse und alle Schnitte gilt, ergibt sich daraus unmittelbar folgendes Korollar:

Korollar 1.10 *Für jedes Netzwerk gilt*

$$\max_{f \text{ } s\text{–}t\text{–}Flu\ss} w(f) \quad \leq \quad \min_{(W,\overline{W}) \text{ } s\text{–}t\text{–}Schnitt} c(W, \overline{W})$$

Nun sind wir in der Lage, die folgende Aussage über die Korrektheit und Laufzeit des Algorithmus von Ford–Fulkerson zu formulieren:

Theorem 1.11 *Für Netzwerke mit einer ganzzahligen Kapazitätsfunktion $c : A \to \mathbb{N}_0$ terminiert der Algorithmus von Ford–Fulkerson nach höchstens $|V|c_{max}$ Augmentierungsschritten mit einem (ganzzahligen) maximalen Fluß, wobei $c_{max} = \max\{c(s) : a \in A\}$.*

Beweis: Lemma 1.7 garantiert, daß der Fluß nach jeder Iteration des Algorithmus zulässig ist. Außerdem ist klar, daß der Fluß immer ganzzahlig ist, d. h., daß $f(e) \in \mathbb{N}_0 \quad \forall e \in A$, da der Nullfluß zu Beginn des Algorithmus diese Bedingung erfüllt und alle Operationen während der Iteration (Minimumsbildung, Addition und Subtraktion) die Ganzzahligkeit erhalten.

Da der Fluß in jedem Iterationsschritt echt erhöht wird und ganzzahlig bleibt, muß er also jedesmal um mindestens eins wachsen. Jede von der Quelle s ausgehende Kante kann also höchstens c_{max}–mal an einem augmentierenden Pfad beteiligt sein. Da jeder augmentierende s–t–Pfad eine solche Kante enthalten muß und s weniger als $|V|$ direkte Nachbarn hat, folgt unmittelbar die Oberschranke für die Anzahl der Iterationen[2].

Es bleibt also nur noch zu zeigen, daß der Fluß f_0, mit dem der Algorithmus terminiert, maximal ist.

Hierzu betrachten wir $W = \{x \in V : \text{ es existiert ein augmentierender s–x–Pfad bezüglich } f_0\}$.

Es gilt $s \in W$ nach Definition und außerdem $t \notin W$, da der Algorithmus sonst einen augmentierenden s–t–Pfad gefunden und nicht terminiert hätte. D. h. $(W, \overline{W})$ ist ein s–t–Schnitt.

Nach Definition von W gilt:

1. $f_0(x, v) = c(x, v) \quad \forall (x, v) \in A$ mit $x \in W, v \notin W$
2. $f_0(u, x) = 0 \quad \forall (u, x) \in A$ mit $u \notin W, x \in W$

Wenn man diese Werte in Proposition 1.9 einsetzt, so erhält man

$$w(f_0) = c(W, \overline{W}) \geq \max_{f \text{ s–t–Fluß}} w(f)$$

wobei sich die Ungleichung aus Korollar 1.10 ergibt. Daraus folgt dann unmittelbar die Behauptung. ■

Aus Theorem 1.11 können wir direkt folgende Verschärfung von Proposition 1.5 ableiten.

Korollar 1.12 *Ist $N = (V, A, C, s, t)$ ein Netzwerk mit ganzzahligen Kapazitäten, so existiert ein ganzzahliger maximaler Fluß.*

Ergänzend wollen wir noch ein bekanntes Resultat der Graphentheorie angeben.

[2] Da uns, wie immer, nur die Größenordnung interessiert, verzichten wir auf weitere Argumente, diese Schranke noch um eins oder zwei zu erniedrigen

Theorem 1.13 (Max–Flow–Min–Cut–Theorem)

Für jedes Netzwerk $N = (V, A, C, s, t)$ gilt:

$$\max_{f \ s\text{–}t\text{–}Fluß} w(f) \quad = \quad \min_{(W,\overline{W}) \ s\text{–}t\text{–}Schnitt} c(W, \overline{W})$$

Beweis: „$\leq$" haben wir in Korollar 1.10 bereits gezeigt. Bleibt also nur noch „$\geq$" nachzuweisen. Betrachte hierzu einen maximalen Fluß f_0, der nach Proposition 1.5 immer existiert. Bezüglich f_0 definiere man eine Menge W_0 genauso wie im Beweis von Theorem 1.11. Genau dieselbe Argumentation wie dort liefert $w(f_0) \geq c(W_0, \overline{W_0})$. ∎

1.2.3 Anmerkungen und Erweiterung auf nichtganzzahlige Netzwerke

Anzahl der Iterationen

Die in Theorem 1.11 angegebene Schranke für die Anzahl der Iterationen ist bis auf eine Konstante bestmöglich. Man betrachte hierzu das Netzwerk in Abbildung 1.3.

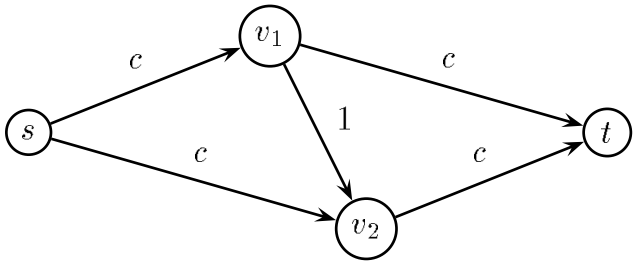


Abbildung 1.3: Worst–case–Beispiel für Ford–Fulkerson

Wenn der Algorithmus abwechselnd die augmentierenden Pfade s, v_1, v_2, t und s, v_2, v_1, t wählt, so sind $2c$ Iterationen nötig. Daraus folgt, daß der Algorithmus von Ford–Fulkerson kein effizienter, d. h. polynomieller Algorithmus ist, da die Kapazitäten mit $\mathcal{O}(\log c_{max})$ Bits kodiert werden können. Wenn man also hinreichend große Kapazitäten zuläßt, so ist die Laufzeit des Algorithmus exponentiell in der Größe der Eingabe.

Dennoch kann der Algorithmus von Ford–Fulkerson unter gewissen Umständen in der Praxis erfolgreich eingesetzt werden. Beispielsweise läßt sich das Problem der Suche nach einem Maximum Matching[3] in einem bipartiten Graphen auf ein MAXFLOW–Problem in einem Netzwerk mit Kapazitäten $c(e) = 1$ transformieren. Dazu erweitert man den bipartiten Graphen um zwei neue Knoten s und t, die als Quelle und Senke des Netzwerks dienen. Von s aus werden Kanten zu allen Knoten einer Partition gezogen und alle Knoten der anderen Partition werden mit t verbunden. Diese neuen Kanten und alle Kanten des ursprünglichen Graphen werden von der Quelle zur Senke gerichtet. Es ist leicht einzusehen, daß genau die Kanten mit einem Fluß gleich eins in der Lösung des MAXFLOW–Problems ein Maximum Matching bilden. Dieses Flußproblem kann der Algorithmus von Ford–Fulkerson in $\mathcal{O}(n)$ Iterationsschritten lösen, wobei $n = |V|$.

[3] Dieser Begriff wird in einem späteren Kapitel genau definiert.

Nichtganzzahlige Kapazitäten: Algorithmus von Edmonds und Karp

Theorem 1.11 läßt sich leicht auf rationale Kapazitätsfunktionen erweitern, indem man die Kapazitäten mit dem Hauptnenner durchmultipliziert und auf diese Weise ein äquivalentes Netzwerk erhält, das die Voraussetzungen von Theorem 1.11 erfüllt.

Für irrationale Kapazitäten kann der Algorithmus von Ford–Fulkerson völlig versagen. Man kann Netzwerke angeben, in denen der Algorithmus nie terminiert und außerdem der Wert des Flusses gegen einen Wert konvergiert, der echt kleiner als der Maximalfluß ist.

Wir suchen deshalb eine Modifikation des Algorithmus von Ford–Fulkerson, der die folgenden zwei Ziele erreicht:

1. Der Algorithmus soll auch für irrationale Kapazitätsfunktionen funktionieren.
2. Die Laufzeit, bzw. die Anzahl der Iterationen soll nur noch von der Größe des Graphen abhängen, nicht aber von der Größe der Kapazitäten.

Bei der Implementierung des Algorithmus von Ford–Fulkerson hat man noch eine wichtige Wahlmöglichkeit, nämlich die Auswahl des augmentierenden Pfades, entlang dem der Fluß erhöht wird. Edmonds und Karp haben gezeigt, daß man durch geschickte Durchführung dieses Schritts die beiden oben genannten Ziele erreichen kann. Ihre Variante des Algorithmus von Ford–Fulkerson wählt in jedem Iterationsschritt einen *kürzesten* (bezüglich der Anzahl von Kanten) augmentierenden Pfad aus. Dieser läßt sich leicht durch Breitensuche bestimmen.

Da der Beweis zur Laufzeit des Algorithmus von Edmonds und Karp kompliziert ist und wir außerdem im nächsten Kapitel einen noch schnelleren Algorithmus betrachten werden, sei hier nur das Ergebnis ohne Beweis angegeben.

Theorem 1.14 *Der Algorithmus von Edmonds und Karp terminiert für jedes Netzwerk nach höchstens $(nm)/2$ Iterationen mit einem maximalen Fluß, wobei $m = |A|$ und $n = |V|$.*

1.3 Die Push-/Relabel–Methode nach Goldberg und Tarjan

Die Idee von Goldberg/Tarjan besteht darin, am Anfang die Quelle „voll aufzudrehen", d. h. auf allen Ausgängen der Quelle schöpft man die Kapazitäten voll aus. Danach wird der Fluß iterativ weiter „Richtung Senke" geleitet (Push–Aktion), soweit es die Kapazitäten und die am jeweiligen Knoten verfügbare Flüssigkeit erlauben. Wenn der Fluß irgendwann bei der Senke angekommen ist und aufgrund der beschränkten Kapazitäten „ins Stocken" gekommen ist, werden eventuell noch vorhandene Überschüsse zurück zur Quelle geleitet und „der Hahn wieder etwas zugedreht".

Für diese Methode benötigen wir die Voraussetzung, daß $D = (V, A)$ symmetrisch ist, d. h. daß $(u, v) \in A \Longrightarrow (v, u) \in A$, damit wir die Flüssigkeit sowohl von der Quelle zur Senke, als auch wieder zurück transportieren können. Dies kann man durch Hinzufügen von Kanten (x, y) mit $c(x, y) = 0$ immer erreichen und stellt somit keine Einschränkung dar.

während der Iterationen des Algorithmus betrachten wir nicht mehr echte Flüsse, sondern Pseudoflüsse bzw. (genauer) Präflüsse. Diese Begriffe werden in den folgenden Definitionen eingeführt.

Definition 1.15 *Eine Abbildung $f : A \to \mathbb{R}$ heißt* Pseudofluß, *wenn gilt:*

1. *$f(u, v) \leq c(u, v)$ für alle $(u, v) \in A$ und*
2. *$f(u, v) = -f(v, u)$ für alle $(u, v) \in A$.*

Definition 1.16 *Gegeben sei ein Netzwerk mit einem Pseudofluß f. Als* Überschuß *(engl.: excess) $e_f(v)$ im Knoten v bezeichnen wir*

$$e_f(v) := \sum_{(u,v) \in A} f(u,v)$$

Hinter dieser einfachen Summe verbirgt sich im Grunde die Bilanz zwischen Zu- und Abfluß im Knoten v. Diese kurze Formulierung wird durch den 2. Teil von Definition 1.15 ermöglicht. Fluß nach v hinein taucht in der Summe als positiver Summand auf und Fluß aus v heraus wird im Gegenzug als negativer Summand berücksichtigt.

Definition 1.17 *Ein* Präfluß *(engl. preflow) ist ein Pseudofluß mit $e_f(v) \geq 0$ für alle $v \in V \setminus \{s,t\}$.*

Zwischen einem Pseudofluß f und einem „echten " Fluß nach Definition 1.1 besteht ein enger Zusammenhang. Wenn man nämlich erreicht, daß alle Überschüsse abgebaut sind, d. h., daß $e_f(v) = 0$ für alle $v \in V \setminus \{s,t\}$, dann kann man leicht einen Fluß f' angeben durch

$$f'(a) = \begin{cases} f(a) & \text{falls } f(a) > 0 \\ 0 & \text{sonst,} \end{cases}$$

Während der Algorithmus überschüssigen Fluß zur Senke oder, wenn deren Aufnahmefähigkeit erschöpft ist, zurück zur Quelle transportiert, betrachten wir die sogenannten aktiven Knoten.

Definition 1.18 *Alle Knoten mit $v \in V \setminus \{s,t\}$ mit $e_f(v) > 0$ bezeichnen wir als* aktive Knoten.

Beim Abbau von überschüssigem Fluß interessieren uns Kanten, über die wir noch Fluß transportieren können.

Definition 1.19 *Die* Restkapazität *$c_f(a)$ einer Kante a ist definiert durch $c_f(a) = c(a) - f(a)$.*

Definition 1.20 *Wenn wir nur Kanten mit positiver Restkapazität betrachten, erhalten wir den* Restgraph *$D_f = (V, A_f)$, wobei $A_f = \{a \in A \mid c_f(a) > 0\}$.*

Definition 1.21 *Gegeben sei ein Netzwerk mit Pseudofluß f. Eine Abbildung $d : V \to \mathbb{N}_0$ mit*

1. *$d(s) = n$, $d(t) = 0$ und*
2. *$d(u) \leq d(v) + 1$ für alle $(u,v) \in A_f$.*

heißt zulässige Markierung.

Lemma 1.22 *Ist f ein Präfluß und d eine zulässige Markierung, dann gibt es in D_f keinen s–t–Pfad.*

Beweis: Wir führen einen Beweis durch Widerspruch. Sei also $s = v_0, \ldots, v_l = t$ ein s–t–Pfad in D_f. Dann ist $(v_{i-1}, v_i) \in A_f$ für alle $i \in \{1, \ldots, l\}$ und $l \leq n - 1$.
Nach Definition gilt für die Markierung:

$$n = d(s) = d(v_0) \leq d(v_1) + 1 \leq (d(v_2) + 1) + 1 \leq \ldots \leq d(v_l) + l = d(t) + l = 0 + l \leq n - 1$$

Damit haben wir den gewünschten Widerspruch erhalten. ∎

Bemerkung 1.23 *Im Beweis von Theorem 1.11 (bei der Definition der Menge W) haben wir bereits folgende Aussage gesehen:*

Wenn f ein Fluß ist, dann gilt: f maximal $\iff$ D_f enthält keinen s–t–Pfad.

Lemma 1.24 *Ist f ein Präfluß und v ein aktiver Knoten, dann gibt es in D_f einen Pfad von v nach s, d. h. Überschuß kann immer zurück zur Quelle transportiert werden.*

Beweis: Sei $W = \{w \in V : w \text{ ist in } D_f \text{ von } v \text{ aus erreichbar}\}$.
Wir nehmen nun an, daß $s \notin W$ und führen dies zum Widerspruch.
Betrachte alle Kanten $(u, w) \in A$ die über den „Rand" der Partition von $\overline{W}$ nach W hinein führen, d. h. mit $u \notin W$ und $w \in W$. Es gilt:

$$(w, u) \notin A_f \quad \Longrightarrow \quad f(w, u) = c(w, u) \geq 0 \quad \Longrightarrow \quad f(u, w) = -f(w, u) \leq 0$$

$$0 \leq \sum_{w \in W} e_f(w) = \sum_{w \in W} \sum_{u \in V} f(u, w) = \underbrace{\sum_{w \in W} \sum_{u \in V \setminus W} f(u, w)}_{\leq 0} + \underbrace{\sum_{w \in W} \sum_{u \in W} f(u, w)}_{=0} \leq 0$$

Das heißt $e_f(w) = 0$ für alle $w \in W$ und damit insbesondere auch $e_f(v) = 0$. Dies widerspricht unserer Annahme, daß v ein aktiver Knoten ist. ∎

15./16.Mai Ausarbeitung: *Alexander Hall*

Der eigentliche Algorithmus sieht folgendermaßen aus:

Basisalgorithmus

// *Initialisierung*
for all $(u, v) \in A$ {
  $f(u, v) := 0$;
  if $u = s$ **then** $f(u, v) := c(u, v)$;
  if $v = s$ **then** $f(u, v) := -c(u, v)$;
}
for all $v \in V$ {
  $e_f(v) = \sum_{w:(w,v) \in A} f(w, v)$;
  if $v = s$ **then** $d(v) := n$ **else** $d(v) := 0$;
}
// *Hauptschleife*
while ($\exists$ aktiver Knoten)
  Führe eine (zulässige) Update-Operation durch;

Update-Prozeduren

PUSH(v, w)
Anwendbarkeit: v aktiv und $d(v) = d(w) + 1$

Sende $\delta := \min\{e_f(v), c_f(v, w)\}$ Flußeinheiten von v nach w;

RELABEL(v)
Anwendbarkeit: v aktiv und $\forall w \in \Gamma^+(v)$ gilt $(v, w) \notin A_f$ oder $d(w) \geq d(v)$

$d(v) := \min\{d(w) + 1 | w \in \Gamma^+(v) \text{ mit } (v, w) \in A_f\}$;

Lemma 1.25 *Invarianten des Basisalgorithmus sind:*

1. *In* RELABEL(v) *wird d(v) um mindestens 1 erhöht.*
2. *f ist immer ein Präfluß und d eine zulässige Markierung.*
3. *Für einen aktiven Knoten v kann immer eine der Operationen* PUSH *oder* RELABEL *ausgeführt werden.*
4. $\mathrm{d}(v) \leq 2n - 1 \qquad \forall v \in V$

Beweis:

1. klar.
2. Wegen der Wahl von δ ist klar, daß f ein Präfluß ist. Mit Induktion über die Anzahl der PUSH / RELABEL Aufrufe zeigen wir, daß d eine zulässige Markierung ist.

   Induktionsbeginn klar.

   Induktionsannahme d ist nach k PUSH / RELABEL Aufrufen eine zulässige Markierung.

   Induktionsschluß $k \leadsto k+1$ Aufrufe. Wir betrachten die $k+1$te Ausführung einer PUSH / RELABEL Prozedur. Hierbei unterscheiden wir, ob dies ein PUSH oder ein RELABEL Aufruf ist.

   RELABEL(v)**:** Es wird nur $\mathrm{d}(v)$ geändert.

   - Für $(w, v) \in A_f$ gilt

   $$\mathrm{d}(w) \overset{\text{Ind. Ann.}}{\leq} \mathrm{d}_{alt}(v) + 1 \overset{\text{Lemma 1}}{\leq} \mathrm{d}_{neu}(v) + 1$$

   - Für $(v, w) \in A_f$ gilt

   $$\mathrm{d}_{neu}(v) \leq \mathrm{d}(w) + 1$$

   nach Konstruktion.

   PUSH(v, w)**:** Es gibt folgende mögliche Auswirkungen:

   - (v, w) wird aus D_f entfernt. Dadurch bleibt d sicherlich eine zulässige Markierung.
   - (w, v) wird zu D_f hinzugefügt. Zu zeigen ist in diesem Fall: $\mathrm{d}(w) \leq \mathrm{d}(v) + 1$. Dies gilt nach Konstruktion, da PUSH(v, w) nur ausgeführt wird, wenn $\mathrm{d}(v) = \mathrm{d}(w) + 1$.

3. Für alle $(v, w) \in A_f$ gilt $\mathrm{d}(v) \leq \mathrm{d}(w) + 1$. Falls PUSH$(v, w)$ nicht anwendbar ist, gilt $\mathrm{d}(v) < \mathrm{d}(w) + 1 \quad \forall (v, w) \in A_f$, d.h. RELABEL$(v)$ ist anwendbar.
4. Nach Lemma 1.24 existiert im Graphen D_f ein Pfad $p = (v_0 = v, v_1, v_2, \ldots, v_l = s)$ von v nach s. D.h.

   $$\mathrm{d}(v) = \mathrm{d}(v_0) \leq \mathrm{d}(v_1) + 1 \leq \cdots \leq \mathrm{d}(v_l) + l = \underbrace{\mathrm{d}(s)}_{=n} + \underbrace{l}_{\leq n-1} \leq 2n - 1$$

■

Als unmittelbare Folgerung erhalten wir aus Lemma 1.25:

Korollar 1.26 *Der Basisalgorithmus terminiert mit einem maximalen Fluß.* ■

Als nächstes wollen wir noch die maximale Laufzeit des Basisalgorithmus abschätzen. Dazu werden wir zunächst Aussagen über die maximal mögliche Anzahl von PUSH bzw. RELABEL Aufrufen herleiten.

Lemma 1.27 *Die Anzahl der Relabelings ist kleiner gleich* $2n^2$.

Beweis: Pro Knoten $v \in V\backslash\{s,t\}$ kommen höchstens $2n-1$ Relabelings vor $\Rightarrow$ Insgesamt sind es maximal $(n-2)\cdot(2n-1) \leq 2n^2$. ∎

Es soll nun auch eine obere Schranke für die Anzahl der Pushs gefunden werden. Dazu unterscheidet man zwischen saturierten und nicht saturierten Pushs. Erstere schöpfen die Restkapazität einer Kante voll aus, letztere nicht.

Lemma 1.28 *Die Anzahl der saturierten Pushs ist kleiner gleich $n \cdot m$.*

Beweis: Wir betrachten die Kante $(v,w) \in A$. Ein saturierter Push $v \to w$ tritt auf, wenn $\mathrm{d}(v)$ gleich $\mathrm{d}(w)+1$ ist. Anschließend gilt: $(v,w) \notin A_f$. Vor dem nächsten Push von v nach w muß daher ein Push von w nach v erfolgen., d.h. $\mathrm{d}(w)$ muß mindestens um 2 erhöht werden und anschließend muß $\mathrm{d}(v)$ um mindestens 2 erhöht werden. Daher kann es höchstens n saturierte Pushs $v \to w$ geben. ∎

Lemma 1.29 *Die Anzahl der nicht saturierten Pushs ist kleiner gleich $2n^2m$.*

Beweis: Wir betrachten das Potential $\phi := \sum_{v:\ aktiv} \mathrm{d}(v)$. Wie verändert sich das Potential während eines Push bzw. Relabel Aufrufs? Wir unterscheiden hierfür drei Fälle:

nicht saturierter Push v ist danach nicht mehr aktiv, da durch den Push der gesamte Überschuß abtransportiert wurde – die Kapazität der Kante wurde ja nicht voll genutzt. w wird unter Umständen aktiv, falls es nicht vorher schon Überschuß hatte $\Rightarrow \phi$ nimmt um mindestens $\mathrm{d}(v)-\mathrm{d}(w)=1$ *ab*.

saturierter Push Falls vor der Ausführung des Pushs bereits $e_f(w)>0$ galt, ändert sich am Potential durch den Push gar nichts. Andernfalls wird w aktiv $\Rightarrow \phi$ nimmt um höchstens $\mathrm{d}(w)=\mathrm{d}(v)-1 \leq 2n-2$ zu $\rightsquigarrow$ insgesamt nimmt ϕ bei saturierten Pushs um höchstens #saturierter Pushs $\cdot(2n-2) \leq nm\cdot(2n-2)$ *zu*.

Relabel Wir ermitteln die Änderung des Potentials über alle r Relabels hinweg. Sei $\mathrm{d}_i(v)\quad \forall\ i = 0\ldots r,\ \forall\ v \in V' = V\setminus\{s,t\}$ die Markierung von v nach dem i-ten Relabel. Bei dem i-ten Relabel ändert sich das Potential um folgenden Wert:

$$\delta_i = \sum_{v:\ aktiv} \mathrm{d}_i(v) - \sum_{v:\ aktiv} \mathrm{d}_{i-1}(v)$$

Bei allen unaktiven Knoten ändert sich nichts, damit ist

$$\delta_i = \sum_{v\in V'} \mathrm{d}_i(v) - \sum_{v\in V'} \mathrm{d}_{i-1}(v)$$

Nun summieren wir über alle Relabels auf (Anfangs sind alle Markierungen in V' gleich 0):

$$\sum_{i=1}^{r}\delta_i = \sum_{i=1}^{r}\left(\sum_{v\in V'} \mathrm{d}_i(v) - \sum_{v\in V'} \mathrm{d}_{i-1}(v)\right) = \sum_{v\in V'} \mathrm{d}_r(v) \leq 2n^2$$

Durch Relabels nimmt das Potential also insgesamt höchstens um $2n^2$ zu.

$\Rightarrow$ das Potential nimmt insgesamt maximal um $nm\cdot(2n-2)+2n^2 \leq 2n^2m$ $(m \geq n)$ durch Relabels und saturierte Pushs zu. Da das Potential andererseits bei jedem nicht saturiertem Push um mindestens eins abnimmt, ist die Anzahl der nicht saturierten Pushs somit kleiner gleich $2n^2m$. ∎

Für eine effiziente Implementierung des Basisalgorithmus benötigen wir nun noch ein möglichst geschicktes Verfahren für die Auswahl der nächsten Update-Operation in der Hauptschleife. Dazu führen wir noch eine weitere Prozedur ein:

Push/Relabel(v). Die Prozedur Push/Relabel(v) führt für einen aktiven Knoten v nach Möglichkeit eine Push, ansonsten eine Relabel Operation durch. Dazu gibt es für jeden Knoten eine Liste der adjazenten Kanten und eine Referenz *current edge* auf eine dieser Kanten. Zu Beginn ($\rightsquigarrow$ Initialisierung) zeige *current edge* auf die erste Kante in der Adjazenzliste.

PUSH/RELABEL(v)

Sei $\{v, w\}$ die derzeitige *current edge* von v;
if (PUSH(v) anwendbar)
  PUSH(v);
else {
  if ($\{v, w\}$ letzte Kante der Kantenliste von v) {
    Mache erste Kante der Kantenliste zur *current edge*;
    RELABEL(v);
  }
  else {
    Ersetze *current edge* durch die nächte Kante der Kantenliste;
    weiter in der ersten Zeile;
  }
}

Lemma 1.30 *In* PUSH/RELABEL(v) *wird* RELABEL(v) *nur ausgeführt, wenn es auch erlaubt ist.*

Beweis: Für alle Kanten oberhalb der *current edge* gilt zu jedem Zeitpunkt $c_f(v, w) = 0$ oder $\mathrm{d}(v) < \mathrm{d}(w) + 1$ ∎

Theorem 1.31 *Der Basisalgorithmus mit der* PUSH/RELABEL *Prozedur benötigt Laufzeit* $\mathcal{O}(n^2 m)$.

Beweis: Ein Relabel hat die Laufzeit $\mathcal{O}(\deg(v))$. An jedem Knoten werden wegen Lemma 1.25 maximal $2n$ Relabels durchgeführt $\Rightarrow$ die Gesamtlaufzeit für alle Relabels ist $\mathcal{O}(\sum_v n \cdot \deg(v)) = \mathcal{O}(nm)$ ($\sum_v \deg(v) = 2m$). Ein Push hat die Laufzeit $\mathcal{O}(1)$, wegen Lemma 1.28 und 1.29 haben die Push Operationen insgesamt eine Laufzeit von $\mathcal{O}(n^2 m)$. Das Abarbeiten der Kantenliste in der Push/Relabel Routine geschieht höchstens $2n$ mal pro Knoten, da es maximal so viele Relabels pro Knoten gibt. Damit hat die Routine eine Gesamtlaufzeit von $\mathcal{O}(\sum_v n \cdot \deg(v)) = \mathcal{O}(nm)$.

Es ergibt sich folgende Gesamtlaufzeit des Basisalgorithmus:

$$\mathcal{O}(nm) + \mathcal{O}(n^2 m) + \mathcal{O}(nm) = \mathcal{O}(n^2 m)$$

∎

Durch eine geschickte Auswahl des nächsten mit PUSH/RELABEL zu bearbeitenden aktiven Knoten, kann man die Anzahl der nicht saturierten Pushs reduzieren. Diese sind mit $\mathcal{O}(n^2 m)$ die größten Laufzeitfresser. Es gibt mehrere Auswahlstrategien, die einfachste (FIFO) werden wir näher betrachten. Bei der FIFO-Strategie werden die aktiven Knoten in einer Queue nach dem *first-in first-out* Prinzip gehalten.

Basisalgorithmus mit FIFO-Strategie

// Initialisierung
... wie bisher ...
Füge zusätzliche alle aktiven Knoten in eine Queue Q ein;
// Hauptschleife
while (Q nicht leer) {
  Entferne den vordersten Knoten v von Q;
  repeat {
    PUSH/RELABEL(v);

Falls dadurch ein Knoten w aktiv wird, füge ihn am Ende von Q an;
} **until** ($e_f(v) = 0$ oder $d(v)$ hat zugenommen)
Falls v noch immer aktiv ist, füge v am Ende von Q an;
}

Theorem 1.32 *Der Basislgorithmus mit* PUSH/RELABEL *und FIFO-Strategie hat Laufzeit* $\mathcal{O}(n^3)$.

Beweis: Wir unterteilen den Algorithmus in Phasen. In der ersten Phase werden alle Knoten abgearbeitet, die wärend der Initialisierung aktiv wurden. In Phase $i+1$ werden die in Phase i eingefügten Knoten abgearbeitet. Aus der Queue wird ja vorne immer ein Knoten entfernt und bearbeitet und die dadurch neu aktiv gewordenen Knoten werden hinten eingehängt.

Wir überlegen uns zunächst, daß es maximal $4n^2$ Phasen gibt. Dazu betrachten wir das folgende Potential: $\phi = \max\{\mathrm{d}(v)|v \text{ aktiv}\}$. Zu Beginn und am Ende ist $\phi = 0$.

Zunächst zeigen wir, daß in einer Phase ohne Relabeling ϕ um mindestens eins abnimmt. Dies sieht man folgendermaßen ein: Für alle Knoten w, die durch einen $\mathrm{Push}(v, w)$ aktiv werden, gilt $\mathrm{d}(w) = \mathrm{d}(v) - 1$. D.h., jeder Knoten der in der Phase aktiv *wird* hat einen d-Wert, der um eins *kleiner* ist als der d-Wert des Knotens, der den Push initiiert. Da nach Annahme alle zu Beginn der Phase aktiven Knoten am Ende der Phase nicht mehr aktiv sind, nimmt ϕ, das ja das Maximum der d-Werte der aktiven Knoten ist, um mindestens eins ab.

Die Zunahme von ϕ ist beschränkt durch die Anzahl der Relabel-Operationen, wir schätzen wieder ähnlich wie bei Lemma 1.29 ab. Nach dem i-ten Relabel (davor und danach sind dieselben Knoten aktiv) nimmt das Potential um

$$\delta_i = \max\{\mathrm{d}_i(v)|v \text{ aktiv}\} - \max\{\mathrm{d}_{i-1}(v)|v \text{ aktiv}\} \leq \sum_{v \in V'} \mathrm{d}_i(v) - \sum_{v \in V'} \mathrm{d}_{i-1}(v)$$

zu, mit d_i und V' wie bei Lemma 1.29. Insgesamt ergibt sich somit eine Änderung von

$$\sum_{i=1}^{r} \delta_i \leq \sum_{i=1}^{r} \left(\sum_{v \in V'} \mathrm{d}_i(v) - \sum_{v \in V'} \mathrm{d}_{i-1}(v) \right) = \sum_{v \in V'} \mathrm{d}_r(v) \leq 2n^2$$

D.h. die gesamte Zunahme von ϕ ist maximal $2n^2$
$\Rightarrow$ es gibt höchstens $4n^2$ Phasen.
Zurück zur Laufzeit:

1. Zeit für Durchlaufen der Kantenlisten in PUSH/RELABEL: Die Adjazenzliste eines jeden Knotens wird zwischen zwei Relabel-Aufrufen genau einmal durchlaufen. Da die Anzahl der Relabel-Aufrufe durch $2n-1$ begrenzt ist, ergibt sich als Gesamtlaufzeit für das Durchlaufen der Kantenlisten in PUSH/RELABEL: $\mathcal{O}(n \cdot \sum_{v \in V} \deg(v)) = \mathcal{O}(nm)$.

2. Zeit für Relabels: $\mathcal{O}(nm)$, s.o.

3. Zeit für saturierte Pushs: $\mathcal{O}(nm)$, s.o.

4. Zeit für nicht saturierte Pushs: $\mathcal{O}(n^3)$, da höchstens ein nicht saturierter Push pro Knoten pro Phase auftritt und es maximal $4n^2$ Phasen gibt.

■

Außer der FIFO-Strategie gibt es noch weitere Auswahlstrategien, beispielsweise:

- HIGHEST LABEL: es wird jeweils der aktive Knoten mit maximalem d-Wert betrachtet. Laufzeit: $\mathcal{O}(n^2\sqrt{m})$

- EXCESS SCALING: es wird nur Fluß von Knoten mit „genügend“ großem Überschuß zu Knoten mit genügend kleinem Überschuß geschickt. Laufzeit: $\mathcal{O}(nm + n^2 \log c_{max})$

Die Analyse dieser Strategien ist etwas aufwendiger. Wir wollen daher an dieser Stelle darauf verzichten. Der interessierte Leser sei auf [AMO93] verwiesen.

Kapitel 2

Zusammenhang

Definition 2.1 *Eine Menge von s-t Pfaden heißt ...*

- *... kreuzungsfrei, falls je zwei der Pfade genau die Knoten s und t gemeinsam haben (siehe Abbildung 2.1).*

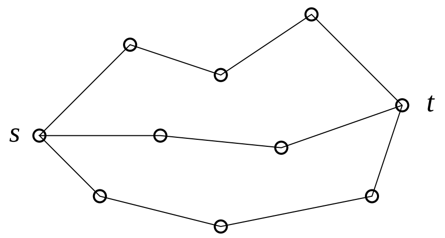


Abbildung 2.1: Beispiel mit kreuzungsfreien Pfaden

- *... kantendisjunkt, falls keine Kante in mehr als einem Pfad vorkommt (siehe Abbildung 2.2).*

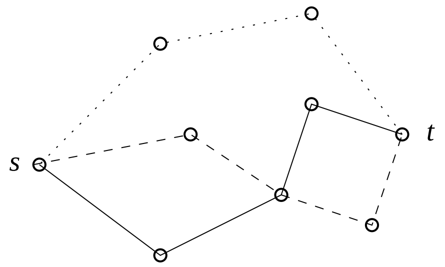


Abbildung 2.2: Beispiel mit kantendisjunkten Pfaden

Definition 2.2 *Ein Graph G auf mehr als $k+1$ Knoten heißt k-*fach zusammenhängend*, falls es zwischen je zwei Knoten s und t in G mindestens k kreuzungsfreie s-t Pfade gibt.*

*Ein Graph G auf mehr als $k+1$ Knoten heißt k-*fach Kanten-zusammenhängend*, falls es zwischen je zwei Knoten s und t in G mindestens k kantendisjunkte s-t Pfade gibt.*

In Abbildung 2.3 sind drei Beispiele zu sehen.

Ergänzend führen wir noch die folgenden beiden Größen ein:

$$\begin{aligned} \kappa(G) &:= \max\{k|G\ k\text{−fach zusammenh.}\} \\ \lambda(G) &:= \max\{k|G\ k\text{−fach Kanten} - \text{zusammenh.}\} \end{aligned}$$

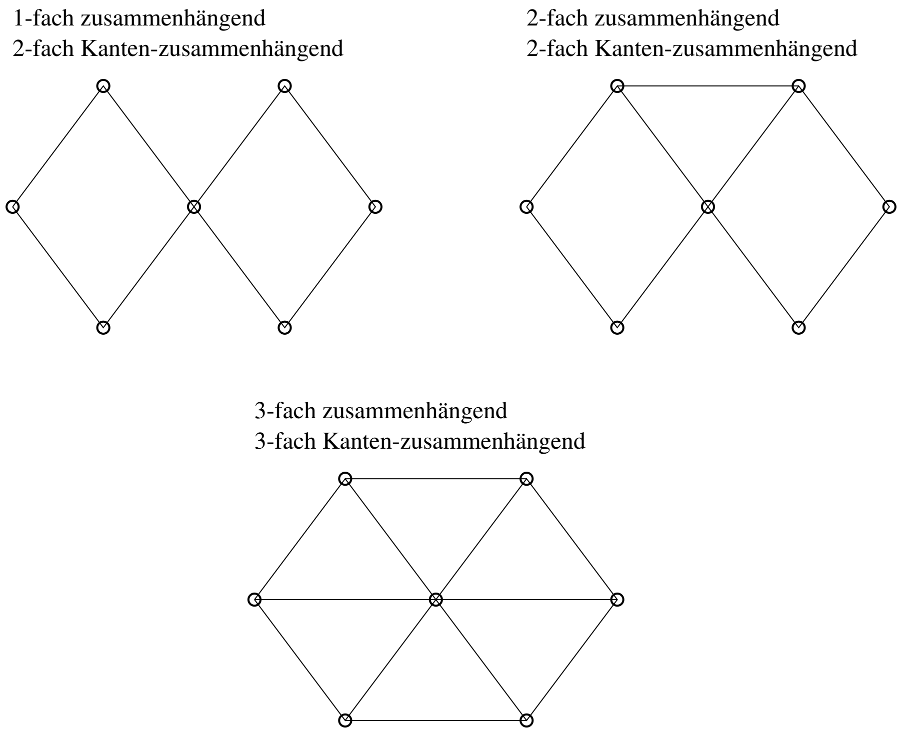


Abbildung 2.3: Drei Graphen mit jeweils 7 Knoten und unterschiedlichen (Kanten-) Zusammenhängen

2.1 Die Sätze von Menger und Whitney

Theorem 2.3 (Menger 1927) *Für alle Graphen $G = (V, E)$ gilt:*

Kantenversion $\forall s, t \in V$ *gilt: Die maximale Anzahl kantendisjunkter s-t Pfade entspricht („=“) der minimalen Kardinalität einer s-t trennenden Kantenmenge.*

Knotenversion $\forall s, t \in V$ *mit* $\{s, t\} \notin E$ *gilt: Die maximale Anzahl kreuzungsfreier s-t Pfade entspricht („=“) der minimalen Kardinalität einer s-t trennenden Knotenmenge.*

Beweis:

Kantenversion

1. „$\leq$“: klar, jeder der Pfade enthält eine andere Kante aus der trennenden Kantenmenge.
2. „$\geq$“: Wir wollen das MaxFlow/MinCut Theorem anwenden. Dazu konstruieren wir das Netzwerk $N = (V, A, c; s, t)$ wie folgt:
   - Alle mit s verbundenen Kanten von s wegrichten.
   - Alle mit t verbundenen Kanten zu t hinrichten.
   - Alle übrigen Kanten $\{u, v\}$ durch ein Paar entgegengesetzt gerichteter Kanten $\{u, v\}$ und $\{v, u\}$ ersetzen.
   - c auf 1 setzen.

   Dann gilt:

   - Jeder s-t Schnitt mit Kapazität ξ entspricht einer s-t trennenden Kantenmenge in G der Kardinalität ξ.
   - Der maximale Fluß ist ganzzahlig (siehe Theorem 1.12). Ein Fluß mit Wert ξ in N entspricht einer Menge von ξ kantendisjunkten Pfaden in G, da der Fluß von den Kantenpaaren immer nur eine der beiden Kanten benutzt.

Knotenversion Der Beweis verläuft ähnlich wie bei der Kantenversion, hier soll nur die Grundidee vermittelt werden: Im ersten Schritt wird wie oben ein Netzwerk N erstellt. Anschließend werden alle Knoten durch zwei Knoten ersetzt, die mit einer „K-Kante“ verbunden werden (siehe Abbildung 2.4). Es ist zu zeigen, daß der minimale Schnitt nur K-Kanten enthält.

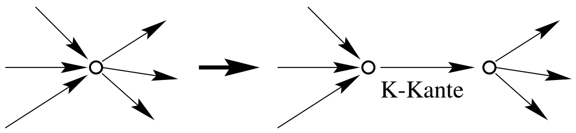


Abbildung 2.4: Jeder Knoten wird durch zwei Knoten und eine K-Kante ersetzt. Der linke Knoten erhält alle ursprünglichen Eingangskanten, der rechte alle ursprünglichen Ausgangskanten

■

Theorem 2.4 (Whitney 1932)

Kantenversion *Für jeden Graphen G sind äquivalent:*

1. *G ist k-fach Kanten-zusammenhängend.*
2. *Jede trennende Kantenmenge hat mindestens die Kardinalität k.*

Knotenversion *Für jeden Graphen G auf mindestens $k+1$ Knoten, der nicht vollständig ist, sind äquivalent:*

1. *G ist k-fach zusammenhängend.*
2. *Jede trennende Knotenmenge hat mindestens die Kardinalität k.*

Beweis: Die Kantenversion folgt sofort aus dem vorher Gesagten. Der Beweis der Knotenversion sei dem interessiertem Leser als Übungsaufgabe überlassen. ■

Verbinden die beiden Sätze von Menger un Whitney, so erhalten wir, daß sich die Knoten- und Kantenzusammenhangszahlen effizient berechnen lassen:

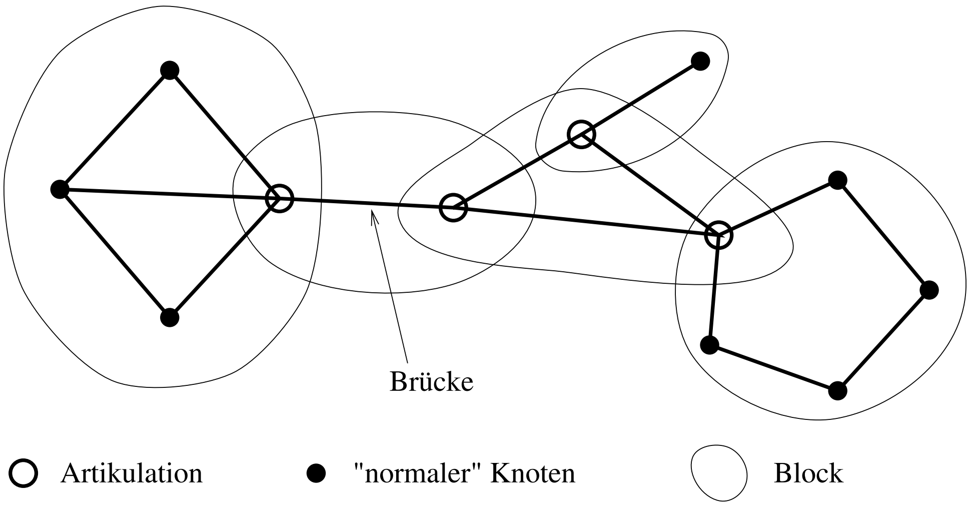
Korollar 2.5 *$\lambda(G)$ und $\kappa(G)$ lassen sich in $\mathcal{O}(n^2m)$ berechnen.*

Beweis: Zur Berechnung sind $\mathcal{O}(n)$ Flußprobleme in Netzwerken mit Kapazität 1 zu lösen. Wir wählen einen Knoten s beliebig und bestimmen den maximalen Fluß bzw. den minimalen Schnitt für alle $t \in V \setminus \{s\}$. Es genügt den Knoten s fest zu wählen, da er sich auf jeden Fall auf einer der beiden Seiten des insgesamt minimalen Schnitts befindet. Zur Berechnung des Flusses wird mit Ford-Fulkerson beispielsweise jeweils die Laufzeit $\mathcal{O}(nm)$ benötigt. ■

2.2 2-Zusammenhang und Blockzerlegung

22./23.Mai Ausarbeitung: *Günther Donderer*

Motivation: Was ist eine ,,vernünftige” Zerlegung eines Graphen? — Intuitiv wird man sich darunter eine Partition der Knotenmenge des Graphen vorstellen, so daß die verschiedenen Teile “möglichst wenig” miteinander zu tun haben. Schauen wir uns ein Beispiel an:



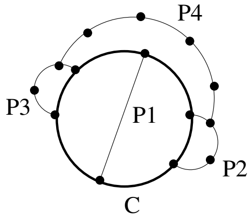
Durch die skizzierte Zerlegung entsteht ein ,,Baum", der aus Komponenten besteht, die entweder 2-fach zusammenhängend oder Kanten sind.

Bemerkung 2.6 *Falls solch eine Zerlegung bekannt ist, dann können Algorithmen oft auf die Blöcke einzeln angewandt werden. Ein Beispiel dafür ist das Minimal Spanning Tree Problem. Weitere Beispiele werden später noch vorgeführt.*

Zunächst folgen ein paar Struktur-Resultate über 2-fach zusammenhängende Graphen.

Theorem 2.7 *(Ohrenzerlegung)*

G ist 2-fach zusammenhängend $\Leftrightarrow G = C \cup P_1 \cup \cdots \cup P_k$, $k \geq 0$, wobei C ein Kreis ist und P_i Pfade sind, die genau die beiden Endpunkte mit $C \cup P_1 \cup \cdots \cup P_{i-1}$ gemeinsam haben.



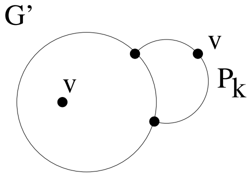
Beweis:

,,$\Leftarrow$": Induktion über k

Anfang ($k = 0$): G ist eine Kreis und daher 2-fach zusammenhängend. $\surd$

Schritt ($k - 1 \Rightarrow k$):

$G' := C \cup P_1 \cup \cdots \cup P_{k-1}$ ist 2-fach zusammenhängend nach der Induktions-Annahme. Wäre $G' \cup P_k$ nicht 2-fach zusammenhängend, dann gäbe es einen trennenden Knoten v.



1. Fall: $v \in G'$

   $\Rightarrow$ v ist insbesondere trennender Knoten von G'

   $\Rightarrow$ Widerspruch zur Annahme, daß G' 2-fach zusammenhängend sei

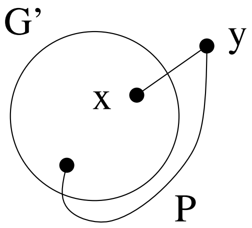
2. Fall: $v \notin G'$

   Da G' zusammenhängend ist, ist auch $G - v$ zusammenhängend.

   $\Rightarrow$ Widerspruch dazu, daß v trennender Knoten sei

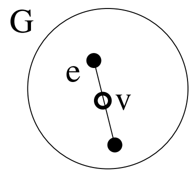
„$\Rightarrow$": Der folgende Algorithmus findet eine Ohrenzerlegung.

Wähle beliebigen Kreis C in G !
// existiert, weil G 2-fach zusammenhängend ist
$G' := C$, $k := 0$
while $G' \neq G$
  Wähle Kante $\{x, y\} \in E(G) \setminus E(G')$, die mit einem Knoten x in G' inzident ist
  // existiert, weil G zusammenhängend ist
  if $y \in G'$ **then** $P_{k+1} = \{x, y\}$
  else
    Bestimme Pfad P in $G - x$, der y mit $V(G') \setminus \{x\}$ verbindet
    // existiert, weil G 2-fach zusammenhängend ist
    Setze $P_{k+1} = P \cup \{x, y\}$
  $k++$;



Aus der Korrektheit des Algorithmus folgt, daß auch die Vorwärtsrichtung der Behauptung stimmt. ∎

Korollar 2.8 *Sei $G = (V, E)$ ein 2-fach zusammenhängender Graph. Dann gilt: Fügt man auf einer beliebigen Kante $e \in E$ einen Knoten v ein, so bleibt G 2-fach zusammenhängend.*



Beweis: Eine Ohrenzerlegung bleibt eine Ohrenzerlegung. :-) ∎

Lemma 2.9 *Sei $G = (V, E)$ ein Graph ohne isolierte Knoten. (D. h., $deg(v) \geq 1 \ \forall \ v \in V$.) Dann sind äquivalent:*

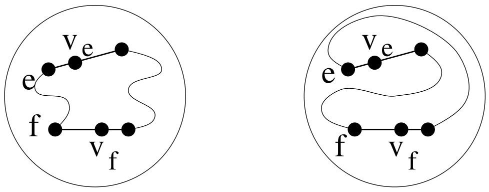
1. *G ist 2-fach zusammenhängend*

2. $\forall\ e, f \in E$ gilt: $\exists$ Kreis, der e und f enthält

3. $\forall\ u, v \in V$ gilt: $\exists$ Kreis, der u und v enthält

Beweis:
„1. $\Rightarrow$ 2.“:

Füge auf e und f zusätzliche Knoten v_e bzw. v_f ein. Der neue Graph ist nach Korollar 2.8 2-fach zusammenhängend. D. h., er enthält zwei kreuzungsfreie $v_e - v_f$ Pfade. Diese bilden den gesuchten Kreis.



„2. $\Rightarrow$ 3.“: Wähle jeweils eine beliebige mit u bzw. v inzidente Kante.
„3. $\Rightarrow$ 1.“: Kreis $\equiv$ zwei kreuzungsfreie Pfade ∎

Definition 2.10 *Sei G ein zusammenhängender Graph.*

Ein Knoten $v \in V$ heißt **Artikulation** *(oder* **trennender Knoten***), falls $G - v$ nicht zusammenhängend ist.*

Definition 2.11 *Ein* **Block** *in einem Graphen G ist ein maximaler zusammenhängender Subgraph von G auf mindestens zwei Knoten, der keinen trennenden Knoten enthält.*

Bemerkung 2.12 *Offenbar gilt folgendes.*

a) Jede Kante ist in einem Block enthalten. Ggf. bildet sie einen eigenen Block.

b) Ein Block besteht entweder aus genau einer Kante, oder er ist 2-fach zusammenhängend.

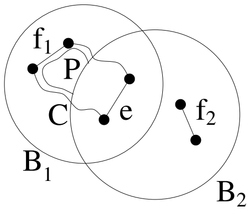
c) Ein Kreis ist stets ganz in einem Block enthalten.

Lemma 2.13 *Die Blöcke bilden eine Partition der Kantenmenge. (D. h., jede Kante ist in genau einem Block enthalten.)*

Beweis:

Annahme: Es gibt eine Kante $e \in E$, die in zwei verschiedenen Blöcken B_1 und B_2 enthalten ist.

Wegen der Maximalitäten enthält jeder der beiden Blöcke B_i eine Kante f_i, die *nicht* im anderen Block enthalten ist.
$\Rightarrow$ B_1 besteht aus mehr als einer Kante, ist daher (Bemerkung 2.12b) 2-fach zusammenhängend und enthält daher (Lemma 2.9) einen Kreis C, der e und f_1 enthält.
Wegen $f_1 \notin B_2$ gibt es einen Teilpfad P von C, dessen Endknoten zu B_2 gehören, aber *nicht* dessen innere Knoten und Kante. Nach Ohrenzerlegung (Theorem 2.7) ist $B_2 \cup P$ 2-fach zusammenhängend.



Das steht im Widerspruch zur Maximalität von B_2. ∎

Lemma 2.14 *(gemeinsame Knoten zweier verschiedener Blöcke)*

a) Zwei verschiedene Blöcke eines Graphen haben höchstens einen *Knoten gemeinsam.*

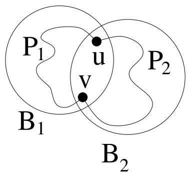
b) Ein Knoten gehört genau dann zu mehreren Blöcken, wenn er eine Artikulation ist.

Beweis:

a) Annahme: $u, v \in V$ sind beide in zwei verschiedenen Blöcken B_1 und B_2 enthalten.

Für $i = 1, 2$ gilt dann jeweils: In B_i existiert ein $u - v$ Pfad P_i.

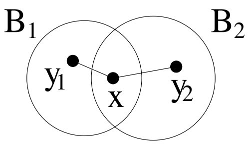
P_1, P_2 sind nach Lemma 2.13 kantendisjunkt. D. h., $P_1 \cup P_2$ enthält einen Kreis, der nicht ganz in einem Block enthalten ist.



Das steht im Widerspruch zu Bemerkung 2.12c.

b) „⇒": Sei $x \in V$ ein Knoten, der in mindestens zwei Blöcken B_1 und B_2 enthalten ist. Zu zeigen ist, daß x eine Artikulation ist.

Wähle Nachbarn y_i mit $\{x, y_i\} \in B_i$.



Wäre x keine Artikulation, so gäbe es in $G - x$ einen $y_1 - y_2$ Pfad. Zusammen mit den Kanten $\{x, y_1\}$ und $\{x, y_2\}$ gäbe es also einen Kreis, der Kanten aus zwei verschiedenen Blöcken enthält. Das steht aber im Widerspruch zu Bemerkung 2.12c.

„⇐": Sei x eine Artikulation.

Dann ist $G - x$ nicht zusammenhängend. D. h., es gibt Nachbarn y_1 und y_2 von x, so daß jeder $y_1 - y_2$ Pfad x enthält. D. h., $\{x, y_1\}$ und $\{x, y_2\}$ liegen auf keinem gemeinsamen Kreis und gehören daher zu verschiedenen Blöcken.

Damit ist die Gültigkeit der beiden Aussagen aus Lemma 2.14 bewiesen. ∎

Bemerkung 2.15 *Wir haben gezeigt, daß die Artikulationen genau die Schnittstellen zwischen den Blöcken eines Graphen bilden.*

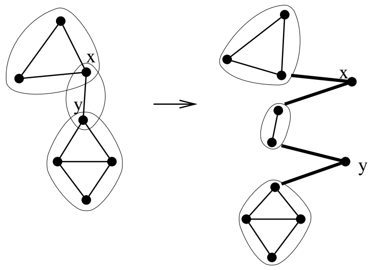
Definition 2.16 *Sei G ein Graph.*

$\mathfrak{B} = \{B \mid B \text{ Block in } G\}$

$\mathfrak{A} = \{a \mid a \text{ Artikulation in } G\}$

Der Block-Artikulationsgraph $BA(G)$ ist ein Graph mit Knotenmenge $\mathfrak{A} \cup \mathfrak{B}$ und Kantenmenge $\{\{a, B\} \mid B \in \mathfrak{B},\ a \in \mathfrak{A},\ a \in B\}$.

Beispiel 2.17 *Hier folgen ein Beispiel-Graph und sein zugehöriger Block-Artikulationsgraph.*

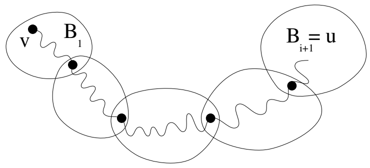


Theorem 2.18 *Der Block-Artikulations-Graph $BA(G)$ eines zusammenhängenden Graphen ist ein Baum.*

Beweis: Zwei Dinge sind zu zeigen:

- Der Graph ist zusammenhängend, d. h., $\exists$ einen Pfad in $BA(G)$ zwischen je zwei verschiedenen Knoten v und u.

  Aus der Konstruktion von $BA(G)$ sieht man, daß es genau dann einen solchen Pfad gibt, wenn es eine Folge $B_1, \ldots, B_k$ von Blöcken in G gibt, wobei jeweils zwei aufeinanderfolgende Blöcke B_i und B_{i+1} eine gemeinsame Artikulation enthalten.

  Dabei müssen B_1 und B_k so gewählt werden, daß sie den Knoten v und u ,,entsprechen", d. h.

  a) $v = B_1$ bzw. $u = B_k$, falls v bzw. u ein Block in G ist, und

  b) $v \in B_1$ bzw. $u \in B_k$, falls v bzw. u eine Artikulation in G ist.

  Weil G zusammenhängend ist und weil die Blöcke in G durch gemeinsame Artikulationen verbunden sind, gibt es zu beliebigen Blöcken B_1 und B_k (der Wert von k ergibt sich erst) eine entsprechende Blockfolge wie oben beschrieben. Damit gibt es aber auch die gesuchten Pfade in $BA(G)$.

- Der Graph ist kreisfrei:

  Aus Kreisen in $BA(G)$ erhält man Kreise in G. Das steht im Widerspruch zu Bemerkung 2.12c.

Also ist $BA(G)$ ein Baum. ∎

Nachdem wir jetzt ein paar Eigenschaften von Blöcken und Artikulationen gezeigt haben, bleibt noch die Frage, wie man sie möglichst effizient bestimmt.

Aufgabe: Bestimme Blöcke und Artikulationen eines Graphen G ! (Ohne Einschränkung betrachten wir im folgenden nur zusammenhängende Graphen G.)

trivialer Algorithmus:

Teste $\forall\ v \in V$: Ist v Artikulation?
Falls ja, dann zerlege G!

Laufzeit: $O(n \cdot m)$

Ziel: Algorithmus mit Laufzeit $O(m)$

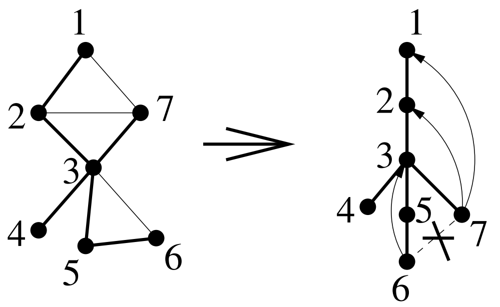
Vorgehensweise: Modifiziere Tiefensuche! Dabei verwenden wir folgende Datenstruktur.

dfs[v] :=<als wievielter Knoten v im DFS-Algorithmus besucht wurde>

pred[v] :=<Vorgänger von v im DFS-Baum>

Beobachtung: Kanten des Graphen zerfallen in zwei Klassen.

- Baumkanten: $\{pred[v], v\}$
- Rückwärtskanten: $\{x, y\}$, wobei y auf dem $x - s$ Pfad von x zur Wurzel s liegt



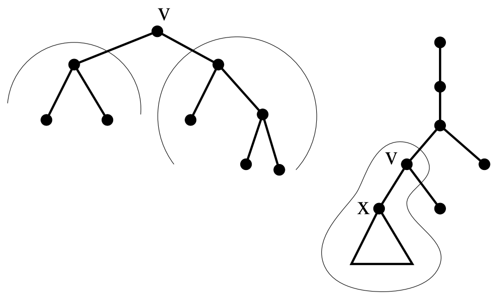
,,Querkanten" kann es nicht geben.

Lemma 2.19 *Sei G ein zusammenhängender Graph und T ein DFS-Baum von G. Dann gilt folgendes. Ein Knoten $v \in V$ ist genau dann eine Artikulation, wenn*

- *v Wurzel von T ist und mindestens zwei Söhne hat,* oder
- *es einen Sohn x von v in T gibt, so daß von* keinem *Knoten im Unterbaum T_x von T mit Wurzel x eine Rückwärtskante zu einem (echten) Vorgänger von v ausgeht.*

Beweis:

,,$\Leftarrow$": Die folgenden zwei Skizzen sollen die Gültigkeit der ,,Linksrichtung" veranschaulichen.



linke Skizze:

> v ist Wurzel und hat mindestens zwei Söhne. Aus der Vorgehensweise von DFS erkennt man, daß es keine Kante vom Teilbaum des einen Kindknoten zum Teilbaum des anderen Kindknoten geben kann. Daher führt jeder Pfad vom einen Teilbaum zum anderen unweigerlich über v.

rechte Skizze:

> Im Teilbaum T_x des Kindknotens x von v gibt es keine Rückwärtskante zu einem echten Vorgänger von v. Da es grundsätzlich keine Querkanten gibt, muß jeder Pfad von T_x zum „restlichen Graphen" über die Abwärtskante zu x und daher auch über v laufen.

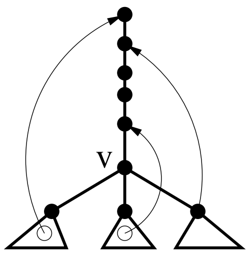
„$\Rightarrow$": Sei v eine Artikulation

1. Fall: v ist Wurzel s von T

$\Rightarrow$ s hat mindestens zwei Söhne (sonst wäre $T - s$ und damit auch $G - s$ zusammenhängend).

2. Fall: v ist nicht Wurzel von T

> Angenommen, die zweite Bedingung wäre nicht erfüllt. Dann enthält jeder Unterbaum T_x einen Knoten mit einer Rückwärtskante zu einem Vorgänger von v. Dann ist aber $G - v$ zusammenhängend.



> Das steht im Widerspruch dazu, daß v eine Artikulation ist.

Also gilt auch die „Rechtsrichtung". ■

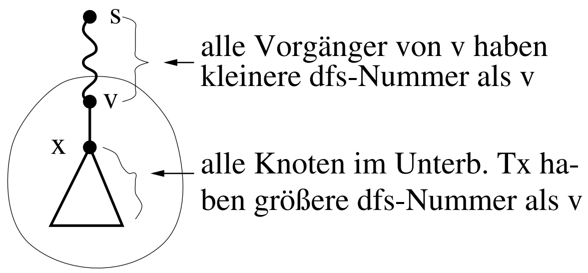
Idee des modifizierten DFS-Algorithmus:

Modifiziere den Algorithmus so, daß zusätzlich folgende Werte berechnet werden:

low[v] $:= min\{$ $dfs[w]$ $|$ w ist von v aus über ≥ 0 Baumkanten abwärts und ≤ 1 Rückwärtskanten aufwärts erreichbar$\}$

Korollar 2.20 *$v \neq s$ ist Artikulation $\Leftrightarrow$ $\exists$ Sohn x von v mit $low[x] \geq dfs[v]$*

Beweis:



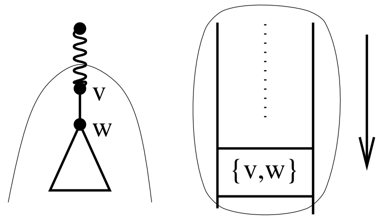
Die Behauptung folgt daher sofort aus Lemma 2.19. ■

Hier folgt nun der angekündigte Algorithmus.

<u>BLOCKZERLEGUNG(G)</u>
Initialisiere einen Stack;
$k := 0$;
$\forall v \in V$: $\texttt{dfs}[v] := 0$;
Wähle einen beliebige Knoten $s \in V$;
SCAN(s);

<u>**procedure** SCAN(v)</u>
$k++$; $\texttt{dfs}[v] := k$; $\texttt{low}[v] := k$;
for all $w \in \Gamma(v)$, $w \neq \texttt{pred}[v]$ **do**
  if $\texttt{dfs}[w] = 0$ **then**
    // Knoten w wurde vorher noch nicht betrachtet
    PUSH($\{v, w\}$);
    $\texttt{pred}[w] := v$;
    SCAN(w);
    if $\texttt{low}[w] \geq \texttt{dfs}[v]$ **then**
      // v ist Artikulationsknoten, falls $v \neq s$ oder
      // v mind. 2 Söhne hat;
      Poppe Kanten vom Stack bis einschl. Kante $\{v, w\}$;
      // ... diese bilden einen Block!
    $\texttt{low}[v] := \min\{\texttt{low}[v], \texttt{low}[w]\}$;
  else if $\texttt{dfs}[w] < \texttt{dfs}[v]$ **then**
    // $\{v, w\}$ ist Rückwärtskante
    PUSH($\{v, w\}$);
    $\texttt{low}[v] := \min\{\texttt{low}[v], \texttt{dfs}[w]\}$;

Bemerkung 2.21 *Die Block-Kanten liegen jeweils auf dem Stack und können von dort ausgelesen werden.*



Theorem 2.22 *Der Algorithmus BLOCKZERLEGUNG bestimmt die Blöcke und Artikulationen eines zusammenhängenden Graphen in $O(|E|)$.*

Beweis:

Korrektheit: ... folgt aus Lemma 2.19 und Korollar 2.20 $\surd$

Laufzeit:

> Jede Kante wird bei DFS höchstens zweimal ,,durchlaufen" und genau einmal auf den Stapel geworfen und davon geholt. $\surd$

Folglich hat der Algorithmus die von uns gewünschten Eigenschaften. ■

2.3 Bestimmung von $\lambda(G)$ (Kantenzusammenhang)

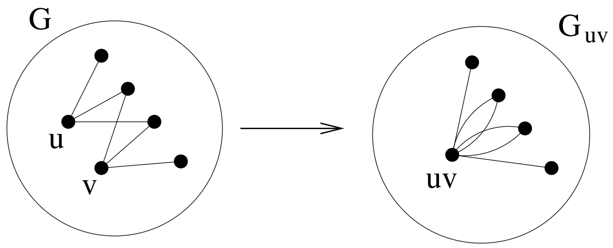
Wiederholung:

- Die Bestimmung von $\lambda(G)$ ist äquivalent zur Bestimmung eines minimalen Schnitts.
- Mit dem Netzwerkflußalgorithmus erhält man einen $O(n^2 \cdot n \cdot m)$ Algorithmus. (n^2 entspricht der Anzahl der $s-t$ Paare.)

Ziel: $O(n \cdot m)$ Algorithmus

Idee: Sei F ein minimaler Schnitt, (F ist eine Menge von *Kanten*) und seien u, v zwei beliebige Knoten. Dann gilt (mind.) eine der beiden folgenden Aussagen:

- F trennt u und v *oder*
- $\lambda(G) = \lambda(G_{uv})$, wobei G_{uv} der Graph ist, in dem u und v ,,verschmolzen" wurden.



Für den so erzeugten Graphen sind *Mehrfachkanten* erlaubt. Daraus ergibt sich unmittelbar:

Lemma 2.23 *Sei $G = (V, E)$ ein (Multi-)Graph und $u, v \in V$. Dann gilt:*

> $\lambda(G) = \min\{cut(u, v), \lambda(G_{uv})\}$, *wobei*
>
> **cut(u,v)** := *Kardinalität einer minimalen u, v trennenden Kantenmenge*

Beweis: Für den Beweis des Lemmas überlegen wir uns zunächst, daß sicherlich gilt

$$\lambda(G) \leq \min\{cut(u, v), \lambda(G_{uv})\}.$$

Dies sieht man wie folgt ein: $cut(u, v)$ ist die minimale Kardinalität einer u-v trennenden Kantenmenge, $\lambda(G)$ ist die minimale Kardinalität (irgend-)einer trennenden Kantenmenge. Daher gilt sicherlich $cut(u, v) \leq \lambda(G)$. $\lambda(G_{uv})$ ist die Kardinalität einer minimalen trennenden Kantenmenge in G_{uv}. „Blasen" wir den Knoten uv wieder auf, erhält man daraus eine trennende Kantenmenge in G, die die gleiche Anzahl Kanten enthält. Also gilt auch $\lambda(G_{uv}) \leq \lambda(G)$.
Jetzt müssen wir noch zeigen, daß auch

$$\lambda(G) \geq \min\{cut(u, v), \lambda(G_{uv})\}$$

gilt. Dazu sei F eine trennende Kantenmenge mit $|F| = \lambda(G)$. Wir unterscheiden nun zwei Fälle:

1. F trennt die beiden Knoten u und v.
   Dann gilt sicherlich $|F| \geq cut(u,v)$ und somit auch
   $$\lambda(G) = |F| \geq \min\{cut(u,v), \lambda(G_{uv})\}.$$

2. F trennt die beiden Knoten u und v nicht.
   Dann entspricht F einer trennenden Kantenmenge in G_{uv} gleicher Kardinalität. D.h., es gilt $|F| \geq \lambda(G_{uv})$ und somit wiederum auch
   $$\lambda(G) = |F| \geq \min\{cut(u,v), \lambda(G_{uv})\}.$$

■

Notation: Für disjunkte Knotenmengen X, Y sei

> ***e(X, Y)*** := Anzahl der Kanten mit einem Endpunkt in X und dem anderen in Y, wobei Mehrfachkanten entsprechend ihrer Vielfachheit gezählt werden.
>
> Gilt $X = \{x\}$, so schreiben wir auch einfach $e(x, Y)$ statt $e(\{x\}, Y)$

Definition 2.24 *Sei $G = (V, E)$ ein (Multi-)Graph. Eine Ordnung $v_1, \ldots, v_n$ der Knoten heißt* **adjazenzmaximal**, *falls*

$$e(v_i, \{v_1, \ldots, v_{i-1}\}) \geq e(v_j, \{v_1, \ldots, v_{i-1}\}) \qquad \forall\ 2 \leq i \leq j \leq n.$$

30. Mai Ausarbeitung: *Wolfgang Mayerle*

Eine adjazenzmaximale Ordnung erhält man also durch eine schrittweise Erweiterung eines Tupels von (verschiedenen) Knoten, indem man mit einem leeren Tupel startet und immer einen Knoten an das Ende anfügt, der maximal viele Nachbarn unter den links von ihm stehenden Knoten besitzt.

Mit Hilfe von Buckets läßt sich eine adjazenzmaximale Ordnung in linearer Zeit (also in $\mathcal{O}(m)$ für zusammenhängende Graphen) bestimmen: Die Bucketnummer entspricht dabei der Anzahl der "linken Nachbarn" im aktuellen Iterationsschritt, und in den Buckets liegen die noch zu ordnenden Knoten. Zu Beginn liegen demnach alle Knoten im ersten Bucket (mit der Nummer Null). Wir merken uns ebenfalls pro Knoten, in welchem Bucket er sich gerade befindet. Weiterhin benötigen wir einen Index, der als obere Schranke für den höchsten belegten Bucket fungiert (dieser zeigt zu Beginn dann ebenfalls auf den ersten). Der Algorithmus sieht wie folgt aus:

```
FindeAdjazenzmaximaleOrdnung(G)
Eingabe: Graph G = (V, E) mit |E| = m
Ausgabe: Adjazenzmaximale Ordnung

// Initialisierung..
for i := 0 to m do bucket[i] := ∅;
bucket[0] := V; for all v ∈ V do index[v] := 0;
bound := 0;
// Schleife..
while noch nicht alle Knoten ausgegeben do
    while bucket[bound] = ∅ do bound − −;
    Entferne einen beliebigen Knoten v aus bucket[bound], gebe ihn aus
    for all w ∈ Γ[v], w noch nicht ausgegeben do
        Lösche w in bucket[index[w]];
        index[w] + +; if index[w] > bound then bound := index[w];
        Füge w in bucket[index[w]] ein;
```

Um die Einfüge- und Lösch-Operationen auf den Buckets in konstanter Zeit durchführen zu können, müssen einerseits die Bucketinhalte als verkettete Liste implementiert und andererseits für jeden Knoten ein Zeiger auf seinen Eintrag gespeichert werden.

Da die Zahl der Verschiebungen der Knoten in einen höheren Bucket durch m beschränkt ist, wird *bound* ebenfalls höchstens um m erniedrigt. Der Algorithmus läuft also in linearer Zeit, was uns einen Schritt näher zu einem effizienten Kantenzusammenhangs-Algorithmus bringt:

Lemma 2.25 *Sei $G = (V, E)$ ein (Multi-)Graph und $v_1, v_2, \ldots, v_n$ eine adjazenzmaximale Ordnung. Dann gilt:*

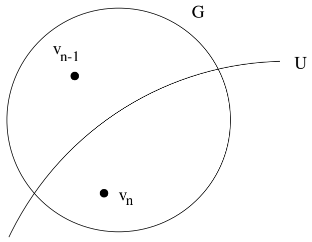
$$cut(v_n, v_{n-1}) = e(v_n, \{v_1, \ldots, v_{n-1}\})$$

Beweis: Wir beweisen die Gleichheit durch beidseitige Inklusion.

"$\leq$": klar, da auf der linken Seite der Wert eines minimalen Schnittes und auf der rechten Seite der Wert irgendeines Schnittes steht.

"$\geq$": Wir führen eine Induktion über n:

$n = 2:$ klar



$n - 1 \Rightarrow n$**:** Sei $(U, \bar{U})$ ein minimaler $v_n - v_{n-1}$-Schnitt mit $v_n \in U$. ObdA. enthalte G keine Kanten $\{v_n, v_{n-1}\}$, denn diese werden auf *beiden* Seiten der Ungleichung gezählt.

1. Fall: $v_{n-2} \in U$

$$\begin{aligned} cut(v_n, v_{n-1}) = e(U, \bar{U}) &\geq cut(v_{n-2}, v_{n-1}) \\ &\geq cut_{G-v_n}(v_{n-2}, v_{n-1}) \\ &\overset{IA}{=} e_{G-v_n}(v_{n-1}, \{v_1, \ldots, v_{n-2}\}) \\ &= e_G(v_{n-1}, \{v_1, \ldots, v_{n-2}\}) \\ &\overset{adj.max.Ord.}{\geq} e_G(v_n, \{v_1, \ldots, v_{n-2}\}) \\ &\overset{oBdA.}{=} e_G(v_n, \{v_1, \ldots, v_{n-1}\}) \end{aligned}$$

2. Fall: $v_{n-2} \notin U$: ähnlich wie der erste Fall

Es gilt also: $cut(v_n, v_{n-1}) = e(v_n, \{v_1, \ldots, v_{n-1}\})$. ■

Zusammen aus den bisherigen Lemmata können wir also folgenden Algorithmus konstruieren:

Algorithmus MinCut(G)
Eingabe: Graph $G = (V, E)$ mit $|V| = n$
Ausgabe: Kantenzusammenhangszahl $\lambda(G)$

// Initialisierung..
$\lambda := \min\{\deg(v) \mid v \in V\}$; // Kandidat für $\lambda(G)$
// Schleife..
for $k := n$ **downto** 2 **do**
  Berechne adjazenzmaximale Ordnung $v_1, \ldots, v_k$ von G;
  if $e(v_k, \{v_1, \ldots, v_{k-1}\}) < \lambda$ **then**
    $\lambda := e(v_k, \{v_1, \ldots, v_{k-1}\})$;
  $G := G_{v_k v_{k-1}}$; // verschmelze v_k und v_{k-1}

Theorem 2.26 *Obiger Algorithmus findet die Kantenzusammenhangszahl in Laufzeit $\mathcal{O}(nm)$.*

Beweis: Wie üblich ist der Beweis zweigeteilt:

Korrektheit: Die Korrektheit läßt sich durch vollständige Induktion nachvollziehen: Der Algorithmus liefert für $n = 2$ offensichtlich das richtige Ergebnis. Und nach Lemma 2.23 sowie Lemma 2.25 ist auch der Induktionsschritt klar.

Laufzeit: Wir müssen mehrere Posten berücksichtigen:

Initialisierung:	$\mathcal{O}(n+m)$
Anzahl Schleifendurchläufe:	$\mathcal{O}(n)$
Zeit pro adjazenzmax. Ordnung:	$\mathcal{O}(m)$
Zeit pro **if**:	$\mathcal{O}(m)$
Zeit pro Verschmelzung:	$\mathcal{O}(m)$

Wir erhalten also eine Gesamtlaufzeit von $\mathcal{O}(nm)$.

■

Bemerkung 2.27 *Der hier vorgestellte Algorithmus ist trotz seiner Einfachheit erst zu Beginn der 90er Jahre entstanden. Der Algorithmus geht auf Nagamochi und Ibaraki [NI92] zurück, die hier vorgestellte Analyse stammt von Stoer und Wagner [SW94].*

Kapitel 3

Planare Graphen

Ein großes Kapitel in der computerorientierten Graphentheorie beschäftigt sich mit der Aufgabe, Graphen möglichst gut zu visualisieren. Und wie so oft steckt auch hier der Teufel im Detail: Was genau ist "möglichst gut"? Die Antwort auf diese Frage ist – wie es bei interessanten Fragestellungen leider fast ausnahmslos der Fall ist: Es kommt darauf an. So will man z. B. bei einem Flußproblem vielleicht den Bottleneck (also den MinCut) übersichtlich präsentiert bekommen. Oder man will in anderen Fällen eine gewisse Struktur des Graphen leicht wiedererkennen können (wie z. B. Symmetrie, Blockzerlegung, ...).

Eine wichtige Rolle in der Graphenvisualisierung spielt die Theorie der *planaren Graphen*, das Thema dieses Kapitels. Sie beschäftigt sich mit den Graphen, welche sich ohne Kantenkreuzungen in der Ebene zeichnen lassen. Es ist wohl offensichtlich, daß solche Graphen in vielen Fällen die Übersichtlichkeit fördern. Darüberhinaus werden jedoch viele Graphenprobleme deutlich einfacher, wenn man sie auf planare Graphen beschränkt, was wohl auch daran liegt, daß planare Graphen stets "dünn" sind (s. u.).

3.1 Definitionen, Eulersche Formel, Satz von Kuratowski

Wir werden in diesem Abschnitt stark an die Intuition appellieren, da eine vollständige theoretische Behandlung sehr aufwendig und für unsere Zwecke wenig lehrreich ist.

Definition 3.1

- *Ein Graph G heißt* planar, *falls es eine Einbettung von G in die Ebene gibt, d. h. eine Darstellung, in der die Knoten durch Punkte in der Ebene und die Kanten durch Jordankurven repräsentiert werden, so daß jede der Jordankurven ihre zugehörigen Endpunkte verbindet und keine zwei Jordankurven sich außer in den Endpunkten schneiden.*
- *Ein* ebener *Graph ist ein (kreuzungsfrei) in die Ebene eingebetteter planarer Graph.*

Bemerkung 3.2 *Die Unterscheidung zwischen "planaren" und "ebenen" Graphen ist an dieser Stelle hilfreich, weil im Moment noch nicht klar ist, ob ein gegebener Graph G planar ist oder nicht, bzw. wie eine Einbettung in ersterem Falle aussieht. Bei einem* ebenen *Graphen ist das klar.*

Definition 3.3 *Sei G ein ebener Graph. Entfernt man die Kanten von G, so zerfällt die Ebene in zusammenhängende* Gebiete. *Der* Rand *des Gebiets sind alle Kanten, die im Abschluß des Gebiets liegen. Genau eines der Gebiete ist unbeschränkt, wir nennen es das* äußere *Gebiet.*

Abb. 3.1 zeigt anhand eines Beispiels die Unterscheidung zwischen den verschiedenen Graphen. Zwei in gewisser Weise maximal mögliche planare Graphen sind in Abb. 3.2 dargestellt, nämlich der vollständige Graph K_4 sowie der vollständige bipartite Graph $K_{2,3}$. Vielleicht erinnert sich der Leser in diesem Zusammenhang an eine Knobelaufgabe, die ihm in einer Zeit, als die Welt noch einfach schien, gegeben wurde (Abb. 3.3): Kann man alle drei Häuser an alle Versorgungspunkte anschließen, ohne daß sich die Leitungen schneiden? (Wir verraten die Lösung ein wenig später.)

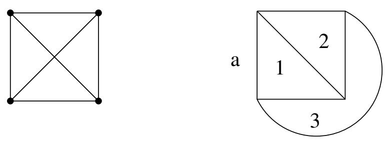


Abbildung 3.1: Ein planarer Graph, einer seiner ebenen Graphen und seine Gebiete

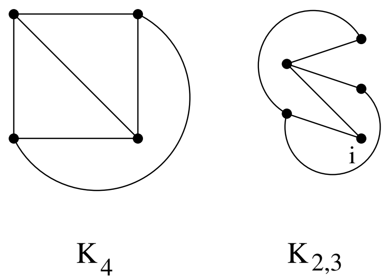


Abbildung 3.2: Zwei in gewissem Sinne (s. Text) maximale, planare Graphen

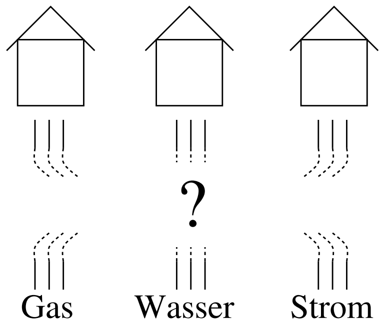


Abbildung 3.3: Ist der $K_{3,3}$ planar?

Proposition 3.4 *Man kann jeden planaren Graphen so in die Ebene einbetten, daß ein beliebiges vorgegebenes Gebiet das äußere Gebiet ist.*

Beweis: Intuitiv kann man jeden Graphen, der sich in der Ebene einbetten läßt, ebenfalls auf einer Kugeloberfläche einbetten. Reißt man nun die Kugel innerhalb des gewünschten Gebietes auf und deformiert die Kugeloberfläche anschließend zu einer Ebene, erhält man die gewünschte Einbettung. ∎

Bemerkung 3.5 *Zwar ist der Beweis von Proposition 3.4 einleuchtend, jedoch liefert er uns noch keinen Algorithmus zum "Umstülpen" eines Graphen. Wir wollen uns jedoch in diesem Kapitel damit zufrieden geben, daß es im Prinzip möglich ist.*

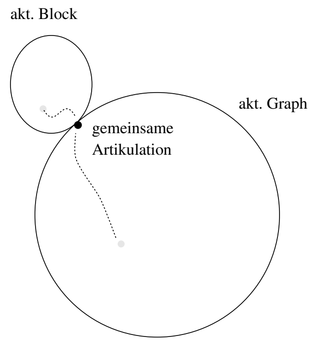
Korollar 3.6 *G ist planar $\Leftrightarrow$ alle Blöcke von G sind planar*

Beweis: Wir müssen zwei Richtungen zeigen:

"$\Rightarrow$": klar.

"$\Leftarrow$": Wir wissen, daß der Blockartikulationsgraph ein Baum ist, also insbesondere planar. Wir sind damit jedoch noch nicht fertig, denn eine Artikulation in einem Block kann ja durchaus *innerhalb* des Blocks liegen (vgl. den mit i markierten Knoten in Figur 3.2).

Wir gehen also wie folgt vor: Wir konstruieren einen ebenen Graphen von G schrittweise durch Hinzunahme eines beliebigen Blocks. Dabei wird vor der Verschmelzung sowohl bei dem gerade im Aufbau befindlichen Graphen als auch beim aktuellen Block die gemeinsame Artikulation nach Proposition 3.4 an den jeweiligen äußeren Rand gebracht und beide Teile anschließend miteinander "verklebt".



∎

Theorem 3.7 (Eulersche Formel)

Sei G ein zusammenhängender ebener Graph. Dann gilt:

$$|V| - |E| + |R| = 2,$$

wobei R die Anzahl Gebiete (Regions) bezeichnet.

Beweis: Wir beweisen den Satz durch eine Induktion über die Anzahl der Kanten.

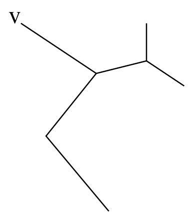
$m = 0$: Da der Graph zusammenhängend sein muß, kann er nur einen Knoten besitzen. Der Graph hat ebenfalls nur ein Gebiet, womit die Formel korrekt ist.

$m - 1 \Rightarrow m$: Wir betrachten zwei Fälle:

1. Fall: G ist ein Baum. Wähle ein beliebiges Blatt v in G. $G - v$ ist zusammenhängend, also gilt nach Induktionsannahme:

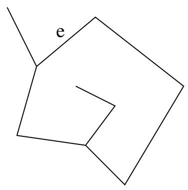
$$\underbrace{|V(G-v)|}_{=|V|-1} - \underbrace{|E(G-v)|}_{=|E|-1} + \underbrace{|R(G-v)|}_{=|R|} = 2,$$

und damit gilt auch der Induktionsschritt.



2. Fall: G ist kein Baum. Sei C ein beliebiger Kreis in G und e eine Kante aus C. $G - e$ ist zusammenhängend, und es gilt wiederum nach Induktionsannahme:

$$\underbrace{|V(G-e)|}_{=|V|} - \underbrace{|E(G-e)|}_{=|E|-1} + \underbrace{|R(G-e)|}_{=|R|-1} = 2,$$



da zwei Gebiete zu einem verschmelzen (eine Kante in einem Kreis trennt genau zwei Gebiete voneinander).

Die Eulersche Formel gilt also für alle zusammenhängenden ebenen Graphen. ∎

Wir werden in Kürze aus diesem Satz folgern, daß planare Graphen "dünn" sind, d. h. nur $\mathcal{O}(|V|)$ viele Kanten besitzen können. Die genaue Schranke läßt sich dabei für spezielle planare Graphen noch enger bestimmen.

Definition 3.8 *Ein Graph G heißt* dreiecksfrei, *falls er keine Kreise der Länge 3 enthält.*

Korollar 3.9

1. *G planar, $|V| \geq 3 \Rightarrow |E| \leq 3|V| - 6$*
2. *G planar und dreiecksfrei, $|V| \geq 3 \Rightarrow |E| \leq 2|V| - 4$*

Beweis: Die Formeln ergeben sich in beiden Fällen aus einer Identität und der Eulerschen Formel:

1. Betrachte eine Einbettung von G. Offenbar gilt:

$$\sum_{A \in R} \underbrace{\sum_{\substack{e \in E:\\ e \in A}} 1}_{\geq 3} = \sum_{e \in E} \underbrace{\sum_{\substack{A \in R:\\ e \in A}} 1}_{\leq 2},$$

da jedes Gebiet mindestens aus drei Kanten besteht und jede Kante zu höchstens zwei Gebieten gehört.

Daraus folgt: $3|R| \leq \cdots \leq 2|E|$, was eingesetzt in die Eulersche Formel

$$\begin{aligned} 2 = |V| - |E| + |R| &\leq |V| - |E| + \frac{2}{3}|E| \\ &= |V| - \frac{1}{3}|E| \text{ bzw.} \\ |E| &\leq 3|V| - 6 \text{ ergibt.} \end{aligned}$$

2. Dieser Fall läßt sich völlig analog beweisen, lediglich die linke Abschätzung verändert sich von ≥ 3 zu ≥ 4.

∎

Korollar 3.10 *Die Graphen K_5 und $K_{3,3}$ sind nicht planar.*

Beweis: Nach Korollar 3.9 gilt, daß planare Graphen mit 5 Knoten höchstens 9 Kanten besitzen dürfen. Der K_5 hat jedoch $\binom{5}{2} = 10$.

Ein Graph ist bipartit genau dann, wenn er keine Kreise ungerader Länge enthält. Danach ist jeder bipartite Graph, also auch unser $K_{3,3}$, dreiecksfrei. Korollar 3.9 liefert uns für dreiecksfreie Graphen mit 6 Knoten eine obere Schranke von 8 für die Anzahl der Kanten. Der $K_{3,3}$ hat jedoch $3 \cdot 3 = 9$, also wieder eine zu viel. ∎

Definition 3.11 *Ein Graph H heißt* Unterteilung *vom Graph G, falls H durch sukzessives Einfügen von Knoten auf Kanten von G erzeugt werden kann.*

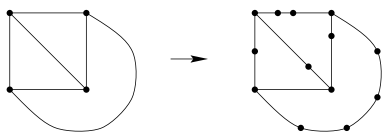


Abbildung 3.4: Ein planarer Graph und eine mögliche Unterteilung

Bemerkung 3.12

- *Wie auch aus dem Beispiel in Abb. 3.4 ersichtlich ist, entsprechen den Kanten im Ursprungsgraphen eine Menge von kreuzungsfreien Pfaden in der Unterteilung.*
- *Falls ein Graph nicht planar ist, so ist auch jede seiner Unterteilungen nicht planar.*

Theorem 3.13 (Satz von Kuratowski, 1930)

Ein Graph G ist planar genau dann, wenn er weder eine Unterteilung des $K_{3,3}$ noch des K_5 enthält.

Bemerkung 3.14 *Wir verzichten an dieser Stelle aus Platzgründen auf einen Beweis, werden jedoch im Laufe des folgenden Abschnitts eine mögliche Beweisidee skizzieren.*

Bemerkung 3.15 *Nach Satz 3.13 sind die Graphen $K_{3,3}$ und K_5 in gewissem Sinne die "einzigen" nicht-planaren Graphen.*

3.2 Planaritätstest

5./6. Juni Ausarbeitung: *Maximilian Fischer*

In vielen Fällen ist nicht unmittelbar zu entscheiden, ob es sich bei einem gegebenen Graphen um einen planaren Graphen handelt. Ziel dieses Abschnittes ist es daher, einen effizienten Algorithmus zu beschreiben, der einen beliebigen Graphen $G = (V, E)$ als Eingabe erhält und als Ausgabe eine mögliche Einbettung von G liefert, falls G planar ist, bzw. erkennt, daß G nicht planar ist. Dabei soll eine Laufzeit von $\mathcal{O}(|V|)$ erreicht werden. Es sei darauf hingewiesen, daß hier keine Versuche unternommen werden, eine möglichst "schöne" Einbettung zu finden. Die Untersuchung solcher Heuristiken ist ein eigenständiges Themengebiet, das den Rahmen dieser Vorlesung leider sprengen würde.

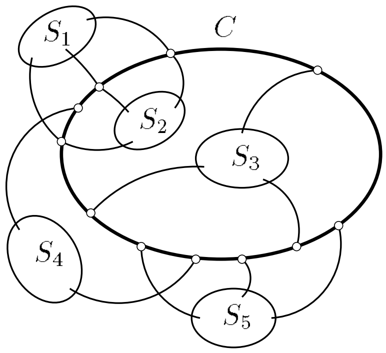
Ohne Beschränkung der Allgemeinheit wird im folgenden angenommen, daß es sich bei G um einen 2-fach zusammenhängenden Graphen handelt. Andernfalls kann der Graph zunächst an seinen Artikulationsknoten zerlegt und die Komponenten einzeln auf Planarität getestet werden. Aus den Einbettungen der Komponenten kann man dann auf einfache Weise eine Einbettung für den Gesamtgraphen erhalten. Zudem können wir nach Korollar **??** annehmen, daß die Bedingung $m \leq 3n - 6$ erfüllt ist. Ansonsten ist der Graph ja auf keinen Fall planar.

Definition 3.16 *Sei $G = (V, E)$ ein Graph, der einen Kreis C enthält (ein solcher existiert, da wir 2-fach Zusammenhang vorausgesetzt hatten). Ein* ***Segment*** *ist eine Komponente von $\left(V \setminus C, E \setminus \binom{C}{2}\right)$ zusammen mit allen von dort ausgehenden Kanten (aber ohne deren Endknoten auf C).*

Die ***Verankerungen*** *eines Segmentes sind die Knoten von C, an denen eine Kante des Segmentes endet.*

Zwei Segmente S_i und S_j heißen ***überlappend****, falls sie mindestens drei gemeinsame Verankerungen haben oder es paarweise verschiedene Verankerungen a, b von S_i und c, d von S_j gibt, so daß diese auf dem Kreis C in der Reihenfolge a, c, b, d vorkommen.*

Nebenstehende Skizze veranschaulicht diese Begriffe. Bei S_1 bis S_5 (incl. der ausgehenden Kanten) handelt es sich um Segmente bezüglich C. Die markierten Punkte auf C stellen die Verankerungen der Segmente dar. Überlappend sind beispielsweise die Segmente S_1 und S_2 (drei gemeinsame Verankerungen), S_3 und S_5 sowie S_4 und S_5. Intuitiv gesprochen sind zwei Segmente also genau dann überlappend, wenn man sie nicht beide auf einer Seite des Kreises (also innen oder außen) einbetten kann, ohne daß sich Kanten kreuzen.



Lemma 3.17 *Sei G ein Graph mit einem Kreis C und den Segmenten $S_1, \ldots, S_k$. Gilt $C + S_i$ planar $\forall\, i = 1, \ldots, k$ und S_i und S_j überlappen sich nicht $\forall\, 1 \leq i < j \leq k$, so ist G planar und es gibt eine Einbettung, in der alle Segmente außerhalb von C liegen (oder alle innerhalb, vgl. Proposition 3.4).*

Beweis: Wir führen den Beweis per Induktion über die Anzahl der Segmente k.

$k = 0$: klar.

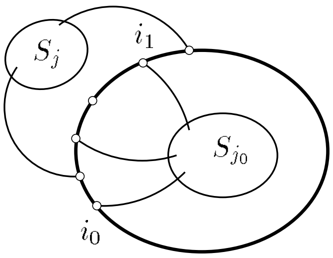
$k - 1 \Rightarrow k$: Seien $1, \ldots, \ell$ die Knoten des Kreises C (in dieser Reihenfolge).

Wähle zwei Knoten des Kreises $i_0, i_1 \in \{1, \ldots, \ell\}$ mit kleinstmöglichem Abstand, so daß ein Segment S_{j_0} existiert, für das alle Verankerungen in $\{i_0, \ldots, i_1\}$ liegen. Falls es mehrere solche S_{j_0} gibt, wähle eines mit maximal vielen Verankerungen.

Behauptung: Für alle Segmente S_j mit $j \neq j_0$ gilt:
S_j enthält keine Verankerung aus $\{i_0 + 1, \ldots, i_1 - 1\}$.

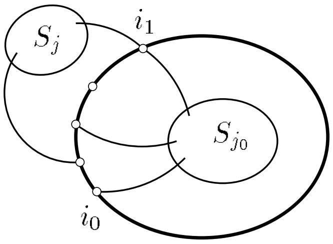
Beweis: Wir zeigen die Behauptung mit einem Widerspruchsbeweis. Angenommen, S_j enthält doch so eine Verankerung. Dann sind drei Fälle zu unterscheiden:

1. Fall: S_j enthält auch eine Verankerung außerhalb von $\{i_0, \ldots, i_1\}$



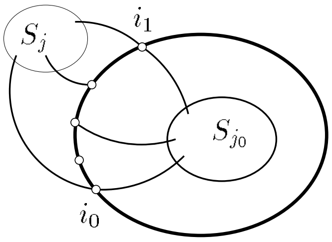
$\Rightarrow$ Widerspruch, da S_{j_0} und S_j überlappen.

2. Fall: S_j enthält keine Verankerung außerhalb von $\{i_0, \ldots, i_1\}$ und i_0 oder i_1 ist keine Verankerung von S_j



$\Rightarrow$ Widerspruch zur Wahl von i_0, i_1.

3. Fall: i_0 und i_1 sind Verankerungen von S_j. Da S_j und damit auch S_{j_0} (wir haben das Segment mit maximal vielen Verankerungen gewählt) eine weitere Verankerung in $\{i_0, \ldots, i_1\}$ hat, folgt



$\Rightarrow$ Widerspruch, da S_{j_0} und S_j überlappen.

(Fortsetzung des Induktionsbeweises:)

$C + (S_i + \cdots + S_{j_0-1} + S_{j_0+1} + \cdots + S_k)$ sind planar nach der Induktionsannahme. Außerdem ist $C + S_{j_0}$ planar nach Voraussetzung.

$\Rightarrow$ fügt man beide zusammen, so erhält man einen planaren Graphen!

Damit ist der Induktionsschritt abgeschlossen und Lemma 3.17 gezeigt. ■

Korollar 3.18 *Sei G ein Graph mit einem Kreis C und den Segmenten $S_1, \ldots, S_k$.*

Dann gilt: G planar $\Leftrightarrow$ (1) $C + S_i$ planar $\forall\, i = 1, \ldots, k$ und

(2) $\exists$ Partition $\{1, \ldots, k\} = I_1 \cup I_2$, so daß sich keine zwei Segmente aus der gleichen Menge überlappen.

Beweis: Zu zeigen sind wieder beide Schlußrichtungen:

“$\Rightarrow$”: Klar: wähle für I_1 die Menge der innen liegenden Segmente und für I_2 die außen liegenden Segmente.

“$\Leftarrow$”: Wende Lemma 3.17 zweimal an: erst auf die Segmente in I_1 (bette diese außen ein), dann auf die Segmente in I_2 (bette diese innen ein).

Damit folgt die Aussage. ■

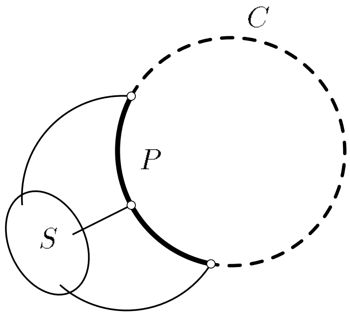
Bemerkung 3.19 *Aufbauend auf Korollar 3.18 kann man (mit einigem Aufwand) den Satz von Kuratowski ($\rightarrow$ Theorem 3.13) beweisen.*

Lemma 3.20 *Sei C ein Kreis, S ein Segment und P ein Pfad von C, der alle Verankerungen von S enthält.*

Dann gilt: $C + S$ planar $\Leftrightarrow$ (1) $P + S$ planar und

(2) es gibt eine Einbettung von $P + S$, in der P zum Rand des äußeren Gebietes gehört.

Beweis: Die Richtigkeit dieser Aussage ist nach folgender Skizze unmittelbar einsichtig:



■

Lemma 3.20 liefert also eine Möglichkeit, den auf Planarität zu testenden Graphen zu verkleinern, wovon im folgenden Gebrauch gemacht werden wird.

Idee für einen Planaritätstest

Die bisherigen Überlegungen führen zu einer ersten Idee, wie sich ein Planaritätstest für einen Graphen G realisieren läßt:

- Wähle Kreis C beliebig, bestimme die Segmente S_i und entsprechende Pfade P_i.
- Verifiziere rekursiv, daß $S_i + P_i$ (hat weniger Kanten als G!) planar ist und es eine Einbettung gibt, in der P_i außen liegt.
- Bestimme den Überlappungsgraphen UG der Segmente S_i:

Knoten von UG	$\hat{=}$	Segmente S_i
Kanten von UG:		S_i und S_j werden genau dann durch eine Kante verbunden, wenn S_i und S_j überlappen.

- Verifiziere, ob UG bipartit ist ($\rightarrow$ÜA 38, EAD I). Ist dies der Fall, so ergeben sich aus den beiden bipartiten Knotenmengen von UG die Segmente, die innen bzw. außen eingebettet werden müssen, und G ist damit planar (vgl. Abbildung 3.5).

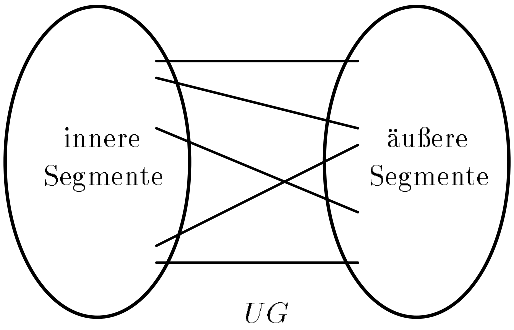


Abbildung 3.5: Bipartiter Überlappungsgraph

Unglücklicherweise ist die erzielbare Laufzeit bei diesem Vorgehen ziemlich miserabel. Schon alleine der Überlappungstest zur Erzeugung von UG hat im allgemeinen eine Laufzeit von $\mathcal{O}(n^2)$.

Folgende Verbesserungen sind daher angebracht:

(1) geschicktere Durchführung der Rekursion und

(2) Überlappungsgraph nicht explizit konstruieren!

Kernidee zur Umsetzung dieser Forderungen ist es, dafür zu sorgen, daß die Segmente in einer bestimmten (geschickten!) Reihenfolge erzeugt werden. Die im Anschluß aufgeführten Punkte sollen diesen Ansatz zunächst zusammenhängend skizzieren, wobei einige ergänzende Laufzeitbetrachtungen eingeflochten sind. Abschließend werden die einzelnen Phasen des geschilderten Planaritätstests dann konkret an einem Beispielgraphen demonstriert werden.

Verbesserte Vorgehensweise

- Bestimme einen DFS-Baum für G und berechne dabei auch die Felder

$$\left.\begin{array}{l}\text{dfs}[v]\\ \text{low}[v]\end{array}\right\}\ \text{wie bisher (vgl. Kapitel ??)}$$

$\text{low2}[v] = \min\{\text{dfs}[v],\ \text{dfs}[w] \mid \text{dfs}[w] \neq \text{low}[v],\ w$ durch ≥ 0 Baumkanten abwärts und genau eine Rückwärtskante von v aus erreichbar$\}$

[D.h., $\text{low2}[v]$ gibt den *zweit*kleinsten dfs-Wert an, der durch ≥ 0 Baumkanten abwärts und genau eine Rückwärtskante von v aus erreichbar ist bzw. den dfs-Wert von v, falls ein entsprechender Knoten nicht existiert.]

$\Rightarrow$ Laufzeit $\mathcal{O}(n)$ [wegen $m \leq 3n - 6$ ist $\mathcal{O}(m) = \mathcal{O}(n)$ bei planaren Graphen]

- Richte Kanten:
  - Baumkanten abwärts
  - Rückwärtskanten aufwärts
- Gewichte die Kanten folgendermaßen:

$$c(v,w) = \begin{cases} 2 \cdot \text{dfs}[w] & \text{falls } (v,w) \text{ Rückwärtskante} \\ 2 \cdot \text{low}[w] & \text{falls } (v,w) \text{ Baumkante und } \text{low2}[w] \geq \text{dfs}[v] \\ 2 \cdot \text{low}[w] + 1 & \text{falls } (v,w) \text{ Baumkante und } \text{low2}[w] < \text{dfs}[v] \end{cases}$$

- Erzeuge für alle $v \in V$ eine Adjazenzliste der ausgehenden Kanten, *aufsteigend sortiert* nach deren Gewicht $c(.)$.

  $\Rightarrow$ Laufzeit $\mathcal{O}(n)$

  Beweis: Werfe zunächst alle Kanten gemäß ihres Gewichtes in einen Bucket der Größe $2n$ ($\rightarrow \mathcal{O}(n)$), hole sie dann gemäß absteigendem Gewicht heraus und verteile sie auf die (anfangs leeren) Adjazenzlisten ($\rightarrow \mathcal{O}(n)$). ∎

- Markiere jede Kante als "neu" (d.h. unbetrachtet).
- Bestimme einen Kreis C:

```
v := ⟨Wurzel des DFS-Baumes⟩;
w := erster Nachbar von v in der Adjazenzliste von v;
markiere (v, w) als "benutzt"; C := {(v, w)};
while (w ≠ ⟨Wurzel des DFS-Baumes⟩)
{
    v := w;
    w := Endknoten der ersten "neuen" Kante in Adjazenzliste von v;
    markiere (v, w) als "benutzt"; C := C + (v, w);
}
Gib Kreis C aus;
```

$\Rightarrow$ C besteht aus Baumkanten und genau einer Rückwärtskante (Beweis ÜA).

- Als nächstes werden die Segmente mit folgendem Algorithmus in "kleinen Teilen" ausgegeben, die im weiteren als *Pfade* bezeichnet werden sollen:

```
v := ⟨Knoten mit größter dfs-Nummer in C⟩;
// dies ist Knoten v am Ende des vorigen Algorithmus

while (v ≠ ⟨Wurzel des DFS-Baumes⟩)
{
    while (es gibt "neue" Kanten in Adjazenzliste von v)
    {
        w := Endknoten der ersten "neuen" Kante in Adjazenzliste von v;
        markiere (v, w) als "benutzt"; P := {(v, w)};
        while (dfs[v] < dfs[w])
        {
            // (v, w) ist eine Baumkante
            v := w;
            w := Endknoten der ersten "neuen" Kante in Adjazenzliste von v;
            markiere (v, w) als "benutzt"; P := P + (v, w);
        }
        Gib Pfad P aus;
    }
    v := Vorgänger von v;
}
```

Bemerkung 3.21 *Sei S ein Segment bezüglich C und v eine Verankerung von S mit größter DFS-Nummer (vgl. Abbildung 3.6). Dann treffen folgende Beobachtungen zu:*

- S wird über eine Baumkante (v, w) von v aus betreten und vollständig ausgegeben, bevor der Algorithmus S wieder verläßt.
- S besteht aus dem Subbaum T_w des DFS-Baumes und allen von dort ausgehenden Rückwärtskanten.
- Sei u der Endknoten des ersten ausgegebenen Pfades. Dann enthält S nur Verankerungen aus P_{uv}, wobei P_{uv} der Pfad des DFS-Baumes (bzw. des Kreises C) ist, der u und v verbindet.

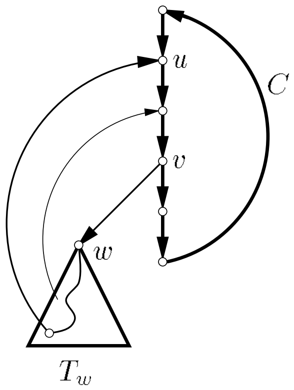
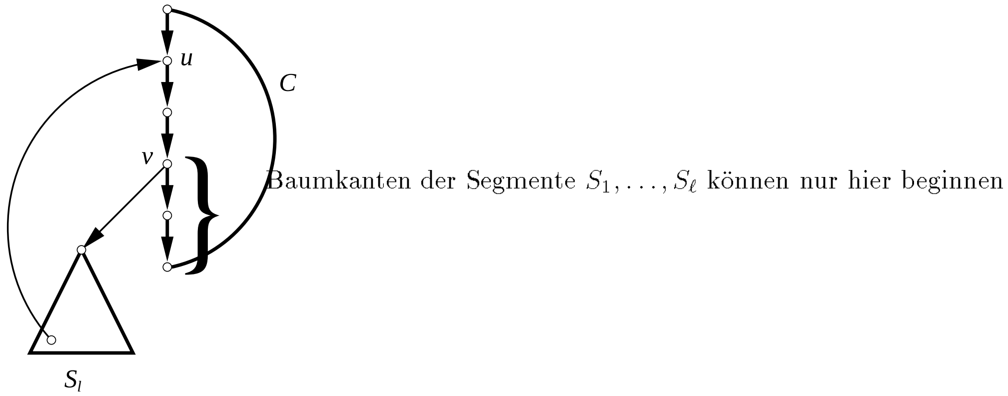


Abbildung 3.6: Skizze zu Bemerkung 3.21

Lemma 3.22 *Seien $S_1, \ldots, S_\ell$ die ersten ℓ Segmente, die vom Algorithmus ausgegeben werden und $P_\ell = (v, \ldots, u)$ der erste Pfad von S_ℓ, der ausgegeben wird. Weiter sei eine Einbettung von $C + S_1 + \cdots + S_{\ell-1}$ gegeben. Dann gilt:*

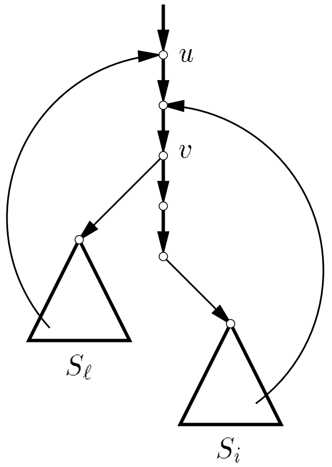
S_ℓ läßt sich genau dann nicht außen (innen) hinzufügen, wenn es ein außen (innen) eingebettetes Segment S_i mit $i \in \{1, \ldots, \ell - 1\}$ gibt, das eine Verankerung zwischen u und v besitzt.

Beweis: Aufgrund der Arbeitsweise des Algorithmus zur Ausgabe der Pfade und den Beobachtungen aus Bemerkung 3.21, können die Baumkanten der bereits vollständig ausgegebenen Segmente $S_1, \ldots, S_{\ell-1}$ nur von den Knoten ausgehen, die in nachstehender Skizze gekennzeichnet sind:



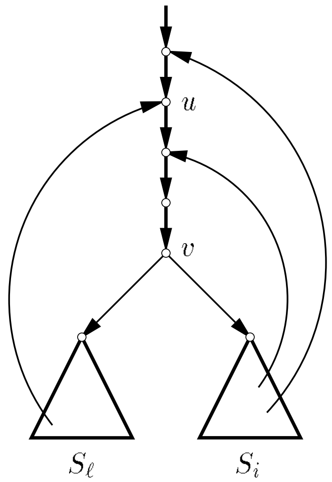
Angenommen, es gibt ein Segment S_i mit $i \in \{1, \ldots, \ell - 1\}$, das eine Verankerung innerhalb des Pfades P_{uv} des DFS-Baumes (bzw. des Kreises C) besitzt. Dann sind folgende Fälle zu unterscheiden:

1. Fall: Die Baumkante, über die S_i betreten wird, beginnt unterhalb von Knoten v



$\Rightarrow$ S_ℓ und S_i überlappen.

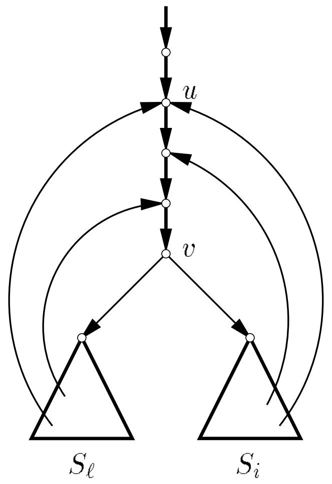
2. Fall: Die Baumkanten, über die S_ℓ und S_i betreten werden, beginnen im gleichen Knoten v und der erste Pfad von S_i, der ausgegeben wurde, endet oberhalb von Knoten u



$\Rightarrow$ S_ℓ und S_i überlappen.

3. Fall: Die Baumkanten, über die S_ℓ und S_i betreten werden, beginnen im gleichen Knoten v und der erste Pfad von S_i, der ausgegeben wurde, endet ebenfalls in Knoten u (beachte, daß der Pfad nicht unterhalb von u enden kann, ansonsten wäre S_ℓ vor S_i ausgegeben worden).

Damit muß auch S_ℓ eine zusätzliche Rückwärtskante haben, die zwischen u und v endet, sonst wäre S_ℓ vor S_i ausgegeben worden (aus diesem Grund brauchten wir das Feld low2[.] !), also folgt



$\Rightarrow$ S_ℓ und S_i überlappen.

Damit ist ein eindeutiges Kriterium gezeigt, das angibt, wann sich eine gegebene Einbettung um ein Segment erweitern läßt, ohne die Planarität zu verletzen. ■

Verbesserte Vorgehensweise (Fortsetzung)

- Der Algorithmus "erkennt", wenn der erste Pfad eines Segmentes ausgegeben wird. Daraufhin schickt er den derzeitigen Stand (die derzeitige Einbettung) auf einen Stack und macht rekursiv mit dem gerade ausgegebenen Pfad zusammen mit einem Teil des alten Kreises als neuem Kreis weiter.

  Es werden folgende Datenstrukturen verwendet:

  L_i : Liste der Pfade, die innen eingebettet werden

  L_o : Liste der Pfade, die außen eingebettet werden

  Diese Listen sind aufgeteilt in Blöcke, für die die Einbettung eines Pfades bereits die Einbettung aller übrigen vorschreibt. Durch diese Form der Organisation kann für jeweils eine ganze Gruppe von Pfaden das Ändern des Ortes der Einbettung von innen nach außen und umgekehrt ("flippen") durch Pointer-Manipulationen in konstanter Zeit bewerkstelligt werden. Auf die genaue Ausführung der zugehörigen Details wollen wir an dieser verzichten. Der interessierte Leser kann sie beispielsweise in [Meh84] nachlesen. Zusammenen mit den bereits angeführten Laufzeitbetrachtungen ergibt sich dann:

Theorem 3.23 *Der Planaritätstest läßt sich mit Laufzeit $\mathcal{O}(n)$ implementieren.*

Damit ist das eingangs formulierte Ziel, einen Planaritätstest mit linearem Laufzeitverhalten anzugeben, erreicht. Der verbleibende Teil dieses Abschnitts ist einem ausführlichen Beispiel gewidmet. Dazu soll der 2-fach zusammenhängende Graph aus Abbildung 3.7a auf Planarität getestet und gegebenenfalls eine mögliche Einbettung gefunden werden.

Zunächst werden mit Hilfe einer modifizierten Tiefensuche die Felder dfs[.], low[.] und low2[.] für den gegebenen Graphen bestimmt. Nehmen wir für unser Beispiel an, daß

$$\overset{a}{\bullet} \longrightarrow \overset{b}{\bullet} \longrightarrow \overset{c}{\bullet} \longrightarrow \overset{d}{\bullet} \longrightarrow \overset{e}{\bullet} \longrightarrow \overset{f}{\bullet} \longrightarrow \overset{g}{\bullet}$$

als DFS-Baum ermittelt wurde (vgl. Abbildung 3.7b), dann ergeben sich für oben genannte Felder die Werte aus Tabelle 3.1, wie man leicht überprüfen kann.

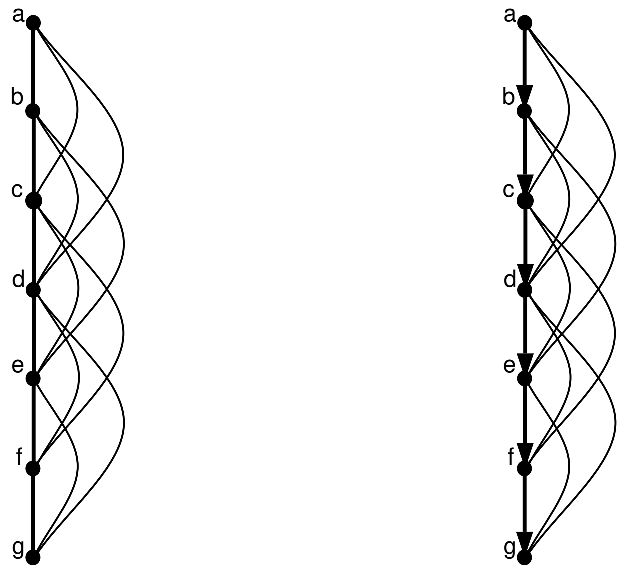


Abbildung 3.7: a) Beispielgraph b) berechneter DFS-Baum

Knoten	dfs[.]	low[.]	low2[.]
a	1	1	1
b	2	1	2
c	3	1	2
d	4	1	2
e	5	2	3
f	6	3	4
g	7	4	5

Tabelle 3.1: Resultat der modifizierten Tiefensuche

Entsprechend dem Resultat der Tiefensuche werden nun die Baumkanten abwärts, die Rückwärtskanten aufwärts gerichtet, sowie die Kantengewichte bestimmt. Man erhält dann den in Abbildung 3.8 wiedergegebenen Graphen.

Nächster Schritt ist die Bestimmung eines Kreises C. Aufgrund der aufsteigend sortierten Adjazenzlisten ergibt sich dieser zu

$$\overset{\text{a}}{\bullet} \longrightarrow \overset{\text{b}}{\bullet} \longrightarrow \overset{\text{c}}{\bullet} \longrightarrow \overset{\text{a}}{\bullet}.$$

Die drei beteiligten Kanten gelten jetzt als benutzt. Offenbar existiert bezüglich dieses Kreises lediglich ein einziges Segment S, das über die Baumkante (c, d) betreten wird, wie Abbildung 3.8 *(rechts)* zeigt. Da alle Verankerungen des Segmentes auf dem Pfad $P = (a, b, c)$ liegen, ist nach Lemma 3.20 der gesamte Graph also bereits dann planar, wenn $P + S$ planar ist und es eine Einbettung gibt, in der P zum Rand des äußeren Gebiets gehört.

In der vorliegenden algorithmischen Umsetzung erfolgt die Ermittlung von S durch die Ausgabe von Pfaden. Der Algorithmus startet hierzu in Knoten c und wählt zunächst die Kante (c, d), weil dies die einzige unbenutzte adjazente Kante zu c ist. Da es sich um eine Baumkante handelt, werden nun die von d ausgehenden Kanten betrachtet und die Rückwärtskante (d, a) benutzt, denn diese weist das geringste Gewicht auf. Damit ist mit

$$\overset{\text{c}}{\bullet} \longrightarrow \overset{\text{d}}{\bullet} \longrightarrow \overset{\text{a}}{\bullet}$$

der erste Pfad von Segment S bestimmt.

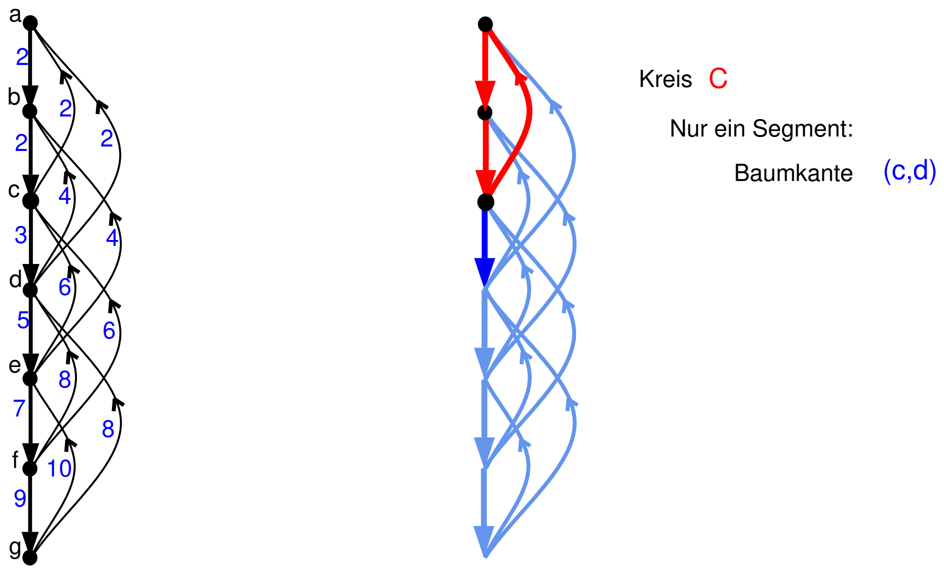


Abbildung 3.8: gewichteter und gerichteter Graph *(links)* und initiales Segment *(rechts)*

Aus diesem Pfad und einem Teil des Kreises C erhält man nun für die Rekursion

$$\overset{a}{\bullet} \longrightarrow \overset{b}{\bullet} \longrightarrow \overset{c}{\bullet} \longrightarrow \overset{d}{\bullet} \longrightarrow \overset{a}{\bullet}$$

als neuen Kreis C'. Also lautet die Aufgabe im Rekursionsschritt wie folgt:

1. Rekursion: Finde planare Einbettung von Abbildung 3.9 *(links)*, so daß $\overset{a}{\bullet} \longrightarrow \overset{b}{\bullet} \longrightarrow \overset{c}{\bullet}$ zum Rand des äußeren Gebietes gehört.

Der gegenüber der Ausgangssituation verkleinerte Graph besitzt zwei Segmente bezüglich des Kreises C'. Zum einen ist dies die Rückwärtskante (d, b), zum anderen der Teilbaum, der über die Baumkante (d, e) betreten wird, zusammen mit allen davon ausgehenden Rückwartskanten, siehe Abbildung 3.9 *(rechts)*.

In analoger Weise ergeben sich die Teilprobleme für die restlichen Rekursionsschritte:

2. Rekursion: Finde planare Einbettung von Abbildung 3.10 *(links)*, so daß $\overset{b}{\bullet} \longrightarrow \overset{c}{\bullet} \longrightarrow \overset{d}{\bullet}$ zum Rand des äußeren Gebietes gehört.

3. Rekursion: Finde planare Einbettung von Abbildung 3.11 *(links)*, so daß $\overset{c}{\bullet} \longrightarrow \overset{d}{\bullet} \longrightarrow \overset{e}{\bullet}$ zum Rand des äußeren Gebietes gehört.

4. Rekursion: Finde planare Einbettung von Abbildung 3.12, so daß $\overset{d}{\bullet} \longrightarrow \overset{e}{\bullet} \longrightarrow \overset{f}{\bullet}$ zum Rand des äußeren Gebietes gehört.

Wie in Abbildung 3.12 zu sehen ist, ist hier der Punkt erreicht, an dem kein weiterer rekursiver Aufruf mehr erforderlich ist, da der soweit reduzierte Graph bereits der gewünschten Einbettung entspricht. Daher wird nun der Rekursionsstack abgearbeitet. Sozusagen "rückwärts" werden jetzt die jeweils vor dem Rekursionsschritt beiseite gelassenen Pfade zu der Einbettung hinzugefügt. In unserem Fall verläuft dies erfolgreich, wie in Abbildung 3.13 dargestellt. Damit ist gezeigt, daß für den Beispielgraphen eine planare Einbettung existiert.

Es bietet sich hier an, nochmals die Bemerkung vom Beginn dieses Abschnitts aufzugreifen. Dort wurde gesagt, daß wir uns damit begnügen wollen, eine mögliche Einbettung zu finden, nicht jedoch eine besonders "schöne". Entsprechend kann man auch für den vorgestellten Beispielgraphen Einbettungen finden, die weitaus schlichter ausfallen, wie Abbildung 3.14 zeigt.

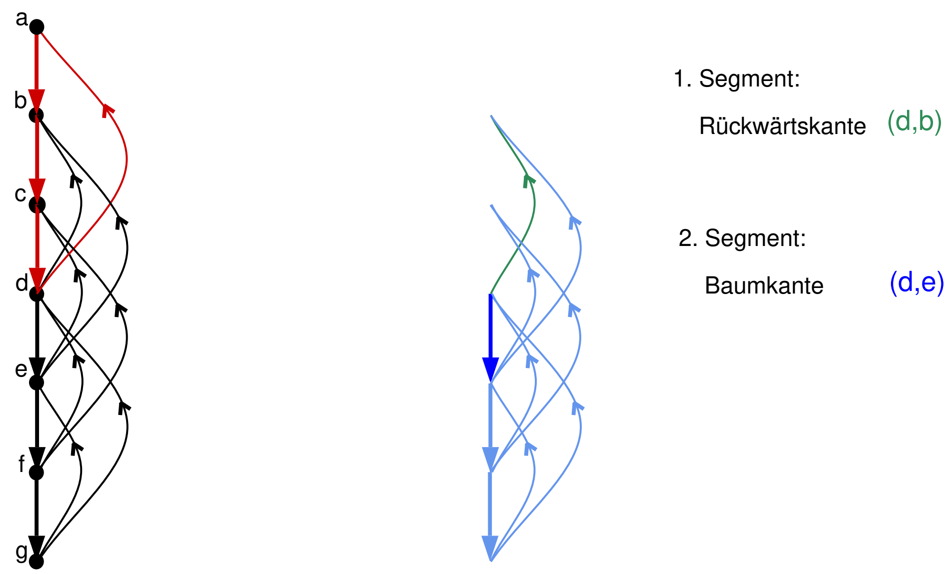


Abbildung 3.9: Problemgraph für die 1. Rekursion *(links)* und aktuelle Segmente *(rechts)*

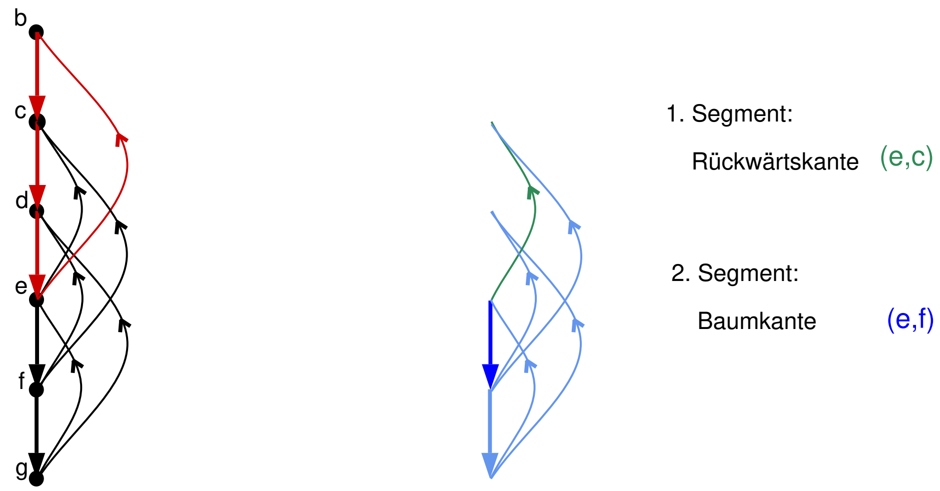


Abbildung 3.10: Problemgraph für die 2. Rekursion *(links)* und aktuelle Segmente *(rechts)*

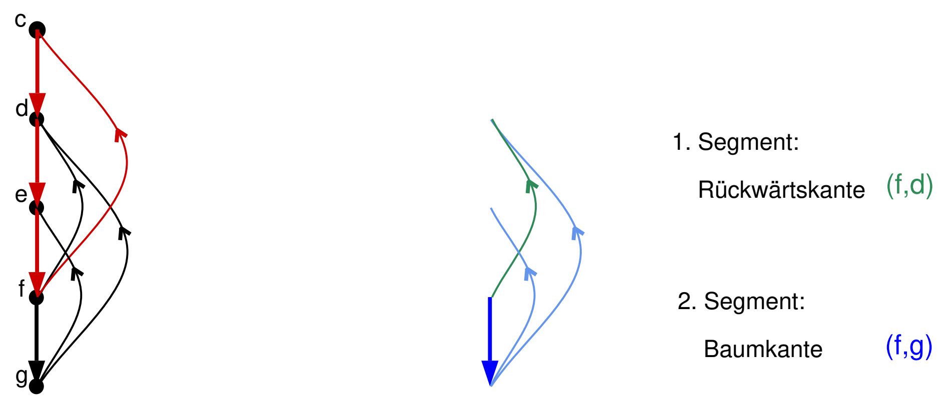


Abbildung 3.11: Problemgraph für die 3. Rekursion *(links)* und aktuelle Segmente *(rechts)*

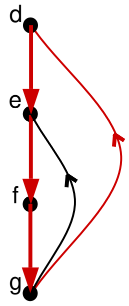


Abbildung 3.12: Problemgraph für die 4. Rekursion

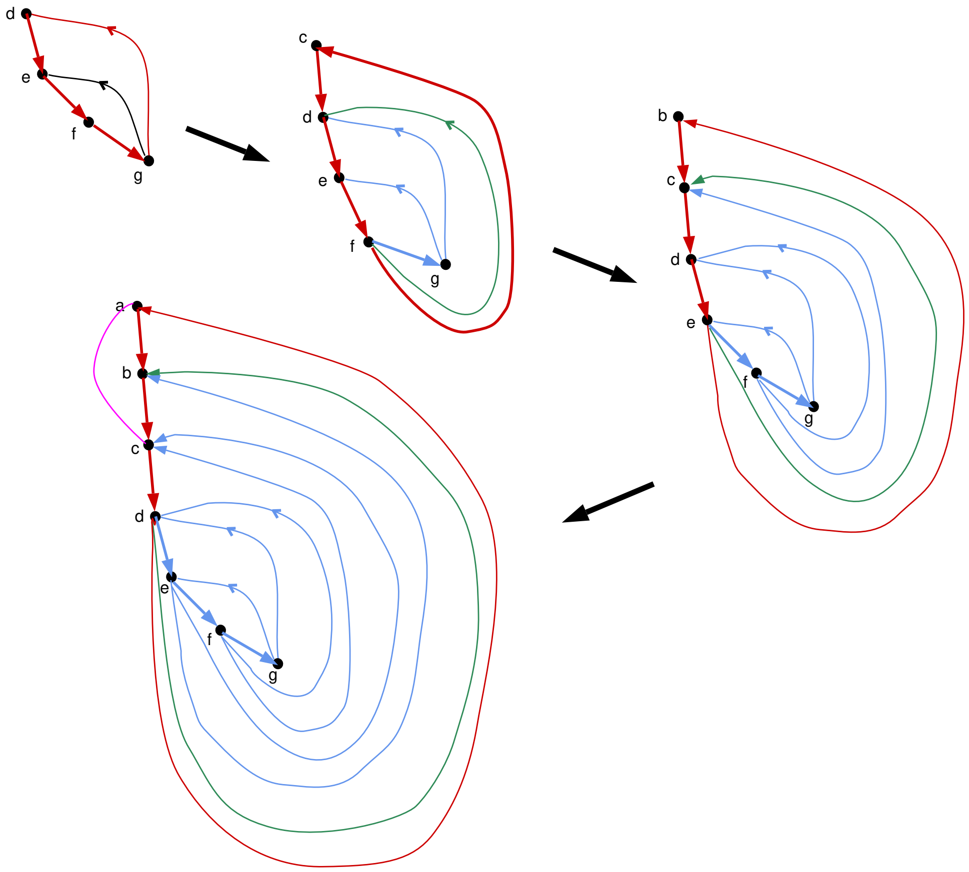


Abbildung 3.13: Abschließende Erzeugung der Einbettung

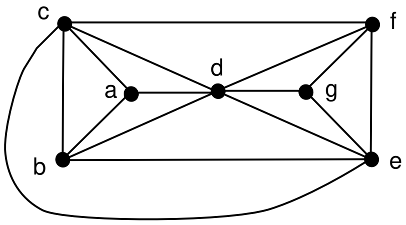


Abbildung 3.14: Alternative Einbettung für den Beispielgraphen

3.3 Separatoren in planaren Graphen

12./13.Juni Ausarbeitung: *Sebastian Heupel*

Mit sog. Devide and Conquer - Methoden lassen sich etliche Optimierungsprobleme einfach lösen.

Dazu bedarf es aber einem Algorithmus zum Aufteilen des Graphen in möglichst unabhängige Teile.

Für Graphenprobleme stellen Separatoren einen geeigneten Teilungsbegriff dar. Allgemein ist ein Separator in einem Graphen eine Teilmenge der Knotenmenge des Graphen, nach deren Entfernung aus dem Graphen dieser in mehrere Komponenten zerfällt.

Wir interessieren uns hier vorallem für eine Zerlegung in zwei gleich große Komponenten. Im allgemeinen ist dies aber a priori nicht möglich. Nach Entfernen von einigen wenigen Knoten, hofft man aber eine solche Zerlegung zu finden.

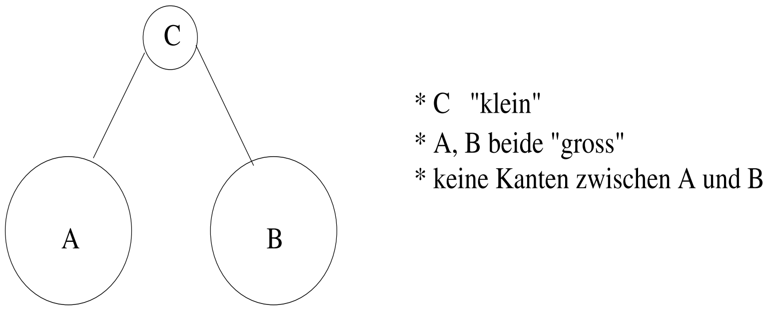


Abbildung 3.15: Separieren eines Graphen

Definition 3.24 *Sei $0 < \alpha < 1$ und $f(n)$ eine Funktion und $G = (V, E)$ ein Graph auf n Knoten. Eine Menge $C \subseteq V$ heißt $\underline{\left(f(n), \alpha\right) - Separator}$, falls*

- $|C| \leq f(n)$
- *$G[V \backslash C]$ in Komponenten zerfällt, die alle Größe $\leq \alpha n$ haben.*

Bemerkung 3.25 *Es bleibt noch zu bemerken, daß nicht alle Graphen (sinnvolle) Separatoren besitzen. Bäume hingegeb besitzen i.a. sehr gute Separatoren.*

Theorem 3.26 *Jeder Baum T besitzt einen $\left(1, \frac{1}{2}\right) - Separator$.*

Beweis: Wir wählen v_0 so, daß die größte Komponente in $T - v_0$ minimal ist. Wir wollen nun zeigen, daß v_0 der gesuchte Separator ist. Dazu nehmen wir das Gegenteil an, nämlich, daß es eine Komponente X in $T - v_0$ mit $|X| > \frac{1}{2}n$ gibt. Wir betrachten dann den (eindeutig bestimmten) Nachbarn v_1 von v_0 in X. Für die größte Komponente in $T - v_1$ gilt offenbar, daß sie

$$\leq max\{|X| - 1, n - |X|\} < |X|$$

ist. Daraus folgt sofort der Widerspruch.

∎

Bemerkung 3.27 *Anhand des Beweises läßt sich auch einfach ein Algorithmus zum Bestimmen eines $\left(1, \frac{1}{2}\right) - Separator$ angeben.*

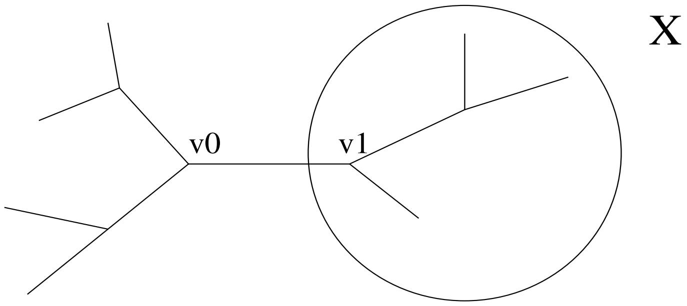


Abbildung 3.16: Skizze zum Beweis

Wir wollen uns jetzt mit Separatoren in planaren Graphen beschäftigen. Wir wissen bereits, daß ein planarer Graph G einen Knoten mit Grad höchstens 5 besitzt. Also enthält jeder planare Graph auf mindestens 5 Knoten eine trennende Menge der Größe höchstens 5. Wir wollen aber den Graphen in etwa gleich große Komponenten zerlegen. Dadurch wird die Aufgabe erheblich erschwert.

Im folgenden wollen wir also folgenden Satz beweisen:

Theorem 3.28 *Jeder planare Graph auf n Knoten besitzt einen* $\left(2\sqrt{2}\sqrt{n}, \frac{2}{3}\right)$*−Separator und ein solcher läßt sich in Zeit* $O(n)$ *bestimmen.*

Zum Beweis des Satzes sind aber noch einige Vorüberlegungen nötig.

Definition 3.29 *Der Durchmesser eines Graphen* $G = (V, E)$ *ist der längste kürzeste Abstand zwischen zwei Knoten, oder formal*

$$diam(G) = max_{u,v \in V}\left(dist(u, v)\right),$$

wobei $dist(x, y)$ *die Länge eines kürzesten Pfades zwischen* x *und* y *angibt.*

Desweiteren spielt folgendes Lemma eine zentrale Rolle beim Beweis obigen Satzes.

Lemma 3.30 *Sei* $G = (V, E)$ *ein planarer Graph und* $T = (V, E_T)$ *ein spannender Baum in G mit Durchmesser s. Dann besitzt G einen* $(s+1, \frac{2}{3})$ *− Separator und ein solcher läßt sich in* $O(n)$ *bestimmen.*

Beweis: Sei G in die Ebene eingebettet und o. E. trianguliert, das heißt, jedes Gebiet wird von genau drei Kanten berandet. Dazu fügen wir einfach solange Kanten ein, bis der Graph trianguliert ist. Dies geht in Zeit $\mathcal{O}(n)$. Offensichtlich ist ein Separator des neuen Graphen auch einer des alten. Sei $e \in E \backslash E_T$ eine Kante. $E_T \cup e$ enthält dann genau einen Kreis $C(e)$ mit höchstens $s + 1$ Knoten, welche die Länge von C angeben. Mit $int\Big(C(e)\Big)$ und $ext\Big(C(e)\Big)$ bezeichnen wir die Anzahl der Knoten im Inneren bzw. im Äußeren von $C(e)$.

Wir suchen nun eine Kante e, so daß sowohl $int\Big(C(e)\Big)$ als auch $ext\Big(C(e)\Big) \leq \frac{2}{3}n$

Dazu beschreiben wir ein Verfahren, daß ausgehend von einer Kante $e \in E \backslash E_T$ mit $int(C(e)) > \frac{2}{3}n$ eine Kante e' konstruiert, so daß

- Das Innere von $C(e')$ echt im Inneren von $C(e)$ enthalten ist.
- $ext\Big(C(e')\Big) \leq \frac{2}{3}n$.

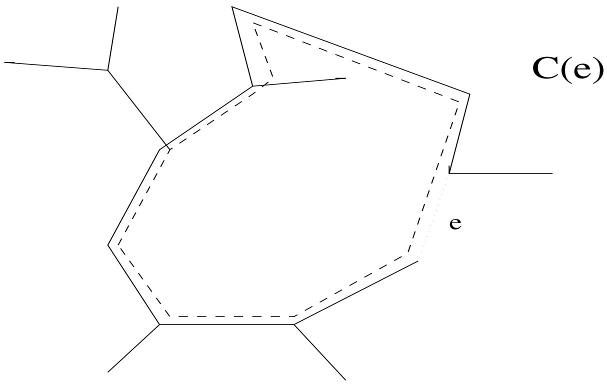
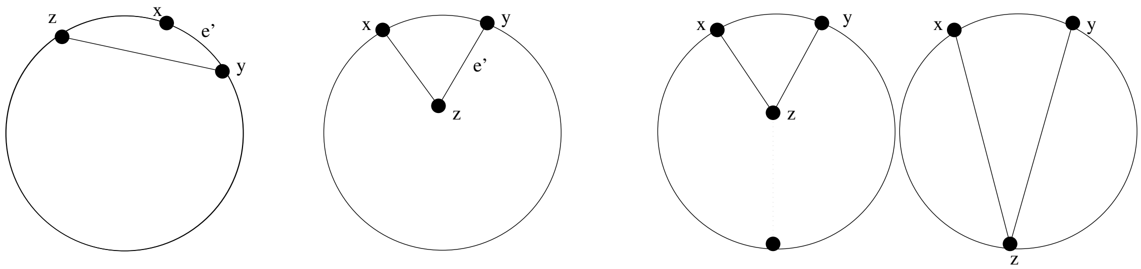


Abbildung 3.17: Skizze zum Beweis

Anzumerken bleibt noch, daß gelten kann $int\Big(C(e^{'})\Big) = int\Big(C(e)\Big)$. Dann enthält das Innere von $C(e^{'})$ aber zumindest eine Fläche weniger als $C(e)$.

Da $int\Big(C(e)\Big)$ nur $O(n)$ viele Flächen (Euler'sche Formel) enthält, terminiert dieser Algorithmus nach linear vielen Schritten und liefert $C(e)$ als gesuchten Separator.
Nun müssen wir uns noch der Bestimmung von $e^{'}$ widmen.

Sei $e = \{x, y\}$ und z der dritte Knoten des Gebiets im Inneren, das e enthält. Da $int\Big(C(e)\Big) > 0$ können nicht sowohl x, z als auch y, z Kanten aus C sein. Bei Verwendung einer geeigneten Datenstruktur kann man z in $O(1)$ bestimmen. Dazu speichert man zu jedem Knoten die inzidenten Kanten in einer zyklischen Liste in der Reihenfolge, wie sie in der Einbettung im Gegenuhrzeigersinn auftreten.



Skizze zu Fall 1 Skizze zu Fall 2a Skizzen zu Fall 2b

Abbildung 3.18: Skizzen zu den folgenden Fällen

Ausgehend von obiger Abbildung unterscheiden wir die folgenden Fälle:

Fall1.

Genau eine der Kanten $\{x, z\}$ oder $\{y, z\}$ ist in $C(e)$ enthalten. Sei $\{x, z\}$ diese Kante. Wenn wir nun $e^{'} := \{y, z\}$ wählen, so umfaßt der neue Kreis genau eine Fläche weniger.

Fall2.

Weder $\{x, z\}$ noch $\{y, z\}$ ist in $C(e)$ enthalten. Daraus ergeben sich mehrere Möglichkeiten.

Fall2a.

Wir betrachten zunächst den Fall, daß $\{x, z\}$ in E_T oder $\{y, z\}$ in E_T enthalten ist. Dann folgt, daß z im Inneren von $C(e)$ liegt, wobei o. E. $\{x, z\}$ in E_T liegt. Also können wir wieder $e^{'} = \{y, z\}$ wählen.

Fall2b.

Nun betrachten wir den Fall, daß sowohl $\{x, z\}$ als auch $\{y, z\}$ nicht in E_T enthalten sind. Wir nehmen nun an, daß $int\Big(C(\{x, z\})\Big) \geq int\Big(C(\{y, z\})\Big)$, dann gilt:

$$
\begin{aligned}
\frac{2}{3}n &< int\Big(C(e)\Big)\\
&= int\Big(C(\{x,z\})\Big) + int\Big(C(\{y,z\})\Big) + \Big|C(\{x,z\}) \cap C(\{y,z\})\Big| - 1\\
&\leq 2int\Big(C(\{x,z\})\Big) + \Big|C(\{x,z\})\Big|
\end{aligned}
$$

und

$$ext\Big(C(\{x,z\})\Big) = n - \underbrace{int\Big(C(\{x,z\})\Big)}_{\geq \frac{1}{2}[int(C(e)) - |C(\{x,z\})|]} - \Big|C(\{x,z\})\Big| \leq n - \frac{1}{3}n - \frac{1}{2}\Big|C(\{x,z\})\Big| \leq \frac{2}{3}n.$$

Somit ergibt sich, daß $e' = \{x,z\}$ die gewünschten Eigenschaften hat.
Jetzt müssen wir uns noch um den Aufwand kümmern, den wir betreiben müssen, um zu zeigen, daß $int\Big(C(\{x,z\})\Big) \geq int\Big(C(\{y,z\})\Big)$ oder umgekehrt. Wir wollen dies in $O(1)$ bestimmen. Dies gelingt uns aber i.a. nicht, denn für eine spezielle Kante e kann der Aufwand sehr hoch sein. Im folgenden sehen wir aber, daß dies nicht so schwer wiegt und wir durch einen geschickte amortisierte Laufzeitanalyse trotzdem die gewünschte Schranke einhalten können.
Für die Bestimmung von z benötigen wir für alle Fälle $2b$ zusammen nur $O(n)$, da jede Baumkante höchstens einmal durchlaufen wird.
Damit können wir uns jetzt der Bestimmung von e' zuwenden. Hierbei genügt es entweder $int(C)$ oder $ext(C)$ zu bestimmen. (vgl. Fall 2b). Desweiteren läßt sich $int\Big(C(\cdot)\Big)$ berechnen, indem man die Baumkanten im Inneren zählt. Also verfolgen wir die Baumkanten abwechselnd im Inneren von $int\Big(C(\{x,z\})\Big)$ und $int\Big(C(\{y,z\})\Big)$, solange bis wir ein Gebiet vollständig durchlaufen haben. Wichtig für die Laufzeit ist, daß die in diesem Gebiet liegenden Baumkanten später nie mehr behandelt werden. Wie wir leicht sehen können ist der Aufwand für das andere Gebiet höchstens genauso groß.
Also ergibt sich auch hier ein Aufwand für alle Fälle von Fall 2b zusammen von $O(n)$.

■

Nach diesen Vorüberlegungen können wir zum Beweis von Satz 3.28 zurückkehren.

Beweis:
Ohne Einschränkung der Allgemeinheit sei G zusammenhängend. Wähle $v_0 \in V$ beliebig als Wurzel und teile die Knoten von G in Mengen $L_i = \{v \in V | dist(v_0, v) = i\}$. Weiter sei $l = max\{i | L_i \neq 0\}$ der maximale Abstand eines Knotens aus G zu v_0.
Falls nun $2l + 1 \leq 2\sqrt{2n}$, dann garantiert Lemma 3.15 den gesuchten Separator.
Anderenfalls setzen wir $s = \lceil \sqrt{\frac{n}{2}} \rceil$ und $S_j := \bigcup\{L_i | i \equiv j \bmod s\}$. Wir wählen nun einen Index j_0 so, daß S_{j_0} die kleineste Menge S_j ist. Für diese gilt dann sicherlich $|S_{j_0}| \leq \lfloor \frac{n}{s} \rfloor$. Wenn wir nun die Komponenten von $G \backslash S_{j_0}$ betrachten können wir zwei Fälle unterscheiden.
<u>1.Fall</u>:
Falls alle Komponenten $\leq \frac{2}{3}n$, dann sind wir wegen $\lfloor \frac{n}{s} \rfloor \leq \sqrt{2}\sqrt{n}$ fertig.
<u>2.Fall</u>:
Es gibt eine (und nur eine!) Komponente $> \frac{2}{3}n$. Wir kontrahieren die Knoten von S_{j_0} in dem Level oberhalb dieser Komoponente. Dadurch entsteht ein BFS-Baum mit Tiefe $\leq s - 1$ und Durchmesser $\leq 2(s-1)$. Aus dem vorherigen Lemma folgt daher, daß diese Komponente einen Separator der Größe $\leq 2s - 1$ hat.
Damit hat G einen Separator der Größe $\leq \lfloor \frac{n}{s} \rfloor + 2s - 1 \leq 2\sqrt{2n}$. Da wir für den BFS-Baum, die Zusammenhängigkeit und zur Berechnung des vorherigen Lemmas jeweils Aufwand $O(n)$ benötigen, läßt sich ein solcher Separator sogar in linearer Zeit bestimmen.

■

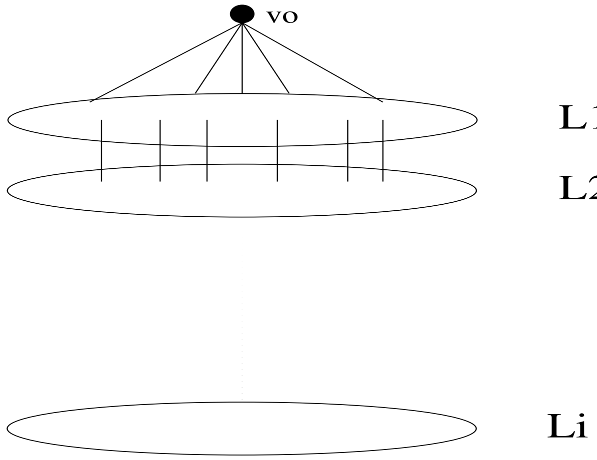


Abbildung 3.19: Skizze zum Beweis

Bemerkung 3.31 *Es läßt sich auch zeigen, daß es in jedem planaren Graph einen*

$$\left(2\sqrt{2n}, \frac{2}{3}\right) - Kreisseparator$$

gibt. D.h., die Knoten des Separators bilden in G einen Kreis.

Abschließend wollen wir nun folgenden Satz zeigen:

Theorem 3.32 *Sei $G = (V, E)$ ein planarer Graph auf n Knoten und $\epsilon = \epsilon(n)$ eine beliebige Funktion, so daß $\epsilon \leq 1$ und $\epsilon n \geq 1$. Dann enthält G einen $\left((2+\sqrt{\frac{3}{2\epsilon n}})\sqrt{6}\sqrt{\frac{n}{\epsilon}}, \epsilon\right) - Separator$. Dieser kann in Zeit $O(n \log \frac{1}{\epsilon})$ bestimmt werden.*

Beweis:
Wir setzen $s = \lceil\sqrt{\frac{\epsilon n}{6}}\rceil$ und wenden folgendes Verfahren an, das eine wiederholte Anwendung des vorigen Algorithmus darstellt.

1.Schritt

Wir benutzen wie im vorigen Theorem Breitensuche um von einem beliebigen Knoten $v_0 \in V$ aus eine Menge S_{j_0} zu bestimmen, so daß $|S_{j_0}| \leq \lfloor\frac{n}{s}\rfloor$ und alle Komponenten von $G[V \backslash S_{j_0}]$ in planare Graphen mit Durchmesser $\leq 2s - 2$ eingebettet werden können. Dann setzten wir $C := S_{j_0}$

2.Schritt

Solange es Komponenten H in $G[V \backslash C]$ der Größe $\geq \epsilon n$ gibt, bestimmen wir mit Hilfe von Lemma 3.30 wie im Beweis von Satz 3.28 eine Menge C' der Größe $\leq 2s - 1$, so daß H nach Entfernen von C' in Komponenten der Größe $\leq \frac{2}{3}|V(H)|$ zerfällt. Dann setzen wir $C := C \cup C'$

Nun bleibt noch zu zeigen, daß C höchstens die angegebene Größe hat und daß die geforderte Laufzeit eingehalten wird. Dazu partitionieren wir alle Konponenten, die im Laufe des Algorithmus auftreten wie folgt.

$N_0 :=$ Menge aller Komponenten $\leq \epsilon n$

$N_i :=$ Menge aller Komponenten $> \left(\frac{3}{2}\right)^{i-1} \epsilon n$ und $\leq \left(\frac{3}{2}\right)^{i} \epsilon n$ für $1 \leq i \leq log_{\frac{3}{2}} \frac{1}{\epsilon}$.

Falls im Laufe des Algorithmus eine Komponente H aus N_i in kleinere zerfällt, so haben diese höchstens $\frac{2}{3}|V(H)|$ Knoten und gehören daher nicht zu N_i. Also sind die Komponenten in N_i paarweise knotendisjunkt, woraus folgt, daß die Laufzeit, die wir benötigen um Komponenten in N_i zu separieren linear ist. Daher beträgt die Gesamtlaufzeit $O(n \cdot \log \frac{1}{\epsilon})$.

Nun müssen wir noch die Größe von C abschätzen. Da die Komponenten in N_i knotendisjunkt sind, gilt

$$|N_i|\left(\frac{3}{2}\right)^{i-1}\epsilon n \leq n$$

Also gibt es höchstens $\lfloor\left(\frac{2}{3}\right)^{i-1}\frac{1}{\epsilon}\rfloor$ Komponenten in N_i. Also ist

$$|C| \leq \left\lfloor\frac{n}{s}\right\rfloor + \sum_{i=1}^{log_{\frac{3}{2}}\frac{1}{\epsilon}} \left\lfloor\left(\frac{2}{3}\right)^{i-1}\frac{1}{\epsilon}\right\rfloor(2s-1) \leq \ \ldots \ \leq \sqrt{6}\sqrt{\frac{n}{\epsilon}}(1+1+\sqrt{\frac{3}{2\epsilon n}}).$$

■

Kapitel 4

Matching Probleme

....

26./27.Juni Ausarbeitung: *Markus Ebersberger*

Definition 4.1 *Sei $G = (V, E)$ ein Graph; M ein Matching in G.*

- *Ein Pfad heißt M-*alternierend *(oder kurz* alternierend*), falls er abwechselnd Kanten aus M und $E \setminus M$ enthält.*
- *Ein Pfad heißt M-*augmentierend *(kurz* augmentierend*), falls er M-alternierend ist und beide Endknoten ungematcht sind.*

Proposition 4.2 *Ist P ein M-augmentierender Pfad für ein Matching M, dann gilt:*

$$M' := M \oplus P \quad [= (M \setminus P) \cup (P \setminus M)]$$

ist ein Matching mit $|M'| = |M| + 1$.
(Anders ausgedrückt: Tauscht man entlang eines augmentierenden Pfades Matching- und Nicht-Matching-Kanten aus, so erhält man ein um eine Kante größeres Matching).

Beweis:

- M' ist ein Matching:
  Angenommen, dies ist nicht der Fall. Dann gibt es $e_1, e_2 \in M'$ mit $e_1 \cap e_2 \neq \emptyset$. Folgende drei Konstellationen sind möglich:
  1. $e_1 \in M$ und $e_2 \in M \quad \longrightarrow$ Widerspruch dazu, daß M ein Matching ist.
  2. $e_1 \in M$ und $e_2 \notin M \Rightarrow e_1 \notin P, e_2 \in P$. Sei Q der gemeinsame Knoten von e_1 und e_2. Q kann nicht der Endpunkt von P sein, da Q in M gematcht und P augmentierend bzgl. M sein soll. Da $e_1 \notin P$, muß folglich von Q eine weitere Kante $e_3 \in P$ ausgehen, die (da P alternierend ist) in M liegen muß. Dies ist jedoch ein Widerspruch zur Voraussetzung, daß M ein Matching ist.
  3. $e_1 \notin M$ und $e_2 \notin M \Rightarrow e_1 \in P, e_2 \in P \quad \longrightarrow$ Widerspruch dazu, daß P M-alternierend ist.
- $|M'| = |M| + 1$:
  Da P alternierend ist und beide Endknoten ungematcht sind, muß gelten:
  - $|P|$ ist ungerade
  - $|P \setminus M| = |P \cap M| + 1$

Daher gilt $|M'| = |M| - |M \cap P| + |P \setminus M| = |M| + 1$.

■

Theorem 4.3 *Ein Matching M ist genau dann maximal, wenn es keinen M-augmentierenden Pfad gibt.*

Beweis:

„$\Rightarrow$" Ergibt sich unmittelbar aus Proposition 4.2.

„$\Leftarrow$" Sei M ein Matching, das nicht maximal ist. Wir zeigen: es gibt einen M-augmentierenden Pfad. Sei $M_{\max}$ ein (beliebiges) maximales Matching. Betrachte den Subgraphen $M \oplus M_{\max}$ (genauer: den durch die Kantenmenge $M \oplus M_{\max}$ induzierten Subgraphen). Es gilt

- Jeder Knoten hat Grad ≤ 2, d. h. $M \oplus M_{\max}$ ist Vereinigung von Kreisen und Pfaden.
- Jeder Kreis ist von gerader Länge und enthält die gleiche Anzahl von Kanten aus M wie aus $M_{\max}$.
- Jeder Pfad enthält abwechselnd Kanten aus M und $M_{\max}$.

Wegen $|M_{\max}| > |M|$ muß es einen Pfad P geben, der mehr Kanten aus $M_{\max}$ als aus M enthält. Anfangs- und Endkanten von P gehören daher zu $M_{\max}$, d. h. die Anfangs- und Endknoten von P sind bezüglich M ungematcht. Also ist P M-augmentierend.

■

Korollar 4.4 *Sei t die Kardinalität eines maximalen Matchings und M ein Matching mit $|M| < t$. Dann gibt es einen M-augmentierenden Pfad, der höchstens die Länge*

$$2\left\lfloor \frac{|M|}{t - |M|} \right\rfloor + 1$$

hat.

Beweis: Betrachte wie im Beweis von Theorem 4.3 den Subgraphen $M \oplus M_{\max}$. Da ein Pfad in diesem Graphen höchstens eine Kante mehr aus $M_{\max}$ als aus M enthalten kann, enthält $M \oplus M_{\max}$ mindestens $s := |M_{\max}| - |M|$ Pfade P_i $(i = 1, \ldots, s)$, die M-augmentierend sind. Sei $l := \min_{1 \leq i \leq s} |E(P_i)|$. Aus

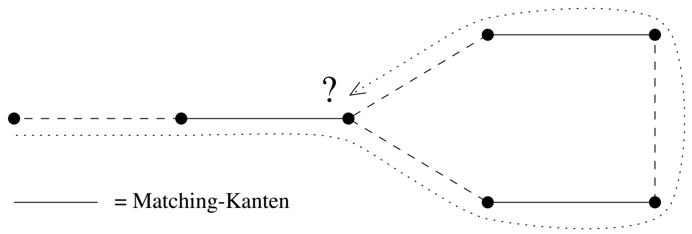
$$\frac{l-1}{2} \cdot s \;\leq\; \sum_{i=1}^{s} \frac{|E(P_i)| - 1}{2} \;\leq\; |M|$$

folgt $l \;\leq\; 2(|M|/s) + 1$, und somit die Behauptung. ■

Ansatz für einen Matching-Algorithmus:

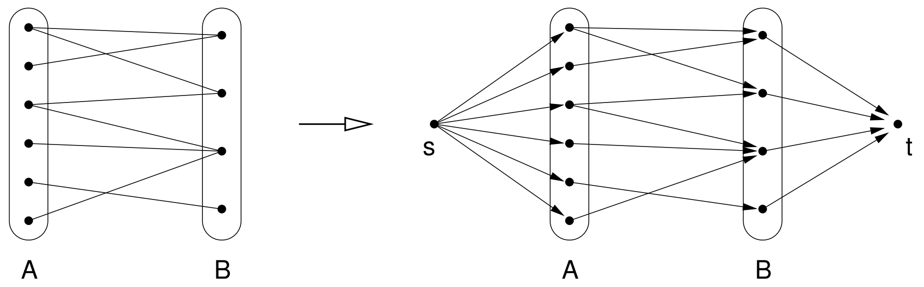
1. Starte mit einem beliebigen Matching (beispielsweise mit dem leeren Matching oder einem, das der Greedy-Algorithmus gefunden hat).
2. Vergrößere das Matching sukzessive entlang augmentierender Pfade (Wie in Proposition 4.2 beschrieben).

Problem: Wie findet man augmentierende Pfade ?
Die naheliegende Idee, bei einem beliebigen nicht gematchten Knoten anzufangen und einen von dort ausgehenden alternierenden Pfad zu verfolgen, bis man auf einen weiteren ungematchten Knoten trifft, kann in eine Sackgasse führen, wie nachfolgende Abbildung zeigt.



4.1 Matching in bipartiten Graphen

Das Matchingproblem in bipartiten Graphen $G = (A \cup B, E)$ entspricht einem Flußproblem mit Kapazitäten $\equiv 1$:



- Richte alle Kanten von A nach B.
- Verbinde einen zusätzlichen Knoten s (Quelle) durch gerichtete Kanten mit allen Knoten in A; verbinde alle Knoten in B durch gerichtete Kanten mit einem zusätzlichen Knoten t (Senke).
- Gebe allen Kanten Kapazität identisch Eins.

Offenbar entspricht dann ein maximales Matching im ursprünglichen Graphen genau einem (ganzzahligen) maximalen Fluß im neuen Graphen.

Theorem 4.5 *In bipartiten Graphen ist die Bestimmung eines maximalen Matchings in $\mathcal{O}(nm)$ Schritten möglich.*

Beweis: Überführe das Matchingproblem wie oben beschrieben in ein entsprechendes Flußproblem. Dieses läßt sich (beispielsweise mit dem Algorithmus von Ford und Fulkerson) innerhalb der geforderten Zeitschranke lösen. ■

Ziel: $\mathcal{O}(\sqrt{n}m)$ - Algorithmus.

Idee:
Versuche, ausgehend von den ungematchten Knoten in A, kürzeste augmentierende Pfade zu finden (da in bipartiten Graphen keine ungeraden Zyklen existieren, tritt das oben erwähnte „Sackgassenproblem“ hier nicht auf), wenn möglich mehrere „gleichzeitig“.

Lemma 4.6 *Sei P ein kürzester M-augmentierender Pfad für ein Matching M und $\widetilde{P}$ ein beliebiger $(M \oplus P)$-augmentierender Pfad. Dann gilt*

$$|\widetilde{P}| \geq |P| + 2|\widetilde{P} \cap P| \geq |P|$$

Beweis: Sei $\widetilde{M} := (M \oplus P) \oplus \widetilde{P}$. Dann gilt (nach Proposition 4.2) $|\widetilde{M}| = |M| + 2$, und $\widetilde{M} \oplus M$ enthält mindestens zwei M-augmentierende Pfade, sagen wir P_1 und P_2.
Also gilt $|\widetilde{P} \oplus P| = |\widetilde{M} \oplus M| \geq |P_1| + |P_2| \geq 2|P|$, da P ein kürzester M-augmentierender Pfad ist. Andererseits ist $|\widetilde{P} \oplus P| = |\widetilde{P}| + |P| - 2|\widetilde{P} \cap P|$. Zusammen $|P| \leq |\widetilde{P}| - 2|\widetilde{P} \cap P|$. ■

Lemma 4.7 *Sei $M_1, \ldots, M_{k+1}$ eine Folge von Matchings und $P_1, \ldots, P_k$ eine Folge von augmentierenden Pfaden, wobei gelte*

- $M_1 = \emptyset$
- *P_i ist ein kürzester M_i-augmentierender Pfad $(i = 1, \ldots, k)$*

- $M_{i+1} = M_i \oplus P_i$ *(i = 1, ..., k)*
- M_{k+1} *ist ein maximales Matching*

Dann gilt

a) $|P_{i+1}| \geq |P_i|$ $(i = 1, \ldots, k)$

b) $|P_i| = |P_j|, i \neq j \;\Rightarrow\; P_i$ *und* P_j *sind knotendisjunkt.*

c) $|\{|P_1|, \ldots, |P_k|\}| \leq \sqrt{2n} + 1$

Beweis:

a) Folgt direkt aus Lemma 4.6.

b) Angenommen, dies gilt nicht. Dann wähle $i < j$ mit $|P_i| = |P_j|$, so daß P_i und P_j *nicht* knotendisjunkt sind, und $j - i$ minimal bezüglich dieser Eigenschaften ist. Dann gilt für alle $i < k < j$: $|P_k| = |P_i|$ und P_k ist knotendisjunkt zu P_i und P_j. Setze $\widetilde{M} := M_i \oplus P_{i+1} \oplus P_{i+2} \oplus \cdots \oplus P_{j-1}$. Da P_i knotendisjunkt zu allen $P_{i+1}, \ldots, P_{j-1}$ ist, ist $\widetilde{M}$ ein Matching, P_i ein kürzester $\widetilde{M}$-augmentierender Pfad und P_j ein $(\widetilde{M} \oplus P_i)$-augmentierender Pfad. Nach Lemma 4.6 gilt: $|P_i| = |P_j| \geq |P_i| + 2|P_i \cap P_j|$ $\Rightarrow |P_i \cap P_j| = 0$, d. h. P_i und P_j sind kantendisjunkt.
Sei nun v beliebig $\in V(P_i) \cap V(P_j)$. v kann kein Endknoten von P_j sein, da jeder Knoten aus $V(P_i)$ in $\widetilde{M} \oplus P_i$ gematcht ist, er kann aber auch kein innerer Knoten sein (da P_j alternierend ist, wäre sonst v ein gemeinsamer Knoten zweier Matching- Kanten in $\widetilde{M} \oplus P_i$). Folglich ist obiger Durchschnitt leer, im Widerspruch zur Annahme.

c) Es gilt $k = |M_{k+1}|$ (=Kardinalität eines maximalen Matchings).

$$|P_{\lfloor k-\sqrt{k}\rfloor}| \overset{\text{Kor. 4.4}}{\leq} 2\left\lfloor \frac{|M_{\lfloor k-\sqrt{k}\rfloor}|}{k - |M_{\lfloor k-\sqrt{k}\rfloor}|} \right\rfloor + 1 \;=\; 2\left\lfloor \frac{\lfloor k-\sqrt{k}\rfloor - 1}{k - (\lfloor k-\sqrt{k}\rfloor - 1)} \right\rfloor + 1 \;=$$

$$= \; 2\left\lfloor \frac{k - \lceil\sqrt{k}\rceil - 1}{\lceil\sqrt{k}\rceil + 1} \right\rfloor + 1 \;\leq\; 2\sqrt{k} - 1$$

Also sind $|P_1|, \ldots, |P_{\lfloor k-\sqrt{k}\rfloor}| \in \{1, 3, 5, \ldots, 2\lfloor\sqrt{k}\rfloor - 1\}$. Insgesamt gibt es daher höchstens

$$\underbrace{\left\lceil \tfrac{2\lfloor\sqrt{k}-1\rfloor}{2} \right\rceil}_{\text{Pfade } |P_1|, \ldots, |P_{\lfloor k-\sqrt{k}\rfloor}|} \;+\; \underbrace{\left\lceil \sqrt{k} \right\rceil}_{|P_{\lfloor k-\sqrt{k}\rfloor+1}|, \ldots, |P_k|} \;\leq 2\sqrt{k} + 1 \overset{k \leq n/2}{\leq} \sqrt{2n} + 1$$

verschiedene Pfadlängen.

■

Algorithmus
Gehe in „Phasen“ vor. Bestimme pro Phase *alle* kürzesten augmentierenden Pfade: Bestimme zunächst die Länge l eines kürzesten augmentierenden Pfades und dann eine (inklusions-) maximale Menge knotendisjunkter augmentierender Pfade der Länge l.

<u>BIPARTITES MATCHING</u>
Eingabe: zusammenhängender, bipartiter Graph $G = (A \cup B, E)$
Ausgabe: maximales Matching $M \subseteq E$
$M := \emptyset$;
`loop:`
    for all $v \in A \cup B$ **do**   label[v] := 0;
    // beginne BFS bei allen ungematchten Knoten von A

$R := \emptyset$; $l := 1$;
for all $v \in A$, v ungematcht **do**
  for all $\{v, w\} \in E$ **do**
    label[w] := l; $R := R \cup \{w\}$;
while R $\neq \emptyset$ **and** R enthält keinen ungematchten Knoten **do**
  // folge gematchten Kanten aus R zu Knoten aus A
  $L := \emptyset$; $l := l + 1$;
  for all $w \in R$, $\{v, w\} \in$ **do**
    $L := L \cup \{v\}$; label[v] := l;
  // folge ungematchten Kanten aus L zu Knoten aus B
  $R := \emptyset$; $l := l + 1$;
  for all $v \in L$, $\{v, w\} \in E \setminus M$ **do**
    if label[w] = 0 **then**
      $R := R \cup \{w\}$; label[w] := l;
if R = $\emptyset$ **then** STOP; // M ist maximales Matching;
// Starte bei jedem ungematchten Knoten aus A einen DFS
Markiere alle Knoten als "unchecked";
for all $v \in A$, v ungematcht **do**
  Suche augmentierenden Pfad P Länge $\leq l$ mit
    Startknoten v, betrachte dabei nur Knoten, die als "unchecked" markiert sind und gehe DFS-mäßig vor; markiere alle bei der Suche betrachteten Knoten als "checked";
  if ein Pfad P gefunden wurde **then**
    $M := M \oplus P$; // augmentiere entlang P
goto `loop`;

Theorem 4.8 *Der Algorithmus* BIPARTITES MATCHING *findet in* $\mathcal{O}(\sqrt{n}m)$ *Schritten ein maximales Matching.*

Beweis:

Korrektheit: Im ersten Teil jedes Schleifendurchlaufs wird zunächst die Länge eines kürzesten augmentierenden Pfades bestimmt. Da augmentierende Pfade gleicher Länge knotendisjunkt sind (Lemma 4.7 b), werden im zweiten Teil des Schleifendurchlaufs alle augmentierenden Pfade dieser Länge gefunden.

Laufzeit: Wegen Lemma 4.7 c) gibt es höchstens $\mathcal{O}(\sqrt{n})$ Schleifendurchläufe. Jeder Durchlauf besteht aus einem (modifizierten) BFS und einem (modifizierten) DFS. Jede Kante wird also nur $\mathcal{O}(1)$ mal betrachtet. Daher ist die Laufzeit pro Durchlauf $\mathcal{O}(m)$.

■

Der Algorithmus BIPARTITES MATCHING bestimmt in jedem bipartiten Graphen ein maximales Matching. Für welche bipartite Graphen gibt es nun aber ein Matching, das alle Knoten einer Klasse überdeckt? Hierfür geben wir abschliessend noch ein recht einfaches Kriterium an, das in der Literatur auch unter dem Namen „Heiratssatz“ bekannt ist.

Theorem 4.9 *Sei* $G = (A \cup B, E)$ *ein bipartiter Graph mit* $|A| \leq |B|$. *Dann gilt:*
Es existiert ein Matching, das ganz A überdeckt $\iff$ $|\Gamma(X)| \geq |X| \quad \forall X \subseteq A$.

Beweis:

„$\Rightarrow$“ Dies ist unmittelbar einsichtig.

„$\Leftarrow$“ Sei M ein maximales Matching in G. Setze

$$\begin{aligned} X_0 &= \{u \in A \mid u \text{ ungematcht }\} \\ X &= \{u \in A \mid \exists\ s \in X_0 \text{ und } \exists \text{ alternierender Pfad von } s \text{ nach } u\} \\ Y &= \{u \in B \mid \exists\ s \in X_0 \text{ und } \exists \text{ alternierender Pfad von } s \text{ nach } u\} \end{aligned}$$

Dann gilt offensichtlich:

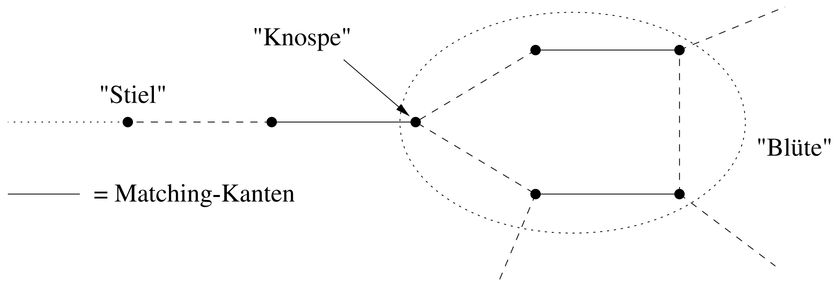
- $\Gamma(X) \subseteq Y$
- Jeder Knoten in Y ist gematcht (sonst gäbe es einen augmentierenden Pfad im Widerspruch zur Maximalität von M) $\Rightarrow$ $|X \setminus X_0| = |Y|$.

Also $|X_0| + \underbrace{|X \setminus X_0|}_{=|Y|} = |X| \leq |\Gamma(X)| \leq |Y| \Rightarrow |X_0| = 0.$

■

4.2 Matching in allgemeinen Graphen

Dieses Problem ist, vor allem wegen der Existenz ungerader Kreise, schwieriger als das Matching in bipartiten Graphen.



Kapitel 5

Lineare Programmierung

18./24.Juli Ausarbeitung: *Ulrich Voll*

5.1 Dualitätstheorie

Dualer Simplexalgorithmus

GEGEBEN: Simplex Tableau

$$T = \begin{pmatrix} \bar{A} = B^{-1}A & \bar{b} = B^{-1}b \\ \bar{c} = c^T - c_B^T B^{-1} A & -c_0 = -c_B^T B^{-1} b, \end{pmatrix}$$

wobei B dual zulässige Basis (d.h., $\bar{c} \geq 0$).
Wiederhole, bis eine der Abbruchbedingungen erfüllt ist:

- (Optimalitätsprüfung)
  Gilt $\bar{b} \geq 0 \quad \longrightarrow$ STOP: Lösung x mit $x_B = \bar{b}$ und $x_N = 0$ ist auch primal zulässig und daher optimal;
- (Bestimmung einer Pivotzeile)
  Wähle einen (beliebigen) Index $r \in \{1, \ldots, m\}$ mit $\bar{b}_r < 0$;
  [verschiedene Auswahlstrategien ...]
- (Prüfung auf Beschränktheit des Optimums)
  Gilt $\bar{A}_{r\cdot} \geq 0 \quad \longrightarrow$ STOP: duales Problem ist unbeschränkt und primales hat daher keine zulässige Lösung
- (Bestimmung der Pivotspalte)
  Wähle einen (beliebigen) Index $s \in \{1, \ldots, n\}$ mit

  $$\frac{\bar{c}_s}{\bar{a}_{rs}} = \max\{\frac{\bar{c}_t}{\bar{a}_{ts}} \mid t \in \{1, \ldots, n\} \text{ und } \bar{a}_{ts} < 0\};$$

  [verschiedene Auswahlstrategien ...]
- (Pivotoperation, Basistausch)
  bzgl $\bar{a}_{rs}$, wie im primalen Algorithmus

Abbildung 5.1: Dualer Simplexalgorithmus.

Betrachte die zueinander dualen linearen Programme

$$\begin{array}{lll} (P) & min & c^T x \\ & & Ax \quad = b \\ & & x \quad \geq 0 \end{array} \qquad \text{und} \qquad \begin{array}{lll} (D) & max & y^T b \\ & & y^T A \quad \leq c^T \\ & & \end{array} .$$

Sei B eine Basis, die dual zulässig ist, nicht aber primal zulässig, also
$y^T := c_B^T B^{-1}$ erfüllt $y^T A \leq c^T$, aber
$x := (x_B, x_N) = (B^{-1}b, 0)$ erfüllt *nicht* $x \geq 0$ (d. h. $B^{-1}b \not\geq 0$).

Der duale Simplexalgorithmus (vgl. Abbildung 5.1) erzeugt ausgehend von B einen Folge von „immer besseren“ dual zulässigen Basen. Sobald eine Basis gefunden wird, die auch primal zulässig ist, wird abgebrochen.

Bemerkung 5.1 *Wenn auch in der Beschreibung des Algorithmus in Abbildung 5.1 das* primale *Tableau manipuliert wird, entspricht der Ablauf des dualen Simplexalgorithmus auf* (P) *dem des primalen auf* (D)*. Alle Begriffe, die man für den primalen Simplexalgorithmus gebildet hat, lassen sich deshalb geeignet auf den dualen übertragen.*

Theorem 5.2 *Der duale Simplexalgorithmus ist korrekt und terminiert bei Verwendung der Indexstrategie von Bland in endlicher Zeit.*

Beweis: Wenn man die obige Bemerkung ausführt, kann man den Beweis auf Theorem ?? reduzieren. Hier soll zumindest für die Korrektheit direkt argumentiert werden.
Wir wissen bereits: Ist $B_{\text{alt}} = (A_{i_1}, \cdots, A_{i_m})$, so gilt

$$B : [= B_{\text{neu}}] = (A_{i_1}, \cdots, A_{i_{r-1}}, A_q, A_{i_{r-1}}, \ldots, A_{i_m}),$$

und das neue Tableau T_{neu} entspricht dem Simplextableau zur Basis $B[= B_{\text{neu}}]$:

$$T_{\text{neu}} = \begin{pmatrix} \bar{A} = B^{-1}A & \bar{b} = B^{-1}b \\ \bar{c} = c^T - c_B^T B^{-1}A & -c_0 = -c_B^T B^{-1}b, \end{pmatrix}.$$

Zu zeigen ist:

1. Gilt $\bar{b} \geq 0$, so ist $x := (x_B, x_N) = (B^{-1}b, 0)$ eine optimale Lösung.
2. Gilt $\bar{A}_{r.} \geq 0$, so ist das duale Problem unbeschränkt.
3. $B[= B_{\text{neu}}]$ ist wieder dual zulässige Basis.

Ad 1. $x := (x_B, x_N) = (B^{-1}b, 0)$ ist primal zulässige Lösung, $y^T := c_B^T B^{-1}$ ist dual zulässige Lösung, beide haben den gleichen Zielfunktionswert $y^T = c_B^T B^{-1}b$. Aus Satz ?? folgt daher: x und y sind optimal.

Ad 2. Schreibe

$$B^{-1} =: \begin{pmatrix} t_1^T \\ \vdots \\ t_m^T \end{pmatrix}$$

und betrachte nun $y_\lambda^T = c_B^T B^{-1} - \lambda t_r^T$. Für alle $\lambda \geq 0$ gilt dann

$$y_\lambda^T A = c_B^T B^{-1}A - \lambda \underbrace{t_r^T A}_{= \bar{A}_{r.} \geq 0} \leq \underbrace{c_B^T B^{-1} A}_{\leq c^T},$$

wegen der Voraussetzung und der dualen Zulässigkeit von B. h. y_λ ist für alle $\lambda \geq 0$ dual zulässig, und

$$y_\lambda^T b = c_B^T B^{-1}b - \lambda \underbrace{t_r^T b}_{= \bar{b}_r < 0} \xrightarrow{\lambda \to \infty} \infty.$$

Ad 3. Entsprechend der Pivotoperation gilt: $\bar{c}_{\text{neu}} = \bar{c} - \frac{\bar{c}_s}{\bar{a}_{rs}} \bar{a}_{r\cdot} \geq 0$ nach Wahl von s. Um dies einzusehen, betrachte die j–te Komponente $\bar{c}_j - \frac{\bar{c}_s}{\bar{a}_{rs}} \bar{a}_{rj}$.

Falls $\bar{a}_{rs} \geq 0$, gilt $\bar{c}_j - \frac{\bar{c}_s}{\bar{a}_{rs}} \bar{a}_{rj} \geq \bar{c}_j \geq 0$,
gilt andererseits $\bar{a}_{rs} \leq 0$, dann ist $\bar{c}_j - \frac{\bar{c}_s}{\bar{a}_{rs}} \bar{a}_{rj} \geq 0 \Leftrightarrow \frac{\bar{c}_j}{\bar{a}_{rj}} \leq \frac{\bar{c}_s}{\bar{a}_{rs}}$, wobei auf der rechten Seite jeweils die Were des *alten* Tableaus stehen.

∎

Im Folgenden stelle wir drei mögliche Anwendungen des dualen Simplexalgorithmus vor:

$$
\begin{array}{lrrrcr}
\min & 3x_1 & +3x_2 & +6x_3 & & \\
 & x_1 & +2x_2 & 4x_3 & \leq & 14 \\
 & -2x_1 & -x_2 & -5x_3 & \leq & -25 \\
 & x_1 & & 2x_3 & \leq & 14 \\
 & & & x_i & \geq & 0
\end{array}
$$

		x_1	x_2	x_3	s_1	s_2	s_3	
T_0:	s_1	1	2	4	1	0	0	14
	s_2	-2	-1	$\boxed{-5}$	0	1	0	-25
	s_3	1	0	2	0	0	1	14
		3	3	6	0	0	0	0
T_1:	s_1	$\boxed{-\frac{3}{5}}$	$\frac{6}{5}$	0	1	$\frac{4}{5}$	0	-6
	x_2	$\frac{2}{5}$	$\frac{1}{5}$	1	0	$-\frac{1}{5}$	0	5
	s_3	$\frac{1}{5}$	$-\frac{2}{5}$	0	0	$\frac{2}{5}$	1	4
		$\frac{3}{5}$	$\frac{9}{5}$	0	0	$\frac{6}{5}$	0	-30
T_2:	x_1	1	-2	0	$-\frac{5}{3}$	$-\frac{4}{3}$	0	10
	x_3	0	1	1	$\frac{2}{3}$	$\frac{1}{3}$	0	1
	s_3	0	1	0	1	$\frac{2}{3}$	1	2
		0	3	0	1	2	0	-36

$\Longrightarrow x = (10, 0, 1)$ ist optimal!

Abbildung 5.2: Beispiel zur Anwendung des dualen Simplexalgorithmus.

1. Wenn in

$$
\begin{array}{lll}
(P) & min & c^T x \\
 & & Ax \quad = b \\
 & & x \quad \geq 0
\end{array}
$$

$c \geq 0$, aber nicht $b \geq 0$, dann ist $y \equiv 0$ dual zulässig. Der duale Simplexalgorithmus kann also sofort gestartet werden, wohingegen der primale die Phase I benötigt, um überhaupt eine zulässige Startlösung zu finden. Vgl. das Beispiel in Abbildung 5.2.

2. Bei nachträglicher Änderung der rechten Seite ändert sich an der dualen Zulässigkeit einer Lösung nichts. Da bei geringfügiger Änderung der rechten Seite b das neue Optimum „wahrscheinlich in der Nähe des alten Optimums liegt“ , bietet es sich in diesem Fall an, die aktuelle Basis beizubehalten, anstatt den Algorithmus völlig neu zu starten, wie man das bei Verwendung des primalen Algorithmus tun müßte.

3. Wenn nachträglich Ungleichungen hinzukommen, wie das etwa beim *Schnittebenenverfahren* der Fall ist, bildet die alte Basis zusammen mit den Schlupfvariablen der hinzugefügten Ungleichungen eine dual zulässige Lösung. Diese ist nicht notwendigerweise auch primal zulässig.

Mit dem folgenden Satz lassen sich Paare von primal/dual–optimalen Lösungen charakterisieren.

Theorem 5.3 (Satz vom schwachen komplementären Schlupf.) *Gegeben das Paar zueinander dualer linearer Programme*

$$
\begin{array}{llll}
(P) & min & c^T x & \\
& & Ax & = b \\
& & x & \geq 0
\end{array}
\qquad und \qquad
\begin{array}{llll}
(D) & max & y^T b & \\
& & y^T A & \leq c^T \\
& & & .
\end{array}
$$

Sei x zulässige Lösung von (P) und y zulässige Lösung von (D). Dann gilt

$$x \text{ optimal für } (P) \text{ und } y \text{ optimal für } (D). \quad \Longleftrightarrow \quad \forall i = 1, \cdots, n : (c_i - y^t A_{.i}) x_i = 0. \tag{5.1}$$

Beweis: Da x und y jeweils zulässige Lösungen sind, gilt

$$\forall i = 1, \cdots, n : (c_i - y^t A_{.i} \geq 0 \text{ und } x_i \geq 0).$$

Also ist

$$0 \leq \sum_{i=1}^{n} (c_i - y^t A_{.i}) x_i = c^T x - y^T A x = c^T x - y^T b.$$

Nach Satz ?? gilt darum:

$$x \text{ optimal für } (P) \text{ und } y \text{ optimal für } (D). \quad \Longleftrightarrow \quad c^T x = y^T b.$$

■

5.2 Primal–duale Methode

Sei wieder

$$
\begin{array}{llll}
(P) & min & c^T x & \\
& & Ax & = b \\
& & x \geq 0 &
\end{array}
\qquad \text{und} \qquad
\begin{array}{llll}
(D) & max & y^T b & \\
& & y^T A & \leq c^T
\end{array}
$$

gegeben. Die primal–duale Methode (vgl. Abbildung 5.3) ...
... betrachtet *gleichzeitig* primale und duale Probleme
... und verwendet die Bedingungen des komplementären Schlupfs (5.1).

Sei x zulässige Lösung von (P) und y zulässige Lösung von (D). Dann gilt:

$$x \text{ optimal für } (P) \text{ und } y \text{ optimal für } (D) \quad \Longleftrightarrow \quad \begin{cases} \text{i)} & y^T A_{.i} < c_i \Rightarrow x_i = 0 \\ \text{ii)} & x_i > 0 \Rightarrow y^T A_{.i} = 0 \end{cases}$$

Gegeben ein dual zulässiger Vektor y_0, setze

$$S := \{i |\ 1 \leq i \leq n, y_0^T A_{.i} = c_i\}.$$

Mit Hilfe von S definiere die eingeschränkten Probleme

$$
\begin{array}{llll}
(RP) & min & \sum_{i=1}^{n} z_i & \\
& & Ax + z & = b \\
& & x_i & \begin{cases} \geq 0 & i \in S \\ = 0 & i \notin S \end{cases} \\
& & z & \geq 0
\end{array}
\qquad \text{und} \qquad
\begin{array}{lllll}
(RD) & max & u^T b & & \\
& & u^T A_{.i} & \leq 0 & i \in S \\
& & u & \leq 1 & \\
& & & . &
\end{array}
$$

(RD) ist das duale Problem zu (RP), aus dem die Variablen mit Index $i \notin S$ gestrichen wurden.

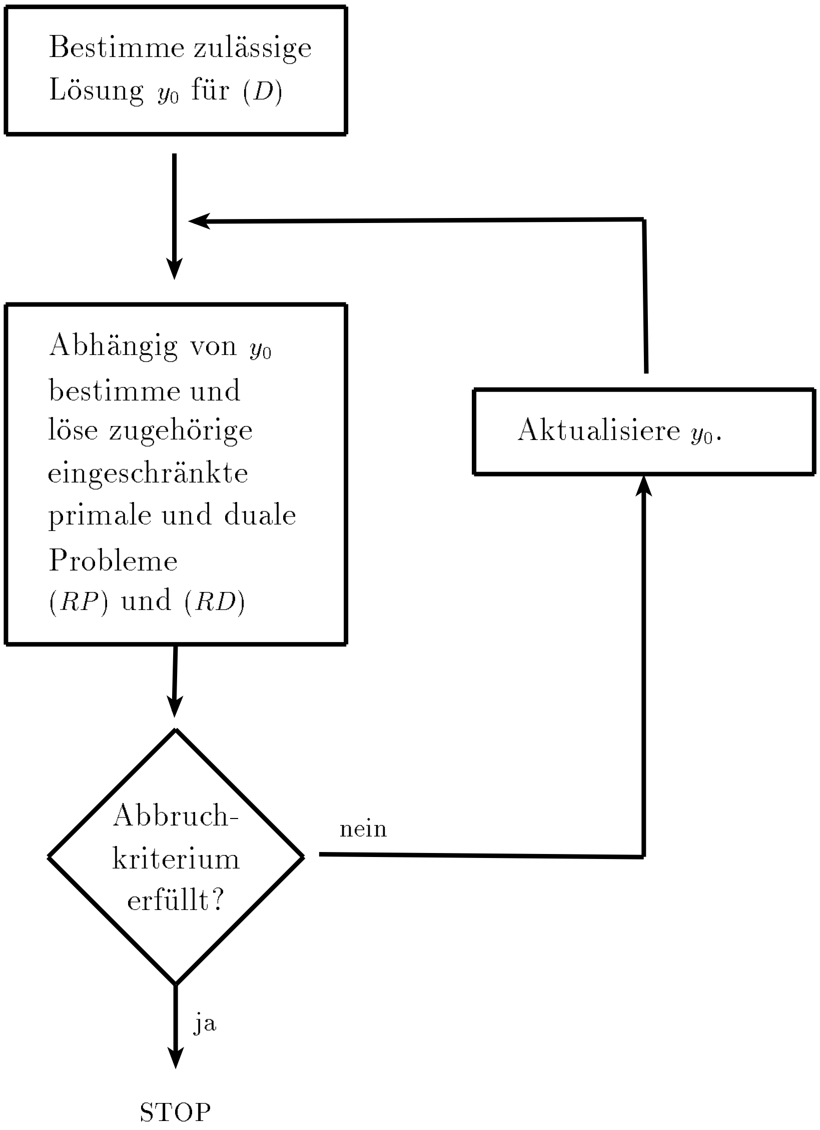


Abbildung 5.3: Schematische Darstellung der primal–dualen Methode.

Beobachtung: Ist y_0 zulässig für (D), (x_0, z_0) optimale Lösung von (RD), und ist $z_0 \equiv 0$, so gilt:
y_0 ist optimal für (D) und
x_0 ist optimal für (P).

Beweis: Offensichtlich sind x_0 und y_0 zulässig. Zudem sind die Bedingungen (5.1) des komplementären Schlupfs erfüllt.

Dies motiviert den in Abbildung 5.4 dargestellten Algorithmus.

Theorem 5.4 *Die primal–duale Methode ist korrekt und terminiert bei Verwendung der Indexstrategie von Bland in endlicher Zeit.*

Beweis:
Korrektheit:
Zu zeigen ist:

1. Falls $z_0 \equiv 0$, dann sind x_0 und y_0 optimal.
2. Falls $u_0^T A \leq 0$, dann ist (P) unlösbar.

Primal-duale Methode

GEGEBEN: zulässige Lösung y_0 für (D)
Wiederhole, bis eine der Abbruchbedingungen gegeben ist:

- Bestimme zugehörige eingeschränkte Probleme (RP) und (RD)
- Berechne optimale Lösung (x_0, z_0) von (RP)
- Falls $z_0 = 0 \quad \longrightarrow \quad$ STOP: x_0 und y_0 sind optimal
- Berechne optimale Lösung u_0 von (RD)
- Falls $u_0^T A \leq 0 \quad \longrightarrow \quad$ STOP: (P) ist unlösbar
- Bestimme neue dual zulässige Lösung:

$$y_0 \leftarrow y_0 + \epsilon_0 u_0,$$

wobei $\epsilon_0 := \min\limits_j \{ \frac{c_j - y_0^T A_{\cdot j}}{u_0^T A_{\cdot j}} \mid u_0^T A_{\cdot j} > 0 \}$

Abbildung 5.4: Primal–duale Methode.

3. Das aktualisierte y_0 ist wieder dual zulässig.

Ad 1. $\surd$ (s. o.)

Ad 2. Da y_0 dual zulässig ist, gilt $y_0^T A \leq c^T$.

$$\Rightarrow (y_0 + \lambda u_0)^T A = y_0 A + \lambda \underbrace{u_0^T A}_{\leq 0} \leq c^T \qquad \forall \lambda > 0$$

D. h. $y_0 + \lambda u_0$ ist dual zulässig für alle $\lambda > 0$.

Betrachte nun den Zielfunktionswert:

$$\Rightarrow (y_0 + \lambda u_0)^T b = y_0 b + \lambda \underbrace{u_0^T b}_{> 0} \xrightarrow{\lambda \to \infty} \infty$$

$u_0^T b$ ist der Wert der optimalen Lösung von (RD) und damit gleich dem Wert der optimalen Lösung von (RP). Dieser ist positiv da $z_0 \not\equiv 0$.

D. h. (D) ist unbeschränkt $\Rightarrow (P)$ ist unlösbar.

Ad 3. Wir zeigen $(y_0 + \lambda_0 u_0)^T A \leq c^T$. Betrachte die j–te Komponente:

$$\begin{aligned} & (y_0 + \lambda_0 u_0)^T A_{.j} \leq c_j \\ \Leftrightarrow \quad & \lambda_0 u_0^T A_{.j} \leq c_j - y_0^T A_{.j} \quad \surd \quad \text{[Erfüllt nach Konstruktion von } \lambda_0.] \end{aligned}$$

Terminierung:
Es gilt: Jede optimale Lösung (x_0, z_0) des alten (RP)–Problems ist zumindest eine zulässige Lösung des neuen (RP)–Problems. [Nachrechnen.]

Also terminiert der primal–duale Algorithmus nach endlich vielen Iterationen, falls eine geeignete Implementierung des Simplexalgorithmus verwendet wird ($\rightsquigarrow$ Blands Regel).

$\blacksquare$

Literaturverzeichnis

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[Meh84] Kurt Mehlhorn. *Data structures and algorithms 2: Graph algorithms and NP-Completeness.* Springer-Verlag, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong, 1984. EATCS Monographs on Theoretical Computer Science.

[NI92] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Disc. Math.*, 5(1):54–66, 1992.

[SW94] M. Stoer and F. Wagner. A simple min cut algorithm. In *Proceedings European Symposium on Algorithms ESA '94*, Lecture Notes in Computer Science 855, pages 141–147, 1994.